CS193j, Stanford
Winter, 2001-02

Handout #22
Nick Parlante

Threading 4

Co-operation

Synchronization is the first order problem with concurrency. The second problem is cooperation -- getting multiple threads to exchange information.

Checking condition under lock

Suppose you want to execute the statment "if (len >0) len++;" but other threads also operate on len.

Acquire the lock first, then look at len -- otherwise some other thread may change len in between the test and the len++

Do operations with the lock so the data is not changing out from under you -- this is just a basic truism of threads that read and write shared data.

wait() and notify()

Every Object has a wait/notify queue

You must have that object's lock first before doing any operation on the queue (the queue is like "len" in the above example)

Use the wait/notify queue coordinate the actions of threads -- get them to cooperate, signal each other

wait()

obj.wait();

Send to any object -- wait on its queue

"Suspend" on that object's queue -- efficient blocking

Must first have that object's lock (or get a runtime error)

The waiting thread releases that object's lock (but not other held locks)

interrupt() will pop the thread out of wait()

notify

obj.notify();

Send to any object -- notifies waiters on that object's queue

The sender must first have the object lock

A random waiting thread will get woken out of its wait() when the sender releases the lock. Not necessarily FIFO. Not right away.

The wait will re-acquire the lock before resuming

"dropped" notify

- if there are no waiting threads, the notify does nothing
- wait()/notify() **does not count up and down** to balance things -- you need to build a Semaphore for that feature

variant: notifyAll() notifies all the waiting threads, not just a single one

barging / Check again

When coming out of a wait(), check for the desired condition again -- it may have become false again in between when the notify happened and when the wait/return happened.

while

Essentially, the wait is always done with a while loop, not an if statement.

1. AddRemove

```
/*
 Producer/Consumer problem with wait/notify
 This code works correctly.

 -"len" represents the number of elements in some imaginary array
 -add() adds an element to the end of the array. Add() never blocks --
 we assume there's enough space in the array.
 -remove() removes an element, but can only finish if there
 is an element to be removed. If there is no element, remove()
 waits for one to be available.

 Strategy:
 -The AddRemove object is the common object between the threads --
 they use its lock and its wait/notify queue.
 -add() does a notify() when it adds an element
 -remove() does a wait() if there are no elements. Eventually,
 an add() thread will put an element in and do a notify()
 -Each adder adds 10 times, and each remover removes 10 times,
 so it balances in the end.
*/
class AddRemove {
   int len = 0;   // the number of elements in the array
   final int MAX = 10;

   public synchronized void add() {
      len++;
      System.out.println("Add elem " + (len-1));
      notify();
   }

   public synchronized void remove() {
      // If there is no element available, we wait.
      // We must check the condition again coming out
      // of the wait because of "barging" (use while instead of if)
      while (len == 0) {
         try{ wait();} catch (InterruptedException ignored) {}
      }
      // At this point, we have the lock and len>0
      System.out.println("Remove elem " + (len-1));
      len--;
   }
```

```
private class Adder extends Thread {
   public void run() {
      for (int i = 0; i< MAX; i++) {
         add();
         yield(); // this just gets the threads to switch around more,
                  // so the output is a little more interesting
      }
   }
}

private class Remover extends Thread {
   public void run() {
      for (int i = 0; i< MAX; i++) {
         remove();
         yield();
      }
      System.out.println("done");
   }
}

public void demo() {

   // Make two "adding" threads
   Thread a1 = new Adder();
   Thread a2 = new Adder();


   // Make two "removing" threads
   Thread r1 = new Remover();
   Thread r2 = new Remover();

   // start them up (any order would work)
   a1.start();
   a2.start();
   r1.start();
   r2.start();

   /*
   output
      Add elem 0
      Add elem 1
      Remove elem 1
      Add elem 1
      Add elem 2
      Add elem 3
      Remove elem 3
      Remove elem 2
      Add elem 2
      Add elem 3
      Remove elem 3
      Remove elem 2
      Add elem 2
      ...
      Remove elem 3
      Remove elem 2
      done
      Remove elem 1
      Remove elem 0
```

```
         done
      */

   }

   public static void main(String args[]) {
      new AddRemove().demo();
   }
}
```

2. WaitDemo

```
/**
 Demonstrates the "dropped notify" problem.
 Have one thread generate 10 notifies for use by another thread.
 Does not work because of the "dropped notify" problem.
*/
class WaitDemo {
   // The shared point of contact between the two
   Object shared = new Object();

   // Collect 10 notifications on the shared object
   class Waiter extends Thread {
      public void run() {
         for (int i = 0; i<10; i++) {
            try {
               synchronized(shared) {
                  shared.wait();
               }
            } catch (InterruptedException ingored) {}
         }
         System.out.println("Waiter done");  // it never gets to this line
      }
   }

   // Do 10 notifications on the shared object
   class Notifier extends Thread {
      public void run() {
         for (int i = 0; i<10; i++) {
            synchronized(shared) {
               shared.notify();
            }
         }
         System.out.println("Notifier done");
      }
   }


   public void demo() {
      new Waiter().start();
      new Notifier().start();
   }

   public static void main(String[] args) {
      new WaitDemo().demo();
   }
}
```