CS193k, Stanford Handout #8
Spring, 2000-01 Nick Parlante

Threads 3

t.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

Frequently 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.

Basic Code Examples

Here are a couple complete examples that demonstrate the basic synchronization issues we've seen...

Thread1

```
/**
 Demonstrates classic synchronization --
 multiple threads simultaneously message a single object.

 The code works if incr() and otherIncr() are declared
 synchronized.
*/
class Thread1 {
   Inner inner;
   final int ITER = 1000000;

   private class Inner {
      int count = 0;

      // this needs to be synchronized
      public void incr() {
         count++; // this line is not atomic
      }

      // this also needs to be synchronized
      public void otherIncr() {
         count++;
      }

   }
   private class Worker extends Thread {
      public void run() {
         for (int i=0; i<ITER; i++) {
            inner.incr();
            inner.otherIncr();
         }
      }
   }


   public void demo() {
      inner = new Inner();

      // The workers send incr() and otherIncr() to inner
      Worker w1 = new Worker();
      Worker w2 = new Worker();

      w1.start();
      w2.start();

      // wait for the workers to finish
      try {
         w1.join();
         w2.join();
      }
      catch (InterruptedException ignored) {}

      System.out.println(inner.count);
   }

   public static void main(String args[]) {
      new Thread1().demo();
   }

}
```

Thread2

```
/**
 Demonstrates another form of error -- the objects
 are correctly synchronized for their own state,
 but there is simultaneous access to a (shared) static
 that causes problems.
*/
class Thread2 {
   static int shared = 0;

   final int ITER = 1000000;

   private class Inner {
      int count;

      public synchronized void incr() {
         count++;    // this is fine

         // this has simultaneous access problems
         shared++;
         // Could repair with a sharedLock object...
         //synchronized(sharedLock) {
         // shared++;
         //}
      }
   }

   private class Worker extends Thread {
      Inner myInner;

      public Worker(Inner inner) {
         myInner = inner;
      }

      public void run() {
         for (int i=0; i<ITER; i++) {
            myInner.incr();
         }
      }
   }


   public void demo() {
      Inner i1 = new Inner();
      Inner i2 = new Inner();

      // each worker has its own Inner object
      Worker w1 = new Worker(i1);
      Worker w2 = new Worker(i2);

      w1.start();
      w2.start();

      try {
         w1.join();
         w2.join();
      }
      catch (InterruptedException ignored) {}

      System.out.println(i1.count); // right
      System.out.println(i2.count); // right
      System.out.println(shared);   // wrong

      /*
       output
       10000000
       10000000
       19402932
      */
   }

   public static void main(String args[]) {
      new Thread2().demo();
   }

}
```

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

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.

Code Examples:

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

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
 -remove() removes an element, but can only finish if there
 is an element to be removed

 Strategy:
 -The AddRemove object is the common object between the threads
 -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()
*/
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 (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("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();
   }

}
```

3. TurnDemo

```
/**
 Another demonstration of the dropped-notify problem.
 Try to get A and B to alternate. 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!");
   }

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

}
```

4. Semaphore

```
/*
 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();
   }
}
```

5. TurnDemo2

```
/*
 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() {
        final Semaphore finished = new Semaphore(0);
        new Thread() {
            public void run() {
                for (int i = 0; i< 10; i++) {b(); }
                finished.incr();
            }
        }.start();

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


        finished.decr();
        finished.decr();
        System.out.println("All done!");
    }

    public static void main(String args[]) {
        new TurnDemo2().demo();
    }
}
```

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.