

CS162

Operating Systems and Systems Programming

Lecture 5

Semaphores, Conditional Variables

September 18, 2013

Anthony D. Joseph and John Canny

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

Goals for Today

- Atomic instruction sequence
- Continue with Synchronization Abstractions
 - Semaphores, Monitors and condition variables

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiawicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.2

Atomic Read-Modify-Write instructions

- Problems with interrupt-based lock solution:
 - Can't give lock implementation to users
 - Doesn't work well on multiprocessor
 - » Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
 - These instructions read a value from memory and write a new value atomically
 - Hardware is responsible for implementing this correctly
 - » on both uniprocessors (not too hard)
 - » and multiprocessors (requires help from cache coherence protocol)
 - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.3

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;
 }
}`

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.4

Implementing Locks with test&set

- Simple solution:

```
int value = 0; // Free
Acquire() {
    while (test&set(value));
}
Release() {
    value = 0;
}
```

```
test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```

- Simple explanation:

- If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits
- If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
- When we set value = 0, someone else can get lock

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.5

Problem: Busy-Waiting for Lock

- Positives for this solution

- Machine can receive interrupts
- User code can use this lock
- Works on a multiprocessor



- Negatives

- Inefficient: busy-waiting thread will consume cycles waiting
- Waiting thread may take cycles away from thread holding lock!
- **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock \Rightarrow no progress!

- Priority Inversion problem with original Martian rover

- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!

- Even if OK for locks, definitely not ok for other primitives
- Homework/exam solutions should not have busy-waiting!

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.6

Better Locks using test&set

- Can we build test&set locks without busy-waiting?

- Can't entirely, but can minimize!
- 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
- Why can't we do it just before or just after the sleep?

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.7

Locks using test&set vs. Interrupts

- Compare to “disable interrupt” solution (last lecture)

```
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
        Place on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

- Basically replace

- **disable interrupts** \rightarrow **while (test&set(guard));**
- **enable interrupts** \rightarrow **guard = 0;**

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.8

Recap: Locks

```
lock.Acquire();
...
critical section;
...
lock.Release();
```

```
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}
```

```
int value = 0;
Acquire() {
    // Short busy-wait time
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() ???
    } else {
        value = 1;
        enable interrupts;
    }
}

Release() {
    // Short busy-wait time
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.9

Recap: Locks

```
lock.Acquire();
...
critical section;
...
lock.Release();
```

```
Acquire() {
    while(test&set(guard));
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() & guard = 0;
    } else {
        value = 1;
        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 = 0;
    }
    guard = 0;
}
```

Threads waiting to enter critical section busy-wait

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.10

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

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.11

Semaphores



- Semaphores are a kind of generalized locks
 - 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
 - Think of this as the wait() operation
 - V()**: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
 - Think of this as the signal() operation
- Note that **P()** stands for "*proberen*" (to test) and **V()** stands for "*verhogen*" (to increment) in Dutch

9/18/13

Anthony D. Joseph and John Canny

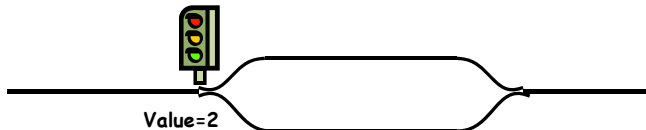
CS162

©UCB Fall 2013

Lec 5.12

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
 - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
 - Here is a semaphore initialized to 2 for resource control:



9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.13

Two Uses of Semaphores

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

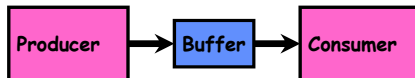

```
semaphore.P();
// Critical section goes here
semaphore.V();
```
- Scheduling Constraints (initial value = 0)
 - Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 **schedules** thread 1 when a given **constrained** is satisfied
 - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

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

9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.14

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
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
 - 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

9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.15

Correctness constraints for solution

- Correctness Constraints:
 - Consumer must wait for producer to fill slots, if empty (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)
- 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

9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.16

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;
}
```

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.17

Discussion about Solution

• Why asymmetry?

- Producer does: emptySlots.P(), fullSlots.V()
- Consumer does: fullSlots.P(), emptySlots.V()

Decrease # of empty slots

Increase # of occupied slots

Decrease # of occupied slots

Increase # of empty slots

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.18

Discussion about Solution

- Is order of P's important?
- Is order of V's important?
 - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?

```
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;
}
```

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.19

5min Break

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.20

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

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.21

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
 - Most others use actual locks and condition variables

9/18/13

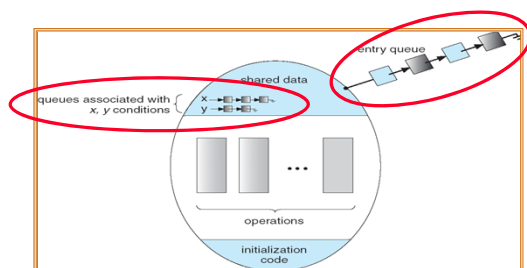
Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.22

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

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.23

Simple Monitor Example

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;
```

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

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

- Not very interesting use of “Monitor”
 - It only uses a lock with no condition variables
 - Cannot put consumer to sleep if no work!

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.24

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!

9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.25

Complete Monitor Example (with condition variable)

- 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);
}
  
```

9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.26

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:


```

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

 - Why didn't we do this?

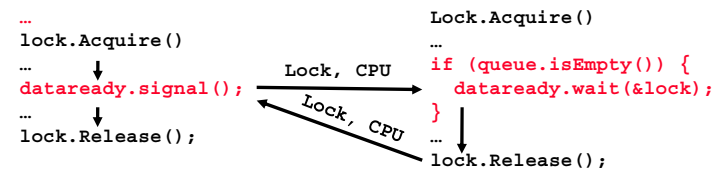

```

if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
          
```
- Answer: depends on the type of scheduling
 - Hoare-style
 - Mesa-style

9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.27

Hoare monitors

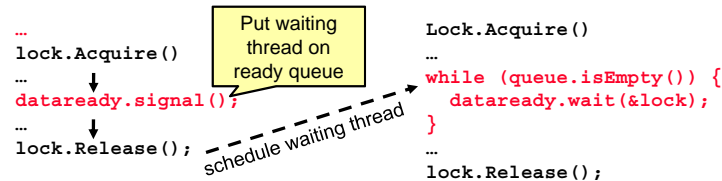
- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
- Most textbooks



9/18/13 Anthony D. Joseph and John Canny CS162 ©UCB Fall 2013 Lec 5.28

Mesa monitors

- Signaler keeps lock and processor
- Waiter placed on a local “e” queue for the monitor
- **Practically, need to check condition again after wait**
- Most real operating systems



9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.29

Mesa monitors – lock transfer

Q: How do the scheduled threads get a lock on the monitor when they restart?

A: At every exit from the monitor, and the end of every wait call where there would normally be a Release, there is a call to “schedule”: which does a Release or transfer.

```

...
lock.Acquire()
...
dataready.signal();
...
lock.Release();
    
```

```

wait() {
    add this thread to this.queue;
    schedule();
    sleep();
}
    
```

```

schedule() {
    if there is a thread in e
        select and remove one thread from e and restart it
    else
        lock.Release()
    
```

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.30

Mesa monitors – lock transfer

Q: How do the scheduled threads get a lock on the monitor when they restart?

A: At every exit from the monitor, and the end of every wait call where there would normally be a Release, there is a call to “schedule”: which does a Release or transfer.

```

...
lock.Acquire()
...
dataready.signal();
...
schedule();
    
```

```

wait() {
    add this thread to this.queue;
    schedule();
    sleep();
}
    
```

```

schedule() {
    if there is a thread in e
        select and remove one thread from e and restart it
    else
        lock.Release()
    
```

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.31

Mesa monitors – lock transfer

Q: How do the scheduled threads get a lock on the monitor when they restart?

A: At every exit from the monitor, and the end of every wait call where there would normally be a Release, there is a call to “schedule”: which does a Release or transfer.

```

...
lock.Acquire()
...
dataready.signal();
...
schedule();
    
```

```

wait() {
    add this thread to this.queue;
    schedule();
    sleep();
}
    
```

```

schedule() {
    if there is a thread in e
        select and remove one thread from e and restart it
    else
        lock.Release()
    
```

XFER

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.32

Mesa monitors – lock transfer

Q: How do the scheduled threads get a lock on the monitor when they restart?

A: At every exit from the monitor, and the end of every wait call where there would normally be a Release, there is a call to “schedule”: which does a Release or transfer.

```
... lock.Acquire()
    ↓
... dataready.signal();
    ↓
... schedule();

wait() {
    add this thread to this.queue;
    schedule();
    sleep();
}

schedule() {
    if there is a thread in e
        select and remove one thread from e and restart it
    else
        lock.Release()

```

Release

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.33

Summary

- Locks construction based on atomic seq. of instructions
 - Must be very careful not to waste/tie up machine resources
 - » Shouldn't spin wait for long
 - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Semaphores
 - Generalized locks
 - Two operations: **P()**, **V()**
- Monitors: A synchronous object 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()**

9/18/13

Anthony D. Joseph and John Canny

CS162

©UCB Fall 2013

Lec 5.34