Lecture 10: Synchronization
Announcements

- Project 1 due Friday
  - TAs will send out instructions for how to submit

- Mid-term coming up soon
  - Feb 8, Monday.
  - Two review sessions before then
Readers/Writers Problem

- Go back to Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers but only one writer
    - Let \( r \) be the number of readers, \( w \) be the number of writers
    - Safety: \((r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \Rightarrow (w = 0))\)

- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading
Readers/Writers

// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    Read;
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
}
Readers/Writers

// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    wait(mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex); // unlock readcount
    Read;
    wait(mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex); // unlock readcount
}
Readers/Writers Notes

- `w_or_r` provides mutex between readers and writers
  - Readers wait/signal when `readcount` goes from 0 to 1 or 1 to 0
- If a writer is writing, where will readers be waiting?
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
  - Is it guaranteed that all readers will fall through?
- If readers and writers are waiting, and a writer exits, who goes first?
- If read in progress when writer arrives, when can writer get access?
- In Java:
  - `readWriteLock.readLock().lock()`
  - `readWriterLock.writeLock().lock()`
Semaphore Summary

- Semaphores can be used to solve any of the traditional synchronization problems
- However, they have some drawbacks
  - They are essentially shared global variables
    » Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
    » Note that I had to use comments in the code to distinguish
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support
Java Synchronization Support

Object foo; // shared across threads
synchronized (foo) {
    /* Do some stuff with 'foo' locked... */
    foo.counter++;
}

Compiler ensures that lock is released before leaving the synchronized block --- Even if there is an exception!!

try {
    synchronized(foo) {
        if (foo.doSomething() == false)
            throw new Exception("Bad!!");
    }
    catch (Exception e) {
        /* Lock was released before getting here! */
        System.err.println("Something bad happened!");
    }
}
Condition Variables

- **Main idea:**
  - make it possible for thread to sleep inside a critical section

- **Approach:**
  - by atomically releasing lock, putting thread on wait queue and sleep

- Each variable has a queue of waiting threads
  - threads that are sleeping, waiting for a condition

- Each variable is associated with one lock
Condition Variables in Java

- All condition variable operations must be within a synchronized block on the same object

```java
/* Thread A */
synchronized (foo) {
    while (foo.counter < 10) {
        foo.wait();
    }
}

/* Thread B */
synchronized (foo) {
    foo.counter++;
    if (foo.counter >= 10) {
        foo.notify();
    }
}
```

- Why is the “synchronized” necessary?
Condition Vars != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics
  - However, they each can be used to implement the other

- Condition variable is protected by a lock
  - `wait()` blocks the calling thread, and gives up the lock
    » To call `wait`, the thread has to be in the monitor (hence has lock)
    » `Semaphore::wait` just blocks the thread on the queue
  - `signal()` causes a waiting thread to wake up
    » If there is no waiting thread, the signal is lost
    » `Semaphore::signal` increases the semaphore count, allowing future entry even if no thread is waiting
    » Condition variables have no history
Monitor

- monitor = a lock + the condition variables associated with that lock
- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - A thread enters an object’s monitor by
    - Executing a method declared “synchronized”
      - Can mix synchronized/unsynchronized methods in same class
    - Executing the body of a “synchronized” statement
      - Supports finer-grained locking than an entire procedure
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()
There are two flavors of monitors that differ in the scheduling semantics of signal():

- **Hoare** monitors (original)
  - signal() immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling

- **Mesa** monitors (Mesa, Java)
  - signal() places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter runs again
    - Returning from wait() is only a hint that something changed
    - Must recheck conditional case