CS193j, Stanford
Summer, 2003

Handout #20
Manu Kumar

Threading 2

Two Threading Challenges

Mutual exclusion
    Keeping the threads from interfering with each other. Worry about memory shared by multiple threads.
Cooperation -- join/wait/notify
    Get threads to cooperate. Typically this centers on handing information from one thread to another, or signaling one thread that another thread has finished doing something.
    (we'll cover this a little later)

Race Condition / Critical Section

A section of code that causes problems if two or more threads are executing it at the same time
Typically the problem revolves around some shared memory that both threads are reading or writing at the same time
Establish "mutual exclusion" -- only one thread executes the critical section at a time

Race Condition Example

```
class Pair {
   private int a, b;

   public Pair() {
      a = 0;
      b = 0;
   }

   // Returns the sum of a and b. (reader)
   public int sum() {
      return(a+b);
   }

   // Increments both a and b. (writer)
   public void inc() {
      a++;
      b++;
   }
}
```

Reader/Writer conflict

e.g. thread1 runs inc() while thread2 runs sum() on the same object.
In that case, the sum() thread could get an incorrect value if the inc() is halfway done.
In part, this happens because the lines of inc() and sum() interleave

In fact, even the single statement a++ is not atomic, so the interleave also happens at a scale finer than a single java statement.

Java guarantees that 4-byte reads and writes are atomic, but that's it. The statement a++ actually expands to a three-step: read, increment, write.

Note that this is only a problem if the threads are executing against the same object so they both try to touch the same memory.

Writer/Writer conflict

e.g. thread1 runs inc() while thread2 runs inc() on the same object.

The two inc()'s can interleave to mess up the object state

a++ is not atomic -- it can interleave with another a++ to produce wrong results. This is true in most languages.

Random Interleave -- Hard to Observe

Race conditions depend on two or more threads "interleaving" their execution in just the right way to exhibit the bug. It happens rarely and randomly, but it happens.

The likelihood of the interleave is seemingly random -- depending on the system load and the number of processors.

You are more likely to observe the race condition problem on a system with multiple CPUs.

This is why locating concurrency bugs can be so hard -- they exhibit themselves sporadically.

Many of the bugs seen in shipping software are concurrency bugs -- they were not seen or tracked down in in-house testing.

Java Locks

Object Lock + Synchronized Method

In Java, every object has a "lock"

A "synchronized" method respects the lock of the receiver object.

For a thread to execute a synchronized method against a receiver, it first obtains the lock of the receiver.

The lock is released when the method exits

If the lock is already held by another thread, the calling thread blocks (efficiently) waiting for the other thread to exit and so make the lock available.

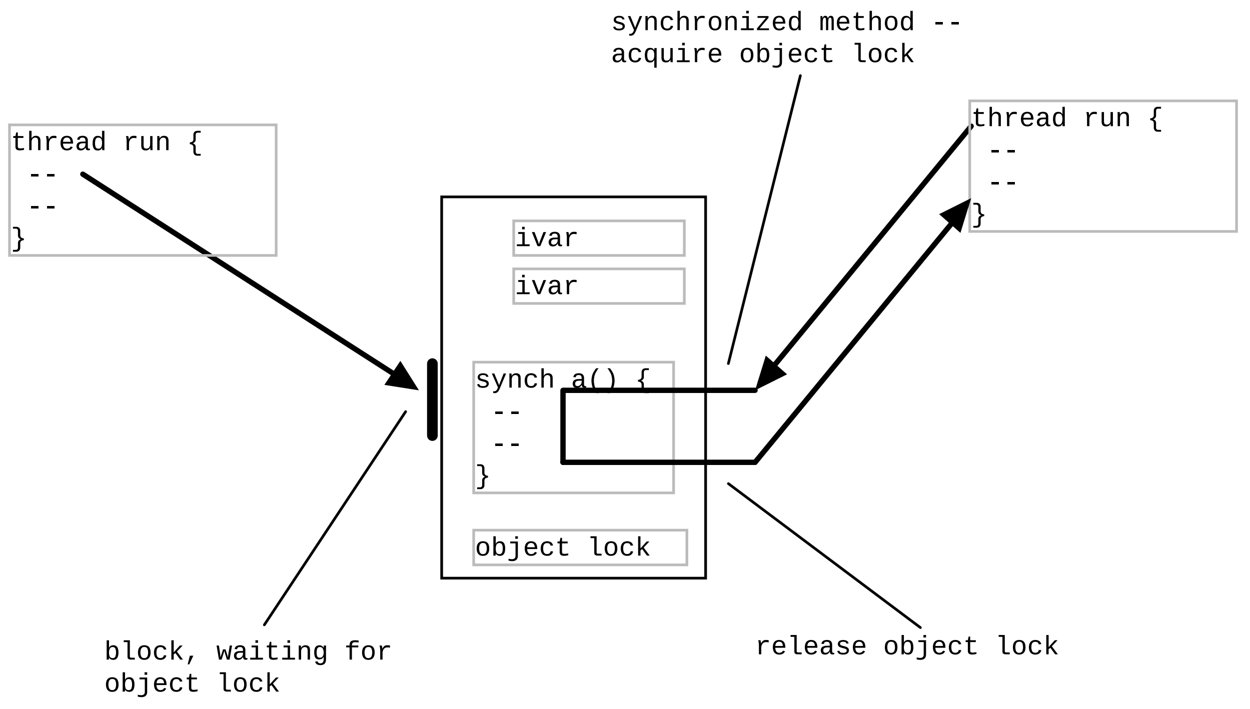
In this way, multiple threads, in effect, take turns on who can execute against the receiver.

Receiver Lock

The lock is in the receiver object -- it provides mutual exclusion for multiple threads sending messages to **that object**. Other objects have their own locks.

If a method is not synchronized, it will ignore the lock and just go ahead.

Synchronized Method Picture



Synchronized Method Example

```
/*
 A simple class that demonstrates using the 'synchronized'
 keyword so that multiple threads may send it messages.
 The class stores two ints, a and b; sum() returns
 their sum, and inc() increments both numbers.

 <p>
 The sum() and incr() methods are "critical sections" --
 they compute the wrong thing if run by multiple threads
 at the same time. The sum() and inc() methods are declared
 "synchronized" -- they respect the lock in the receiver object.
*/
class Pair {
   private int a, b;

   public Pair() {
      a = 0;
      b = 0;
   }

   // Returns the sum of a and b. (reader)
   // Should always return an even number.
   public synchronized int sum() {
      return(a+b);
   }

   // Increments both a and b. (writer)
   public synchronized void inc() {
      a++;
      b++;
   }
}




/*
 A simple worker subclass of Thread.
 In its run(), sends 1000 inc() messages
 to its Pair object.
*/
class PairWorker extends Thread {
   public final int COUNT = 1000;
   private Pair pair;

   // Ctor takes a pointer to the pair we use
   public PairWorker(Pair pair) {
      this.pair = pair;
   }

   // Send many inc() messages to our pair
   public void run() {
      for (int i=0; i<COUNT; i++) {
         pair.inc();
      }
   }


   /*
    Test main -- Create a Pair and 3 workers.
    Start the 3 workers -- they do their run() --
    and wait for the workers to finish.
   */
   public static void main(String args[]) {
      Pair pair = new Pair();
      PairWorker w1 = new PairWorker(pair);
      PairWorker w2 = new PairWorker(pair);
      PairWorker w3 = new PairWorker(pair);

      w1.start();
      w2.start();
      w3.start();
      // the 3 workers are running
      // all sending messages to the same object

      // we block until the workers complete
      try {
         w1.join();
         w2.join();
         w3.join();
      }
      catch (InterruptedException ignored) {}

      System.out.println("Final sum:" + pair.sum());  // should be 6000
      /*
       If sum()/inc() were not synchronized, the result would
       be 6000 in some cases, and other times random values
       like 5979 due to the writer/writer conflicts of multiple
       threads trying to execute inc() on an object at the same time.
      */
   }
```

```
}
```

Multiple acquisition -- ok

A thread can acquire the same lock multiple times -- that works fine.
Put another way: a thread does not block waiting for itself. If a thread holds a lock, it can acquire that lock again.
e.g. inc() could call sum(), and it will work out right -- the lock will be released only when the thread's lock count goes to 0.
This is sometimes known as "recursive locks"

Exceptions -- lock release ok

A thread releases its locks as it returns from methods, no matter how. In particular, an exception terminating the method will release the locks correctly.
It's nice to have support for this sort of detail built in to the system at a low level.

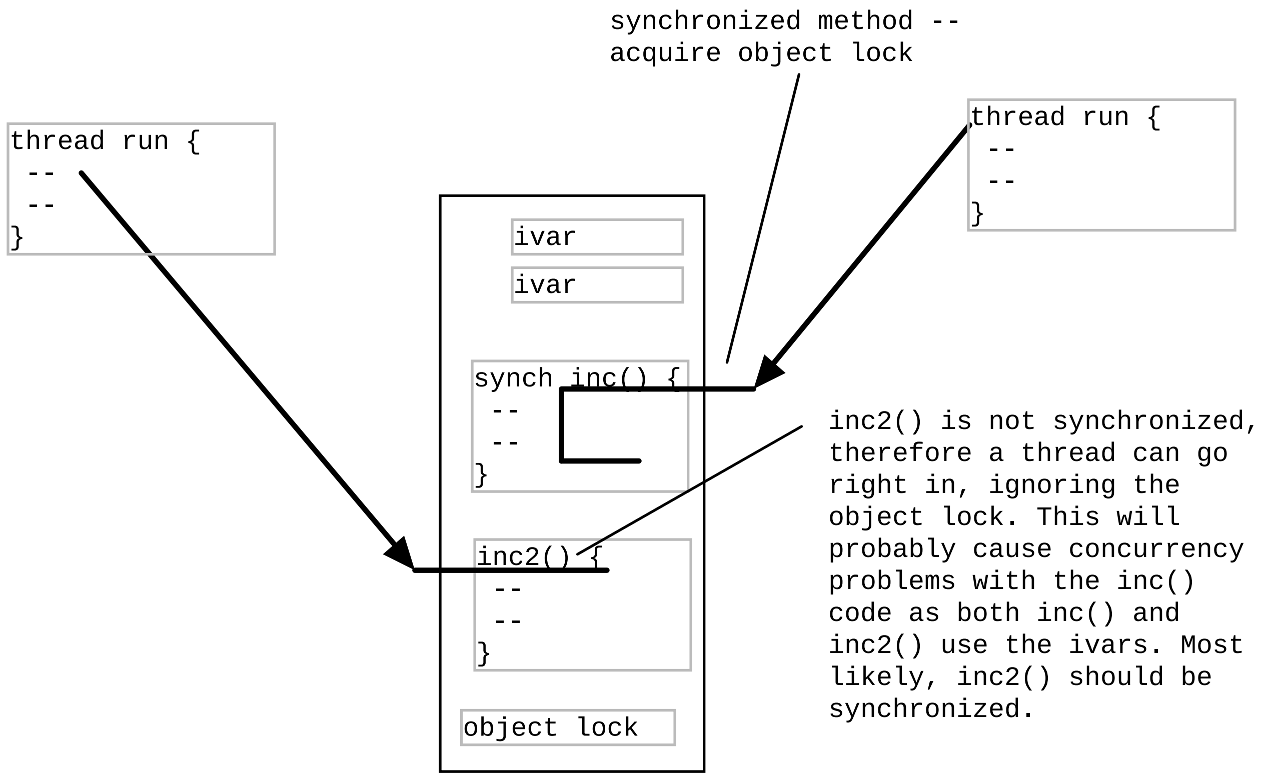
Synchronization Problems

Danger -- Unsynchronized method

```
public void inc2() { // note: not synchronized
   a++;
}
```

inc2() is not declared synchronized, so it does not obey the lock. inc2() could execute at the same time as inc(), causing a writer/writer conflict.

A method must volunteer to obey the lock with the synchronized keyword. If it makes sense for one method to be synchronized, probably they all should be.



inc2() is not synchronized, therefore a thread can go right in, ignoring the object lock. This will probably cause concurrency problems with the inc() code as both inc() and inc2() use the ivars. Most likely, inc2() should be synchronized.

Synchronize Transactions

Databases have an idea of a "transaction" -- a change that happens in full or is "rolled back" to not have happened at all.

Leave in good state

Think of your messages that way. A method gets the lock, makes all its changes (with sole possession of the lock), releases the lock leaving the object fully in the new state. Don't expose an object when it is half-updated. Use the lock to keep out other threads during the update.

Danger -- Split-Transaction

Suppose we have an Account object that responds to getBal() and setBal(), and these are synchronized.

```
class Account {
   int balance;

   public synchronized int getBal() { return(balance); }
   public synchronized void setBal(int val) {balance = val;}
}
```

Problem

Two threads could interleave their calls to get/set in a way to get the wrong answer.

```
Thread1: { int bal = a.getBal(); bal+=100; a.setBal(bal); }
Thread2: { int bal = a.getBal(); bal+=100; a.setBal(bal); }
```

The synch is at too fine grain -- the critical section is larger

This is tricky -- the programmer could think "I used synchronized everywhere" and think it's ok.

Solution

Move the synch out so it covers the whole data read-write cycle

```
public synchronized changeBal(int delta) {
   balance += delta;
}
```

Split-Transaction Vector

The old Vector class was like ArrayList, but get(), set(), and size() were synchronized. This was basically a bad idea, and that's why ArrayList is not synchronized.

Problem: Vector gave programmers the illusion that their client code was thread safe, but it could still suffer from split-transaction.

Problem: Vector made all code pay the lock/unlock cost, even if the client was single threaded.

Example: some code like the following was in the Vector class at one time. Both elementAt() and size() were synchronized, but lastElement() itself was not. What's the problem...

```
public Object lastElement() {
  return(elementAt(size()-1);
}
```

Get In and Get Out

For performance, it's best to hold the lock as little as possible

1. Do setup that does not require the lock
2. Acquire the lock
3. Do the critical operation
4. Release the lock
5. Do cleanup that does not require the lock

Get In and Get Out Example

Suppose our object has some data structure, and for the foo() operation want to add a String[] array to it.
The add() operation itself is a critical section, but setting up the String[] array is not..
add() is synchronized, but the pre-add code is not

```
public void foo() {  // not synchronized
   // note: multiple threads can run these setup steps
   // concurrently -- all stack vars
   String[] a = new String[2];
   a[0] = "hello";
   a[1] = "there";
   add(a);  // synchronized step
}


public synchronized add(String[] array) {
   // some critical section
}
```

Synchronized(obj) {...} Block

A variant of the synchronized method.
Acquire/Release lock for a specific object. Code looks like...

```
void someOperation(Foo foo) {
   int sum = 0;
   synchronized(foo) {  // acquire foo lock
      sum += foo.value;
   }  // release foo lock
   ...
```

Similar to synchronized method
   Uses the same lock as synchronized methods -- the lock in each object.
A little slower
A little less readable
Conclusion: synchronized methods are slightly preferable, but synchronized(obj) gives you flexibility -- you can talk about the lock of an object other than the receiver and in places other than the start/end of a method.

Synchronized(obj) {...} Block example

```
/*
 Demonstrates using individual lock objects with the
 synchronized(lock) {...} form instead of synchronizing methods --
 allows finer grain in the locking.
*/
class MultiSynch {
   // one lock for the fruits
   private int apple, bannana;
   private Object fruitLock;

   // one lock for the nums
   private int[] nums;
   private int numLen;
   private Object numLock;

   public MultiSynch() {
      apple = 0;
      bannana = 0;
      // allocate an object just to use it as a lock
      // (could use a string or some other object just as well)
      fruitLock = new Object();

      nums = new int[100];
      numLen = 0;
      numLock = new Object();
   }

   public void addFruit() {
      synchronized(fruitLock) {
         apple++;
         bannana++;
      }
   }

   public int getFruit() {
      synchronized(fruitLock) {
         return(apple+bannana);
      }
   }

   public void pushNum(int num) {
      synchronized(numLock) {
         nums[numLen] = num;
         numLen++;
      }
   }

   // Suppose we pop and return num, but if the num is negative return
   // its absolute value -- demonstrates holding the lock for the minimum time.
   public int popNum() {
      int result;
      synchronized(numLock) {
         result = nums[numLen-1];
         numLen--;
      }
      // do computation not holding the lock if possible
      if (result<0) result = -1 * result;
      return(result);
   }

   public void both() {
      synchronized(fruitLock) {
         synchronized(numLock) {
            // some scary operation that uses both fruit and nums
            // note: acquire locks in the same order everwhere to avoid
            // deadlock.
         }
      }
   }
}
```

Misc Thread Methods

Thread.currentThread()

The static method Thread.currentThread() returns a pointer to the Thread object for the current running thread.

The static methods below also work on the current running thread -- the thread which calls the method.

If the receiver object is a Thread subclass, it can give the false impression that the method works on the receiver. However, the static methods have no relationship with the receiver. They always affect the current running thread.

Thread.sleep(), Thread.yield()

Thread.sleep(milliseconds) blocks the current thread for approximately the given number of milliseconds. May throw an InterruptedException if the sleeping thread is interrupted.

Thread.yield() -- voluntarily give up the CPU, so that another thread may run. Just a hint to the VM that perhaps now would be a good time to run a different thread. Old VMs did not switch threads pre-emptively, so yield() made a real difference. However, it is probably not necessary with modern, pre-emptive VMs. However, yield() can be used to force more thread switching in an attempt to check for race condition problems. Also, simple VMs (such as on a Palm Pilot), may use a simple, non pre-emptive thread system, in which yield() would have some effect.

The preferred syntax to call these is Thread.sleep() or Thread.yield(), to emphasize that they are static.

Priorities

getPriority()/setPriority() on Thread objects

Threads have priorities, that the scheduler uses to give more time to some threads and less time to others.

Use priorities to optimize behavior, but not to safeguard critical sections -- priorities are not precise in that way. Use synchronization to protect critical sections no matter what the priorities are.

There is a school of thought that priorities introduce more complexity than they are worth in a concurrent program, and should never be used. Some VMs ignore priorities anyway.

getName()

getName() on a Thread object

By default, returns something like Thread-1, Thread-2, ... for each thread

Handy for debugging.

Alternately, the Thread ctor takes a String which it uses for the name.

Interruption

worker.interrupt()

interrupt() on a Thread object

- Send to a thread object to signal that it should stop
- Does not stop the thread right away -- the notification is "asynchronous"
- Essentiallty, this sets an "interrupted" boolean in the thread to true. (or it gets an InterruptedException which we'll deal with later).
- The thread should notice, eventually, that it has been interrupted and exit its run loop cleanly

isInterrupted()

- Send to a thread to see if (boolean) it has been interrupted.
- Typically, a worker thread object sends this message to itself in its run loop periodically to see if it has been interrupted.
- When interrupted, the worker should exit its run, leaving data structures in a clean state.
- boolean interrupted() -- similar to isInterrupted(), but clears the flag -- do not use.

Old stop() style

- Java used to feature synchronous thread methods such as stop(), but these have been deprecated, because it is practically impossible to get the "exits leaving the data structures in clean state" condition correct with them. The thread could get stoped partway through a statement, and so leave a data structure in a half-updated state in a way which makes it impossible for the program to continue reliably. For this reason, stop(), is being phased out.

isInterrupted() vs. InterruptedException

When a thread is interrupted, it will be informed either by having its isInterrupted boolean set, or if the thread is blocked, it will receive an InterruptedException. We'll look at the InterruptedException case later on.

interruption() example

```
/*
 Demonstrates creating a couple worker threads, running them,
 interrupting them, and waiting for them to finish.
*/
class StopWorker extends Thread {
    public void run() {
      long sum = 0;
      for (int i=0; i<5000000; i++) {
         sum = sum + i; // do some work

         // every n iterators... check isInterrupted()
         if (i%100000 == 0) {
            if (isInterrupted()) {
               // clean up, exit when interrupted
               // (getName() returns a default name for each thread)
               System.out.println(getName() + " interrupted");
               return;
            }
            System.out.println(getName() + " " + i);

            Thread.yield();
         }
      }
   }

   public static void main(String[] args) {
      StopWorker a = new StopWorker();
      StopWorker b = new StopWorker();

      System.out.println("Starting...");
      a.start();
      b.start();

      try {
         Thread.sleep(100);   // sleep a little, so they make some progress
      }
      catch (InterruptedException ignored) {}

      a.interrupt();
      b.interrupt();
      System.out.println("Interruption sent");

      try {
         a.join();
         b.join();
      }
      catch (Exception ignored) {}

      System.out.println("All done");
   }
/*
    Starting...
    Thread-0 0
    Thread-1 0
    Thread-1 100000
    Thread-0 100000
    Thread-1 200000
...
    Thread-0 900000
    Interruption sent
    Thread-0 interrupted
    Thread-1 interrupted
    All done
*/
}
```