Threading 2

From Previous Handout

Threads vs. Processes
3 reasons to use threads
Thread class
Using: new, start(), run(), join()
Race condition / critical section
Reader / writer writer / writer conflicts
Synchronized keyword -- object lock

Current Thread

When have a sequence of statements

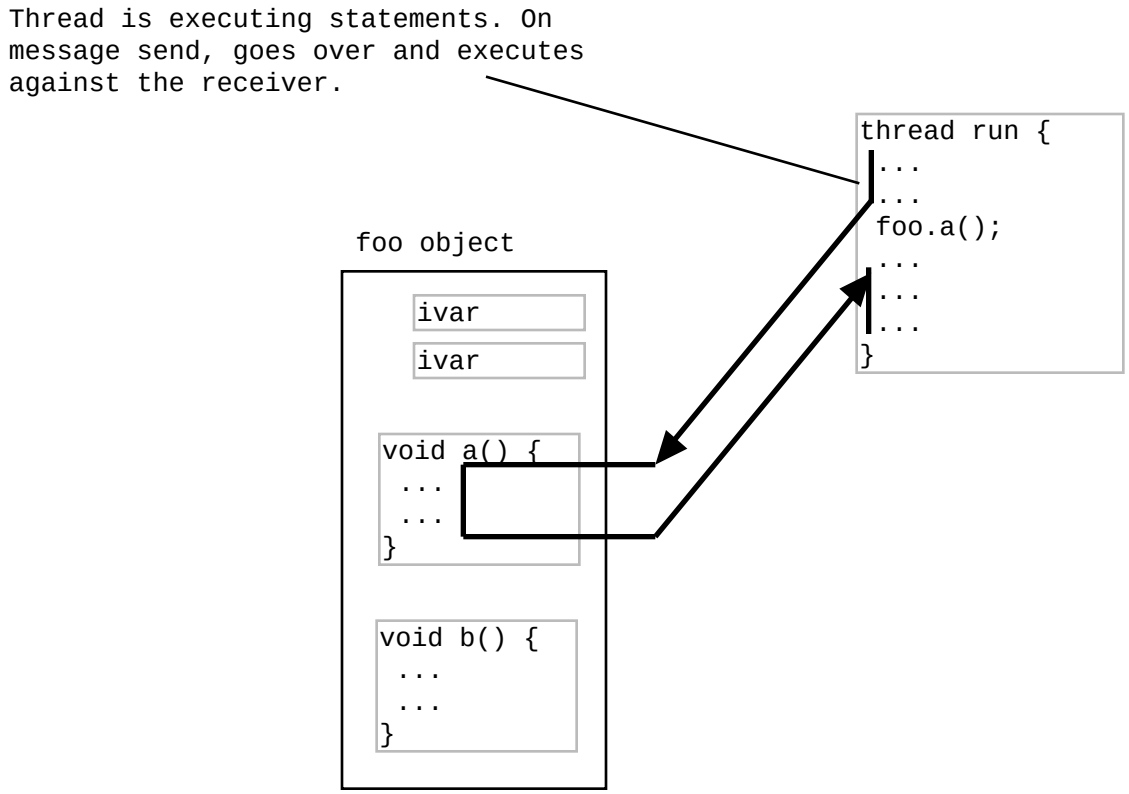
```
int i =7;
while (i<10) {
  foo.a();
  ...
}
```

For a sequence of statements to execute, there must be a thread that has been called to execute them -- the "current running thread".
A message send, in essence, transfers the current running thread temporarily to go execute a method against the receiver.

static void main(String[] args)

A Java program begins with a thread executing main(), and that one thread executes the whole program.
We will see how to create and run other threads which will run concurrently.

Current Thread Picture



Basic Synchronization

(See previous handout)

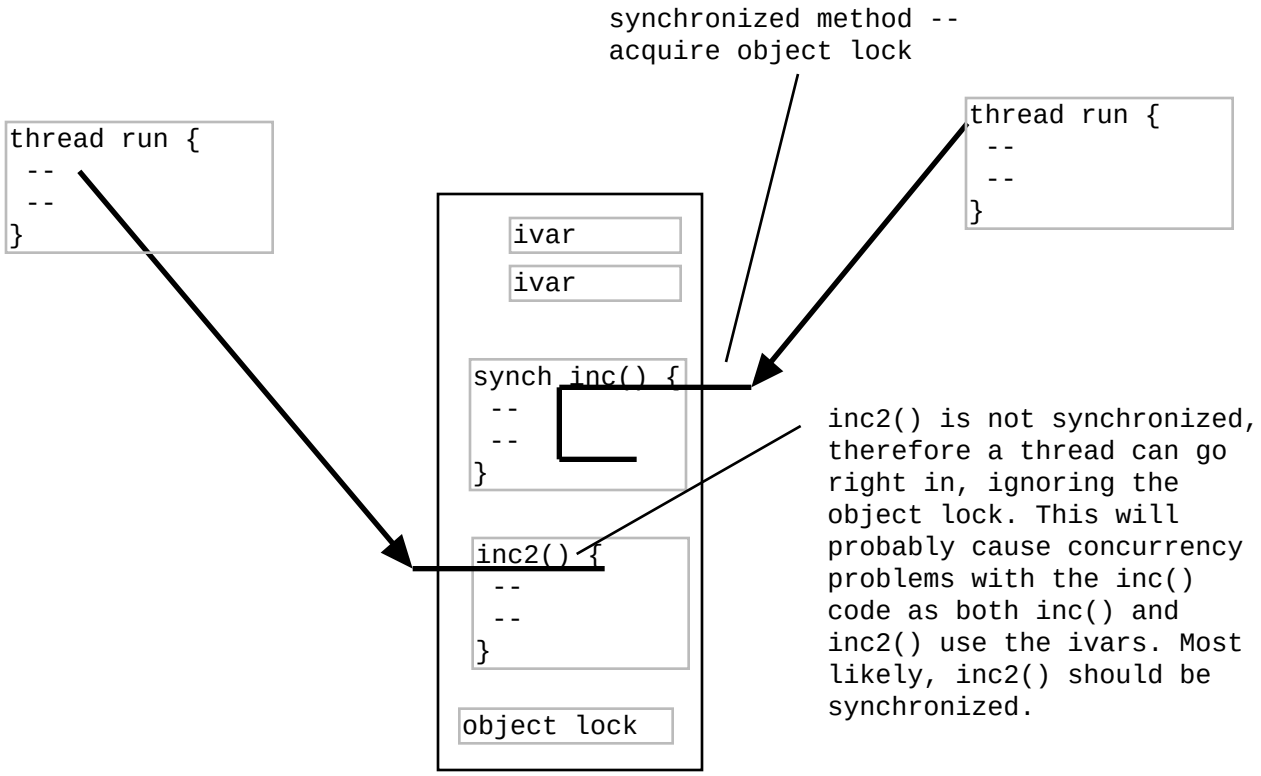
Synchronization Code Issues

1. Unsynchronized methods -- danger

```
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.



2. Multiple acquisition -- ok

A thread can acquire the same lock multiple times -- works fine.

Put another way: a thread does not block waiting for itself.

e.g. inc() could call sum(), and it will work out right -- the lock will be released only when inc() finally exits.

3. 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.

Concurrent Style

Deliberate Design

By default, ignore concurrency issues.

Like subclassing, support for concurrency should be deliberately added to a class if it is needed. Adding support for concurrency is not a trivial little operation.

Design for Concurrency

Does the pattern of locks allow multiple threads to get useful work done at the same time?

There will be some moments when threads must wait for each other, but ideally, much of the time they can each proceed independently.

If the threads must "take turns" -- only one getting work done at a time -- that's a bad sign

Typical good design

> Each thread checks a Foo object out from the foo storage (locked, one at a time). However, once the thread has the foo object,, it can operate on it independently. Another lock is required to for the (fast) check-in operation to merge the results back in to the shared storage.

Typical bad design

> Each thread checks a foo object out from the foo storage, and continues to hold the foo storage lock (or some other globally needed lock) while operating on the foo object. Essentially, only one thread can get work done at a time.

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 release the lock when the object is partially in the new state.

Don't worry about order

> If the above is true, you don't have to worry about the order the methods happened to acquire the lock.

Split-Transaction Problems

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.
- The synch is at too fine grain -- the critical section is larger
- This is tricky -- the programmer could think "I used synchronized everywhere" so they think it's ok.

Solution
- Move the synch out so it covers the whole transaction

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

Note: For HW4a, there is a sort of split transaction in the time between the call to withdraw() and deposit(), but I'm allowing it for that case.

Get In and Get Out

It's generally regarded as good style 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

```
public synchronized add(String[] a) { ...}

public void foo() {
   // 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
}
```

Synchronization cost

Acquiring and releasing a lock each have a moderate runtime cost
Therefore, when running on a machine with only one processor, the concurrent code may be a little slower than the non-concurrent code, since the concurrent code has extra acquire/release costs

Coarse vs. Fine locks

Given that synchronization itself has a cost, a design can be tuned to have just a few broad locks (coarse grain) or many small locks (fine grain)
Fine grain allows more concurrency, but spends more time acquiring and releasing locks
It's a tradeoff that depends on the particular problem

Immutable objects

An object that does not change after construction, such as String or Piece, avoids synchronization problems for clients and implementors -- this makes the immutable design a little more attractive in a threaded environment. It's less work, and it is immune from various scary concurrency bugs.

Other Thread Methods

sleep() / yield()

These are static methods in the Thread class. They do not operate on the receiver -- they operate on the current running thread.

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

sleep(milliseconds) blocks the current thread for the given number of milliseconds. May throw an InterruptedException

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.

Interruption

interrupt()

- Send to a thread object to signal that it should stop
- Does not stop the thread right away -- the notification is "asynchronous"
- 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.

interruption() example

```
/*
 Demonstrates creating a couple worker threads, running them,
 interrupting them, and waiting for them to finish.
*/
class Thread3 {
   // Subclass off Thread and override run()
   static class Worker 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) {
               System.out.println("Working:" + i);

               if (isInterrupted()) {
                  // clean up, exit when interrupted
                  System.out.println("Interrupted");
                  return;
               }

               Thread.yield();
            }
         }
      }
   }

   public static void demo() {
      Worker a = new Worker();
      Worker b = new Worker();

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

      try {
         Thread.sleep(200);   // sleep a little, so they make some progress
      }
      catch (InterruptedException ignored) {}

      System.out.println("Interrupting...");
      a.interrupt();
      b.interrupt();

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

      System.out.println("All done");
   }
```

```
   /*
      Starting...
      Working:0
      Working:100000
      Working:0
      Working:200000
      Working:300000
      Working:100000
      Working:200000
      Working:400000
      Interrupting...
      Working:500000
      Interrupted
      Working:300000
      Interrupted
      All done
   */

}
```

Priorities

getPriority() / setPriority()

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.

synchronized(obj) { ...}

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) {...} 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 the lock inside it
      // (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 it is negative we make
   // it positive -- 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 = Math.abs(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.
         }
      }
   }
}
```