CS193J: Programming in Java
Summer Quarter 2003

Lecture 9
Threading, Synchronization, Interruption

Manu Kumar

sneaker@stanford.edu

Handouts

- 1 Handout for today!
  - #21: Threading 3

Roadmap

- We are half way through this course!
  - We have covered
    - Course Overview / Introduction to OOP/Java
    - OOP / Java
    - Collections and more OOP
    - OOP Inheritance, Abstract Classes and Interfaces
    - Java Swing and LayoutManagers
    - Inner Classes and Listeners
    - Repaint, Mouse Tracking and Advanced Drawing
    - Object Serialization and Introduction to Threading

Coming up…

- Threading – synchronization, wait/notify, swing thread
- MVC / Tables
- Exceptions / Files and Streams
- XML
- SAX XML Parsing
- Advanced Java
- Guest Speaker

Recap

- Last Time
  - Object Serialization
    - Cloning
      - Not Dolly, but Java Objects ☺
    - Serializing
  - Introduction to Threading
    - Motivation
    - Java threads
      - Simple Thread Example
- Assigned Work Reminder
  - HW 2: Java Draw
    - Due before midnight on Wednesday, July 23rd, 2003

Today

- Review Introduction to Threading
  - – Java threads
    - Simple Thread Example
- Threading 2
  - – Race Conditions
  - – Locking
  - – Synchronized Method
  - – Thread Interruption
- We’ll try to end a little early to let you get back to Homework #2!
  - – Due tomorrow

Threads

- The ability to do multiple things at once within the same application
  - – Finer granularity of concurrency
- Lightweight
  - – Easy to create and destroy
- Shared address space
  - – Can share memory variables directly
  - – May require more complex synchronization logic because of shared address space

Advantages of threads…

- Use multiple processors
  - Code is partitioned in order to be able to use n processors at once
    - This is not easy to do! But Moore’s Law may force us in this direction
- Hide network/disk latency
  - While one thread is waiting for something, run the others
  - Dramatic improvements even with a single CPU
    - Need to efficiently block the connections that are waiting, while doing useful work with the data that has arrived
  - Writing good network codes relies on concurrency!
    - Homework #3b will be a good example of this
- Keeping the GUI responsive
  - Separate worker threads from GUI thread

Java Threads

- Java includes built-in support for threading!
  - Other languages have threads bolted-on to an existing structure
- VM transparently maps threads in Java to OS threads
  - Allows threads in Java to take advantage of hardware and operating system level advancements
  - Keeps track of threads and schedules them to get CPU time
  - Scheduling may be pre-emptive or cooperative

Current Running Thread

- “Thread of control” or “Running thread”
  - The thread which is currently executing some statements
- A thread of execution
  - Executing statements, sending messages
  - Has its own stack, separate from other threads
- A message send sends the current running thread over to execute the code in the receiver

Java Thread class

- A Thread is just another object in Java
  - It has an address, responds to messages etc.
  - Class Thread
    - in the default java.lang package
- A Thread object in Java is a token which represents a thread of control in the VM
  - We send messages to the Thread object; the VM interprets these messages and does the appropriate operations on the underlying threads in the OS

Creating Threads in Java

- Two approaches
  - – Subclassing Thread
    - Subclass java.lang.Thread
    - Override the run() method
  - – Implementing Runnable
    - Implement the runnable interface
    - Provide an implementation for the run() method
    - Pass the runnable object into the constructor of a newThread Object

Why two approaches?

- Remember: Java supports only single-inheritance
  - – If you need to extend another class, then cannot extend thread at the same time
    - Must use the Runnable pattern
- Two are equivalent
  - – Whether you subclass Thread or implement Runnable, the resulting thread is the same
  - – Runnable pattern just gives more flexibility

Simple Thread Example

```
/*
 Demonstrates creating a couple worker threads, running them,
 and waiting for them to finish.

 Threads respond to a getName() method, which returns a string
 like "Thread-1" which is handy for debugging.
*/
public class Worker1 extends Thread {
    public void run() {
            long sum = 0;
            for (int i=0; i<100000; i++) {
                    sum = sum + i;      // do some work

                    // every n iterations, print an update
                    // (a bitwise & would be faster -- mod is slow)
                    if (i%10000 == 0) {
                            System.out.println(getName() + " " + i);
                    }
            }
    }
```

Simple Thread Example

```
    public static void main(String[] args) {
            Worker1 a = new Worker1();
            Worker1 b = new Worker1();

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

            // The current running thread (executing main()) blocks
            // until both workers have finished
            try {
                    a.join();
                    b.join();
            }
            catch (Exception ignored) {}

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

Simple Thread Example Output

```
Starting...
Thread-0 0
Thread-1 0
Thread-0 10000
Thread-0 20000
Thread-1 10000
Thread-0 30000
Thread-1 20000
Thread-0 40000
Thread-1 30000
Thread-0 50000
Thread-1 40000
Thread-0 60000
Thread-1 50000
Thread-0 70000
Thread-1 60000
Thread-0 80000
Thread-0 90000
Thread-1 70000
Thread-1 80000
Thread-1 90000
All done
```

Threading 2 (Handout #20)

- Two Threading Challenges
  - Mutual Exclusion
    - Keeping the threads from interfering with each other
    - Worry about memory shared by multiple threads
  - Cooperation
    - Get threads to cooperate
      - Typically centers on handing information from one thread to the other, or signaling one thread that the other thread has finished doing something
    - Done using join/wait/notify

Critical Section

- A section of code that causes problems if two or more threads are executing it at the same time
  - Typically as a result of shared memory that both thread may be reading or writing
- Race Condition
  - When two or more threads enter a critical section, they are supposed to be in a race condition
    - Both threads want to execute the code at the same time, but if they do then bad things will happen

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

- Case
  - thread1 runs inc(), while thread2 runs sum()
    - thread2 could get an incorrect value if inc() is half way done
    - This happens because the lines of sum() and inc() interleave
- Note
  - Even a++ and b++ are *not* atomic statements
    - Therefore, interleaving can happen at a scale finer than a single statement!
    - a++ is really three steps: read a, increment a, write a
  - Java guarantees 4-byte reads and writes will be atomic
  - This is only a problem if the two threads are touching the same object and therefore the same piece of memory!

Writer/Writer Conflict

- Case
  - – thread1 runs inc() while thread2 runs inc() on the same object
    - The two inc()'s can interleave in order to leave the object in an inconsistent state
- Again
  - – a++ is not atomic and can interleave with another a++ to produce the wrong result
  - – This is true in most languages

Heisenbugs

- Random Interleave – hard to observe
  - Race conditions depend on having two or more threads "interleaving" their execution in just the right way to exhibit the bug
    - Happens rarely and randomly, but it happens
  - Interleaves are random
    - Depending on system load and number of processors
    - More likely to observe issue on multi-processor systems
- Tracking down concurrency bugs can be hard
  - Reproducing a concurrency bug reliable is itself often hard
  - Need to study the patterns and use theory in order to pre-emptively address the issue

Java Locks

- Java includes built-in support for dealing with concurrency issues
  - Includes keywords in order to mark critical sections
  - Includes object locks in order to limit access to a single thread when necessary
- Java designed to encourage use of threading and concurrency
  - Provides the tools needed in order to minimize concurrency pitfalls

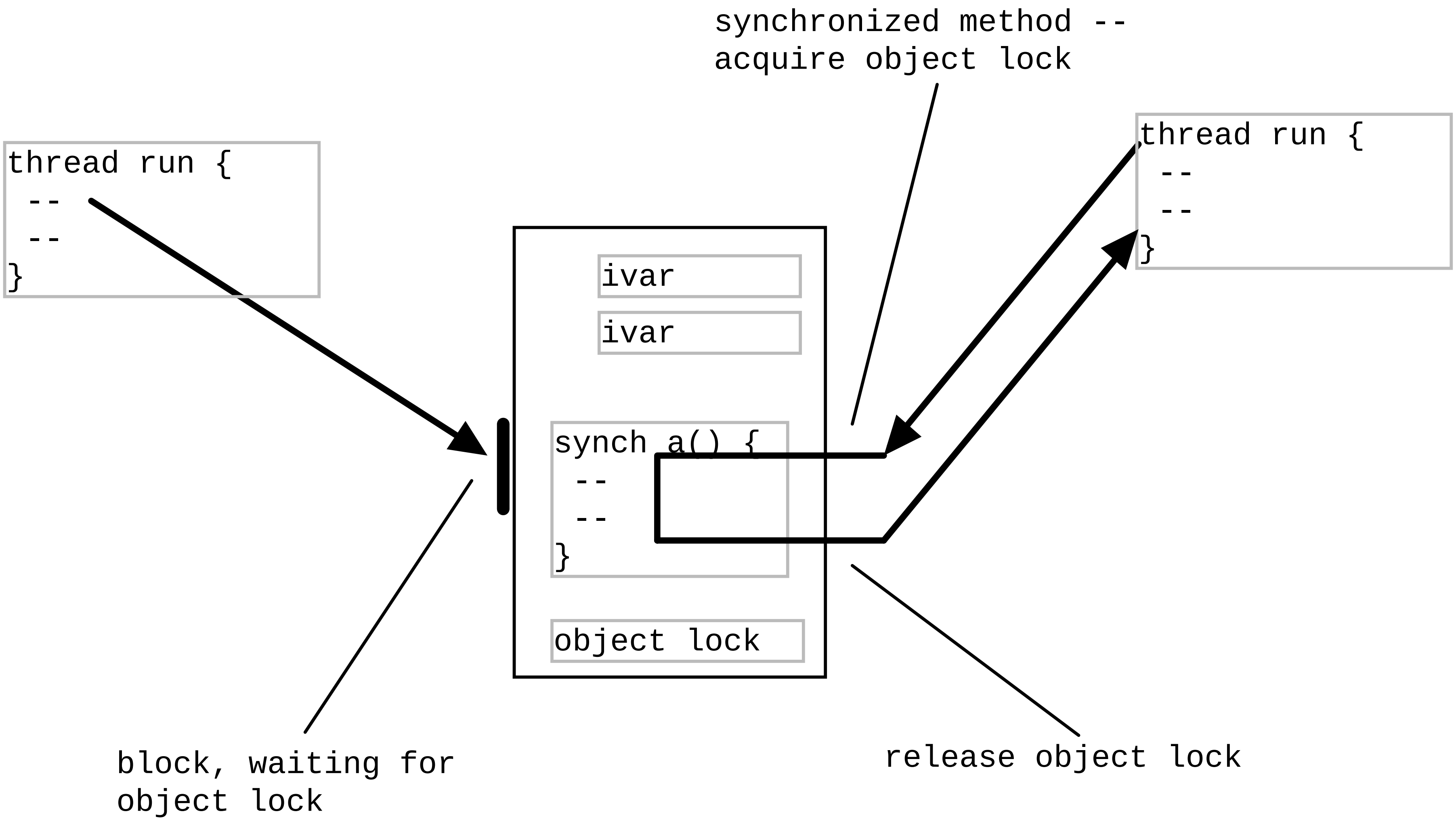
Object Lock and Synchronized keyword

- Every Java Object has as lock associated with it
- A “synchronized” keyword respects the lock of the receiver object
  - For a thread to execute a synchronized method against a receiver, it must first obtain the lock of the receiver
  - The lock is released when the method exits
  - If the lock is held by another thread, the calling thread blocks (efficiently) till the other thread exits and the lock is available
  - Multiple threads therefore take turns on who can execute against the receiver

Receiver Lock

- The lock is in the receiver object
  - Provides mutual exclusion mechanism for multiple threads sending messages to **that object**
  - Other objects have their own lock
- If a method is not sychronized
  - The thread will not acquire the lock before executing the method

Sychronized Method Picture



synchronized method --
acquire object lock

```
thread run {
 --
 --
}
```

```
ivar
```

```
ivar
```

```
synch a() {
 --
 --
}
```

```
object lock
```

```
thread run {
 --
 --
}
```

block, waiting for
object lock

release object lock

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;
      }
```

Synchronized Method Example

```
    // 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++;
    }
}
```

Synchronized Method Example

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

Synchronized Method Example

```
/*
 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
```

Synchronized Method Example

```
        // 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.
        */
    }
}
```

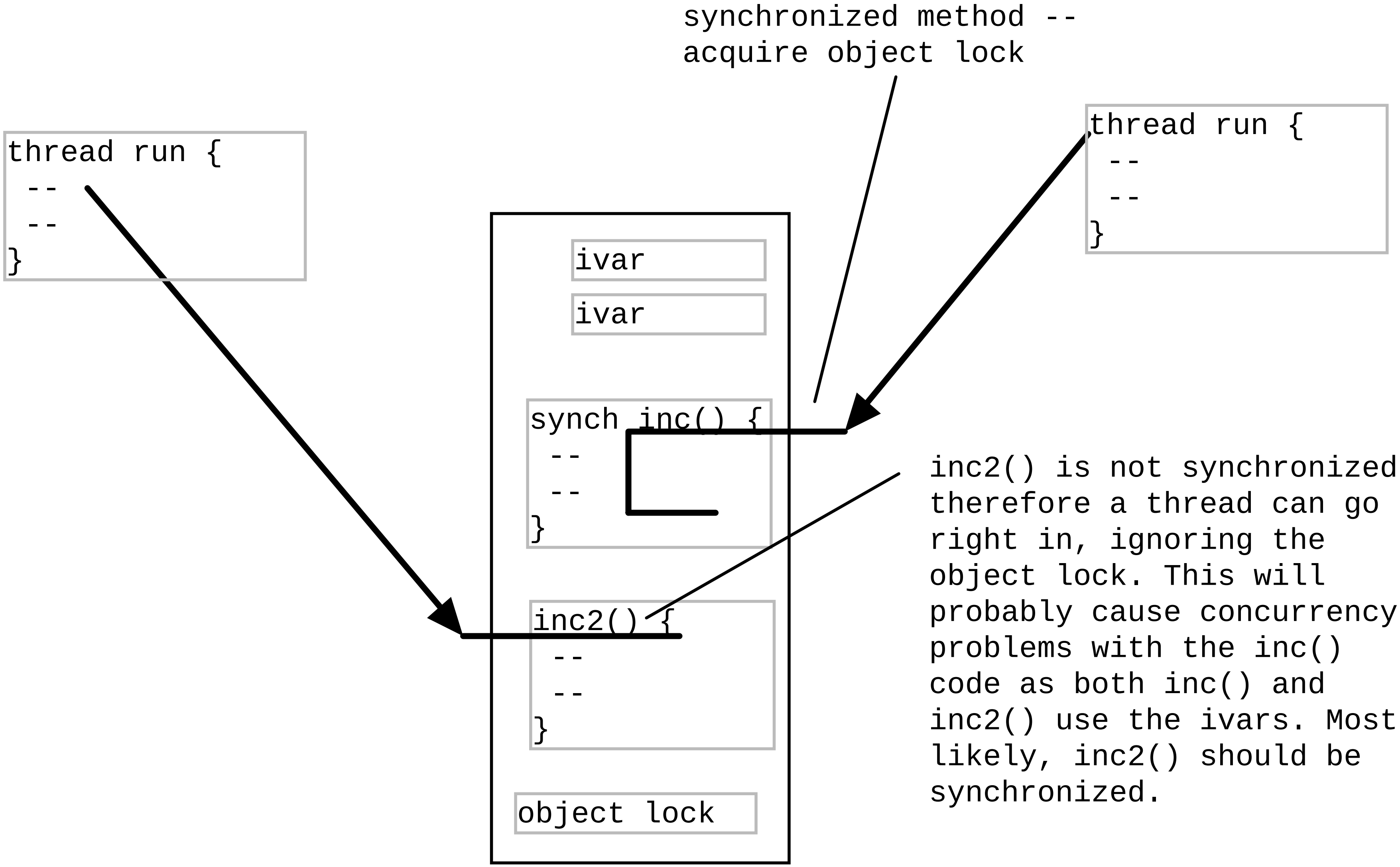
Locks in Java

- Multiple acquisition of locks
  - A thread can acquire the same lock multiple times
    - A thread does not block waiting for itself, if it holds a lock and it can acquire the lock again
  - Example
    - inc() could call sum()
      - The thread can acquire the lock again and will only be released when the lock count goes to zero
    - Sometimes called ‘recursive locks’
- Exceptions release
  - A thread releases the locks regardless of how it exits the method
    - Graceful and ungraceful termination (exceptions) both release locks!
    - This is critical to prevent deadlocks

Synchronization Problems

- Unsynchronized method warning/danger
  - All methods that touch shared state must be synchronized
    - Otherwise a thread could get in to a unsynchronized method without checking the lock
  - A method must volunteer to obey the lock with the synchronized keyword
    - If it makes sense for one method to by synchronized, probably others should be too!

Unsynchronized Method Example



synchronized method --
acquire object lock

```
thread run {
  --
  --
}
```

```
thread run {
  --
  --
}
```

```
ivar
ivar

synch inc() {
  --
  --
}

inc2() {
  --
  --
}

object lock
```

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.

Transaction Semantics

- Similar concept as in Databases
  - Transaction is a change that happens in full or is “rolled back” to have not happened at all
- Leave your objects in a consistent state
  - A method gets the lock
  - Makes changes to the object state (while holding the lock)
  - Releases the lock leaving the object fully in the new state
  - Object is not exposed when half updated
    - The lock is used to keep other threads out during the update

Split Transaction

```
class Account {
        int balance;

        public synchronized int getBal() {
                return(balance);
        }

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

Split Transaction Problem

- Two thread could interleave in a way to give erroneous results

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

- Problem
  - Synchronization is too fine grained
    - Critical section is larger
  - Tricky
    - Programmer may think he/she has synchronized, but not adequately

Split Transaction

- Solution
  - – Move the synchronization to cover entire critical section

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

Split-Transaction Vector

- Vector similar to ArrayList
  - get(), set() and size() were synchronized
- Problem
  - Gave programmers the illusion that their client code was thread-safe
    - Still suffered from split-transaction errors
  - Overhead for locking and unlocking even with single-threaded code
  - The entire critical section was not covered
    - Example

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

Get In and Get Out

- For performance, better 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

- Setting up the array is done outside of critical section

```
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
  - Uses same lock as the synchronized method
    - The lock in the object
  - A little slower
  - A little less readable
- Synchronized methods are preferable
  - But synchronized(obj) {…} gives maximum flexibility
  - Can use the lock of an object other than the receiver
  - Can minimize size of the critical section

Synchronized(obj) {…} syntax

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

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;
```

Synchronized(obj) {…} Block Example

```
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++;
        }
}
```

Synchronized(obj) {…} Block Example

```
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);
}
```

Synchronized(obj) {…} Block Example

```
  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()
  - Static method
  - Returns a pointer to the Thread object for the current running thread
- Warning!
  - If the receiver is a Thread subclass, it can give the false impression that the above (and following methods) work on the receiver!
  - Static methods have not relationship with the receiver
  - They always affect the running thread

Thread.sleep(), Thread.yield()

- Thread.sleep(milliseconds)
  - Blocks the current thread for approximately the given number of milliseconds
    - May thrown an InterruptedException if the sleeping thread is interrupted
- Thread.yield()
  - Voluntarily give up the CPU so that another thread may run
    - A hint to the VM, not guaranteed
    - Not as useful on the pre-emptive multi-tasking OS
      - Useful for things like Palm or phone
- Preferred syntax is Thread.sleep() or Thread.yield() to emphasize static nature

Thread Priorities

- getPriority() and setPriority() on Thread objects
  - Used to optimize behavior
    - Not to safeguard critical sections
  - Some VMs ignore priorities
    - Improvements in hardware and OS may sometimes do a better job of scheduling threads than the programmer!

getName()

- Returns the String name of the Thread
  - Useful when debugging and printing out the name of the thread
  - Thread-1, Thread-2 etc.
- Thread class constructor takes a string argument which sets the name of the thread!

Thread Interruption

- interrupt()
  - – Signal a thread object that it should stop running
  - – Asynchronous notification
    - Does not stop the thread right away
    - Sets an “interrupted” boolean to true
  - – Thread must check and do appropriate thing
- isInterrupted()
  - – Checks to see if a interrupt has been requested
  - – Idiom – check isInterrupted in a loop
    - When interrupted, should exit leaving object in a clean state

Stop() -- deprecated

- stop()
  - Performs a synchronous stop of the thread
  - Usually impossible to ensure that the object is left in a consistent state when using stop
  - Deprecated in favor or using interrupt() and doing a graceful exit

Interruption() example

```
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();
                        }
            }
    }
```

Interruption() example

```
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");
}
```

Interruption() example output

```
/*
    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
*/
```

Summary

- Today
  - Review Introduction to Threading
    - Java threads
      - Simple Thread Example
  - Threading 2
    - Race Conditions
    - Locking
    - Synchronized Methods
    - Thread Interruption
- Assigned Work Reminder
  - HW 2: Java Draw
    - Due before midnight on Wednesday, July 23rd, 2003