

CS162 Operating Systems and Systems Programming Lecture 5

Semaphores, Conditional Variables

February 2, 2011
Ion Stoica
<http://inst.eecs.berkeley.edu/~cs162>

Goals for Today

- Continue with Synchronization Abstractions
 - Monitors and condition variables
- Readers-Writers problem and solution
- Language Support for Synchronization
- Tips for Programming in a Project Team

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiawicz.

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.2

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

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

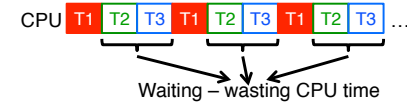
Lec 5.3

Review: Implementing Locks with test&set

- Simple solution:

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

- Advantage:
 - Simple
- Disadvantage:
 - **Busy-wait** until previous thread exists critical section
- Example: thread T1 in critical section, T2, T3 waiting to enter



2/2/11


Ion Stoica CS162 ©UCB Spring 2011

Lec 5.4

Review: 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;
}
```

- Previous example: T2, T3 suspended

2/2/11


Ion Stoica CS162 ©UCB Spring 2011

Lec 5.5

Review: 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

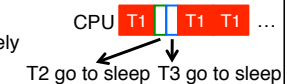
```
int guard = 0;
int value = FREE;
```



```
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;
}
```

- Example: T2 and T3 go to sleep immediately



2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.6

Higher-level Primitives than Locks

- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
 - This lecture and the next presents a couple of ways of structuring the sharing
- Good primitives and practices important!
 - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
 - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.7

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
 - » 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

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.8

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:



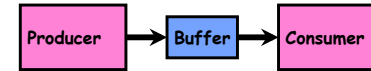
2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.9

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 1: GCC compiler
 - `cpp | cc1 | cc2 | as | ld`
- Example 2: Coke machine
 - Producer can put limited number of cokes in machine
 - Consumer can't take cokes out if machine is empty



2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.10

Correctness constraints for solution

- Correctness Constraints:
 - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
 - Producer must wait for consumer to empty buffers, 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 `fullBuffers`; // consumer's constraint
 - Semaphore `emptyBuffers`; // producer's constraint
 - Semaphore `mutex`; // mutual exclusion

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.11

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptyBuffers.P(); // Wait until space
    mutex.P(); // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V(); // Tell consumers there is
                    // more coke
}

Consumer() {
    fullBuffers.P(); // Check if there's a coke
    mutex.P(); // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
```

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.12

Discussion about Solution

- Why asymmetry?
 - Producer does: `emptyBuffer.P()`, `fullBuffer.V()`
 - Consumer does: `fullBuffer.P()`, `emptyBuffer.V()`
- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?
 - Do we need to change anything?

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.13

Motivation for Monitors and Condition Variables

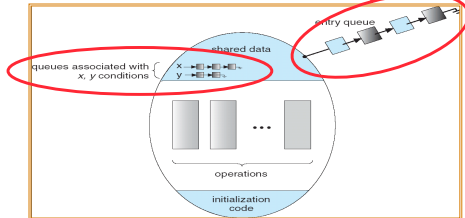
- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
 - Problem is that semaphores are dual purpose:
 - » They are used for both mutex and scheduling constraints
 - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- 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

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.14

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

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.15

Simple Monitor Example (version 1)

- 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!

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.16

Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
 - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- **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!
 - In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.17

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);
}
    
```

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.18

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
    
```

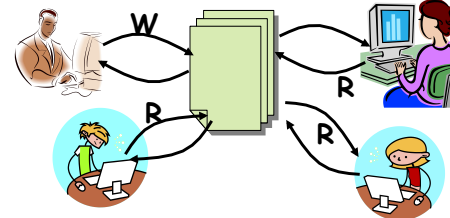
- Answer: depends on the type of scheduling
 - Hoare-style (most textbooks):
 - » Signaler gives lock, CPU to waiter; waiter runs immediately
 - » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
 - Mesa-style (most real operating systems):
 - » Signaler keeps lock and processor
 - » Waiter placed on ready queue with no special priority
 - » **Practically, need to check condition again after wait**

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.19

Readers/Writers Problem



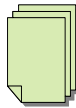
- Motivation: Consider a shared database
 - Two classes of users:
 - » Readers – never modify database
 - » Writers – read and modify database
 - Is using a single lock on the whole database sufficient?
 - » Like to have many readers at the same time
 - » Only one writer at a time

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.20

Basic Readers/Writers Solution



- Correctness Constraints:
 - Readers can access database when no writers
 - Writers can access database when no readers or writers
 - Only one thread manipulates state variables at a time
- Basic structure of a solution:
 - Reader()
 - Wait until no writers
 - Access data base
 - Check out - wake up a waiting writer
 - Writer()
 - Wait until no active readers or writers
 - Access database
 - Check out - wake up waiting readers or writer
 - State variables (Protected by a lock called "lock"):
 - » int AR: Number of active readers; initially = 0
 - » int WR: Number of waiting readers; initially = 0
 - » int AW: Number of active writers; initially = 0
 - » int WW: Number of waiting writers; initially = 0
 - » Condition okToRead = NIL
 - » Condition okToWrite = NIL

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.21

Code for a Reader

```
Reader() {
    // First check self into system
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    lock.release();
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Acquire();
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        okToWrite.signal(); // Wake up one writer
    lock.Release();
}
```

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.22

Code for a Writer

```
Writer() {
    // First check self into system
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--; // No longer active
    if (WW > 0) { // Give priority to writers
        okToWrite.signal(); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        okToRead.broadcast(); // Wake all readers
    }
    lock.Release();
}
```

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.23

Administrivia

- All groups have been assigned
 - We believe we satisfied all your constraints!
- Project 1
 - Out: Today by midnight
 - Design due: Tuesday, February 14
 - Code due: Tuesday, March 1
 - Final design document: March 2

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.24

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
 - R1, R2, W1, R3
- On entry, each reader checks the following:


```
while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                // No longer waiting
}
AR++;                  // Now we are active!
```
- First, R1 comes along:
 - AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
 - AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers may take a while to access database
 - Situation: Locks released
 - Only AR is non-zero

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.25

Simulation(2)

- Next, W1 comes along:


```
while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                // No. Active users exist
    okToWrite.wait(&lock); // Sleep on cond var
    WW--;                // No longer waiting
}
AW++;
```
- Can't start because of readers, so go to sleep:
 - AR = 2, WR = 0, AW = 0, WW = 1
- Finally, R3 comes along:
 - AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
 - AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:


```
if (AR == 0 && WW > 0) // No other active readers
    okToWrite.signal(); // Wake up one writer
```

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.26

Simulation(3)

- When writer wakes up, get:
 - AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:


```
if (WW > 0) { // Give priority to writers
    okToWrite.signal(); // Wake up one writer
} else if (WR > 0) { // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
}
```

 - Writer wakes up reader, so get:
 - AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.27

Questions

- Can readers starve? Consider Reader() entry code:


```
while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                // No longer waiting
}
AR++;                  // Now we are active!
```
- What if we erase the condition check in Reader exit?


```
AR--;                // No longer active
if (AR == 0 && WW > 0) // No other active readers
    okToWrite.signal(); // Wake up one writer
```
- Further, what if we turn the signal() into broadcast()


```
AR--;                // No longer active
okToWrite.broadcast(); // Wake up one writer
```
- Finally, what if we use only one condition variable (call it "okToContinue") instead of two separate ones?
 - Both readers and writers sleep on this variable
 - Must use broadcast() instead of signal()

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.28

Tips for Programming in a Project Team



"You just have to get your synchronization right!"

- Big projects require more than one person (or long, long, long time)
 - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
 - Doesn't seem to be as true of big construction projects
 - » Empire state building finished in **one** year: staging iron production thousands of miles away
 - » Or the Hoover dam: built towns to hold workers

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.29

Big Projects

- What is a big project?
 - Time/work estimation is hard
 - Programmers are eternal optimists (it will only take two days!)
 - » This is why we bug you about starting the project early
- Can a project be efficiently partitioned?
 - Partitionable task decreases in time as you add people
 - But, if you require communication:
 - » Time reaches a minimum bound
 - » With complex interactions, time increases!
 - Mythical person-month problem:
 - » You estimate how long a project will take
 - » Starts to fall behind, so you add more people
 - » Project takes even more time!



2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.30

Techniques for Partitioning Tasks

- Functional
 - Person A implements threads, Person B implements semaphores, Person C implements locks...
 - Problem: Lots of communication across APIs
 - » If B changes the API, A may need to make changes
 - » Story: Large airline company spent \$200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another \$200 million to fix.
- 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 CS162 project teams are functional, but people have had success with task-based divisions

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.31

Communication

- More people mean more communication
 - Changes have to be propagated to more people
 - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
 - "Index starts at 0? I thought you said 1!"
- Who makes decisions?
 - Individual decisions are fast but trouble
 - Group decisions take time
 - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
 - Better not be clueless
 - Better have good people skills
 - Better let other people do work



2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.32

Coordination



- More people \Rightarrow no one can make all meetings!
 - They miss decisions and associated discussion
 - Example from earlier class: one person missed meetings and did something group had rejected
 - Why do we limit groups to 5 people?
 - » You would never be able to schedule meetings otherwise
 - Why do we require 4 people minimum?
 - » You need to experience groups to get ready for real world
- People have different work styles
 - Some people work in the morning, some at night
 - How do you decide when to meet or work together?
- What about project slippage?
 - It will happen, guaranteed!
 - Ex: everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
 - Members have already figured out how to work together

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.33

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()`
- Readers/Writers
 - Readers can access database when no writers
 - Writers can access database when no readers
 - Only one thread manipulates state variables at a time
- Language support for synchronization:
 - Java provides `synchronized` keyword and one condition-variable per object (with `wait()` and `notify()`)

2/2/11

Ion Stoica CS162 ©UCB Spring 2011

Lec 5.34