CS193k, Stanford
Spring, 99-00

Handout #9
Nick Parlante

Threads 3

Checking on condition

Need lock

if (len>0) {
   // len may be 0 at this point

Lock

Check and respond all under the lock so things aren't changing out from under you

wait() and notify()

Every Object has a wait/notify queue
You must have that object's lock first
Use to coordinate the actions of threads -- get them to cooperate, signal each other

wait()

"suspend" on that object's queue
Automatically releases that object's lock (but not other held locks)
interrupt() will pop you out of wait()

notify

must have the lock
a random waiting thread will get woken out of its wait() when you release the lock. Not necessarily FIFO. Not right away.

"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

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.

join

Wait for finish

We block until the receiver thread exits its run(). Use this to wait for another thread to finish. The current thread is blocked. The receiver object is the one we wait for.

Code

t = new ThreadSubclass();
t.start(); // fork off worker
// do something else
t.join(); // wait for worker to finish

Idiom

Frequenly I have some sort of main thread that sets things up, launches a bunch of threads to do something, and then joins them all to wait for them to finish. Then it does some sort of final cleanup.

Code Examples:

1. 10 notifies -> 10 waits NO
2. Take turns w/ wait/notify NO
3. Reader/Writer w/ len OK
4. Semaphore OK
5). Takes turns w/ Semaphore OK

Code

```
/*
 Have one thread generate 10 notifies for use by another thread.
 Does not work because of the "dropped notify" problem.
*/
class WaitDemo {
   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 {
               sleep(1);
               synchronized(shared) {
                  shared.wait();
               }
            } catch (InterruptedException ingored) {}
         }
         System.out.println("Waiter done");
      }
   }

   // Do 10 notifications on the shared object
   class Notifier extends Thread {
      public void run() {
         for (int i = 0; i<10; i++) {
            try {sleep(1);} catch (InterruptedException ignored) {}
            synchronized(shared) {
               shared.notify();
            }
         }
         System.out.println("Notifier done");
      }
   }

   public void demo() {
      for (int i = 0; i< 10; i++) {
         new Waiter().start();
         new Notifier().start();
      }
   }
}


/*
 Try to get A and B to alternate -- does not work because
 of the dropped notify problem. Sometimes, A gets one turn and then
 everything locks up.
*/
class TurnDemo {

   synchronized void a() {
      System.out.println("It's A turn, A rules!");
      notify();
      try{ wait();} catch (InterruptedException ignored) {}
   }

   synchronized void b() {
      try{ wait();} catch (InterruptedException ignored) {}
      System.out.println("It's B turn, B rules!");
      notify();
   }

   public void demo() {

      // Fork off a thread to call a() 10x
      Thread a = new Thread() {
         public void run() {
            for (int i = 0; i< 10; i++) { a(); }
         }
      };
      a.start();

      // Fork off a thread to call b() 10x
      Thread b = new Thread() {
         public void run() {
            for (int i = 0; i< 10; i++) { b(); }
         }
      };
      b.start();

      try{
         a.join();
         b.join();
      }
      catch (InterruptedException ignored) {}
      System.out.println("All done!");
   }
}


/*
 Reader/Writer
 This code works.
 This is the "unbound" version where the writer can always write.
 This works because the "len" variable essentially keeps track of dropped notifies()
--
 it's ok if the writer gets ahead of the reader.
 Note the classic use of "while" in the read() -- to guard against
 "barging":
 r1 blocks in read()
 w1 does a write and notify
 r2 comes through and reads() (this is the barging step)
 r1 finally returns from its wait, but r2 took it
*/
class ReaderWriter {
   int len = 0;

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

   public synchronized void read() {
      //if (len == 0)   // Would not work because of "barging"
      while (len == 0) {
         try{ wait();} catch (InterruptedException ignored) {}
      }
      // At this point, we have the lock and len>0
      System.out.println("Read elem " + (len-1));
      len--;
   }

   public void demo() {

      Thread w1 = new Thread() {
         public void run() {
            for (int i = 0; i< 100; i++) {write(); yield();}
         }
      };

      Thread w2 = new Thread() {
         public void run() {
            for (int i = 0; i< 100; i++) {write(); yield();}
         }
      };

      Thread r1 = new Thread() {
         public void run() {
            for (int i = 0; i< 100; i++) {read(); yield();}
            System.out.println("done");
         }
      };

      Thread r2 = new Thread() {
         public void run() {
            for (int i = 0; i< 100; i++) {read(); yield();}
            System.out.println("done");
         }

      };

      r1.start();
      r2.start();
      w1.start();
      w2.start();
   }
}


/*
 The classic counting semaphore.
 SAVE THIS CODE
 Count>0 represents "available".
 Count==0 means the locks are all in use.
 Count<0 represents the number of threads waiting for the lock.
 A client should decr() to acquire, work, and then incr() to release.
 If the semaphore was not available, decr() will block until it is.
 Note:
 (1) There is no barging since the decr() thread waits if ANY
 other thread is in line.
 (2) The "if" in decr() does not need to be a "while". The notify() in incr()
 _always_ has a matching wait -- the negative count keeps track that there is
someone
 waiting.
 (3) If a thread blocks in a decr() it is still holding all its other locks --
 the wait() does not automatically release them.
*/
class Semaphore {
   private int count;

   public Semaphore(int value) {
      count = value;
   }

   public synchronized void decr() {
      count--;
      if (count<0) try{
         wait();
      }
      catch (InterruptedException inter) {
         // This exception clears the "isInterrupted" boolean.
         // We reset the boolean to true so our caller
         // will see that interruption has happened.
         Thread.currentThread().interrupt();
      }
   }
   public synchronized void incr() {
      count++;
      if (count<=0) notify();
   }
}

/*
 Use two Semaphores to get A and B to take turns.
 This code works.
*/
class TurnDemo2 {
   Semaphore aGo = new Semaphore(1);      // a gets to go first
   Semaphore bGo = new Semaphore(0);

   void a() {
      aGo.decr();
      synchronized (this) {
         System.out.println("It's A turn, A rules!");
      }
      bGo.incr();
   }

   void b() {
      bGo.decr();
      synchronized (this) {
         System.out.println("It's B turn, B rules!");
      }
      aGo.incr();
   }

   /*
    Q: Suppose a() and b() where synchronized -- what would happen?
    A: deadlock! a thread would take the "this" lock and block on its decr(),
    but the other thread could never get in to do the matching incr().
    That's why the synchronized is just around the minimal part --
    Get In and Get Out.
   */
   public void demo() {

      new Thread() {
         public void run() {
            for (int i = 0; i< 10; i++) {b(); }
         }
      }.start();

      new Thread() {
         public void run() {
            for (int i = 0; i< 10; i++) {a(); }
         }
      }.start();
   }
}
```

stop()/suspend() deprecated

The problem is that a thread could be in any state when this happens.

stop() causes the receiving thread to throw ThreadDeath all the way out -- releasing locks and leaving objects where they were.

e.g.

Thread is reading from one buffer and putting into another.

ThreadDeath comes along in mid transfer with the element in a stack variable, so it's just lost.

OSes -- turn off interrupts

OS's deal with this by turning off interrupts for sections of code that must complete -- like a transaction

interrupt()

t.interrupt();

Use interrupt() on a Thread object

Soon, this will set the "interrupted" boolean on that thread

Thread.currentThread().isInterrupted()

Thread should check isInterrupted() periodically to decide if it has been interrupted.

If so, it should cleanly back out, return false, etc.

Doing so will effectively release locks

wait(), sleep(), join()

The interrupt machinery will break out of these with InterruptedException

Blocking on a mutex is not backed out by an interrupt()

I/O -- may obey interrupt

Depending on the VM, I/O and other system calls may unblock on interrupt or not.

This is an area where the Java may be changing

InterruptedException clears bool

Unfortunately, when the system throws InterruptedException, it clears the boolean state, so subsequent checks of isInterrupted() will return false.

Therefore, catches of that exception may wish to Thread.currentThread().interrupt() to re-assert the boolean so code that's checking it will find it.

static boolean interrupted()

(consider never using this method)
use Thread.currentThread().isInterrupted() instead
Tests the current running thread and clears the interrupted state -- so it will only return true once.
Can be used where the code needs to detect an interruption, change its state in some way, and then be able to detect a second interruption.

interrupt() tradeoff

Pro: Clean

Interruption can leave objects in a clean state

Con: requires code

The code to be interrupted needs to check isInterrrupted() periodically and have some plan for how to cleanly get out.

Con: lag

There's lag in between when interrupt() is called and when the interruption effectively happens.

GUI Threading Theory

Launch button

Create a bunch of worker threads to do something
Create a ThreadGroup to collect the worker threads
Pass to Thread ctor

Worker Threads

1. Compute something
2. Send updates back to the Swing state using SwingUtilities.invokeLater(Runnable).
3. Check isInterrupted() now and then

Stop Button

```
/*
 The stop button has been clicked -- go interrupt all the threads.
 In 1.2 you can write threadGroup.interrupt().
 In 1.1 it needs to be written as below.
*/
public void userStop() {
   Thread threads[] = new Thread[100];
   int len = threadGroup.enumerate(threads);
   for (int i=0; i<len; i++) {
      threads[i].interrupt();
   }
}
```