

---

**COMP 322: Principles of Parallel Programming**  
**Lecture 12: Java Concurrency (Chapter 6)**

**Fall 2009**

**Vivek Sarkar**  
**Department of Computer Science**  
**Rice University**  
**vsarkar@rice.edu**



# Summary of Previous Lecture

---

- POSIX Threads as an illustration of parallel programming issues:
  - Fairness
  - Serializability
  - Deadlock
  - Safety
  - Liveness
  - Bounded buffer example
- Many of these issues arise in other parallel programming languages, but constructs in higher level languages such as HJ can help (async, finish, isolated, phasers)

# Acknowledgments for Today's Lecture

---

- Course text: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>
- "Introduction to Concurrent Programming in Java", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
- "Java Concurrency Utilities in Practice", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Tim Peierls, Brian Goetz
- Additional references:
  - Java Concurrency in Practice, by Brian Goetz et al, Addison-Wesley (JCIp)
  - Concurrent Programming in Java, by Doug Lea, Addison-Wesley (CPJ)

# Java Threading Model

---

- The Java virtual machine (JVM)
  - Creates the initial Java thread which executes the main method of the class passed to the JVM
  - Most JVM's use POSIX threads to implement Java threads
  - Creates internal JVM helper threads
    - Garbage collection, finalization, signal dispatching ...
- The code executed by the 'main' thread can create other threads
  - Either explicitly; or
  - Implicitly via libraries:
    - AWT/Swing, Applets
    - Servlets, web services
    - RMI
    - image loading
    - ...

# Java Threads

- Concurrency is introduced through objects of the class `Thread`
  - Provides a 'handle' to an underlying thread of control
- Always a 'current' thread running-static method `Thread.currentThread()`  
Returns `Thread` object associated with currently executing thread
- This program uses two threads: the main thread and a `HelloThread`
  - Each prints a greeting - the order of which is nondeterministic
  - During `Thread.start()` both threads are eligible for scheduling  
Scheduler gets to decide which runs when

```
public static void main(String[] args) {  
    class HelloThread extends Thread {  
        public void run() {  
            System.out.println("Hello from thread "  
                + Thread.currentThread().getName());  
        }  
        Thread t = new HelloThread();    // create HelloThread  
        t.start();                        // start HelloThread  
        System.out.println("Hello from main thread");  
    }  
}
```

# Thread Interaction

---

- `void start()`
  - Creates a new thread of control to execute the `run()` method of the `Thread` object
  - Can only be invoked once per `Thread` object
- `void join()`
  - Waits for a thread to terminate
  - `t1.join(); // blocks current thread until t1 terminates`
- `static void sleep(long ms) throws InterruptedException`
  - Blocks current thread for approximately at least the specified time
- `static void yield()`
  - Allows the scheduler to select another thread to run
  - Can be important to guarantee liveness (in the absence of fairness)

# Code Spec 6.19 Java's Thread class methods.

---

## The Java Thread Class

```
public synchronized void start()
```

- Starts this Thread and returns immediately after invoking the run() method.
- Throws `IllegalThreadStateException` if the thread was already started.

```
public void run()
```

- The body of this Thread, which is invoked after the thread is started.

```
public final synchronized void join(long millis)  
throws InterruptedException
```

- Waits for this Thread to die. A timeout in milliseconds can be specified, with a timeout of 0 milliseconds indicating that the thread will wait forever.

```
public static void yield()
```

- Causes the currently executing Thread object to yield the processor so that some other runnable Thread can be scheduled.

```
public final int getPriority()
```

- Returns the thread's priority.

```
public final void setPriority(int newPriority)
```

- Sets the thread's priority.

### Notes:

For a complete list of public methods for the Thread class, see `java.lang.Thread`.

# Objects and Locks

---

- Every Java object has an associated lock acquired via:
  - **synchronized** statements
    - `synchronized( foo ){`  
`// execute code while holding foo's lock`  
`}`
  - **synchronized** methods
    - `public synchronized void op1(){`  
`// execute op1 while holding 'this' lock`  
`}`
- Only one thread can hold a given lock at a time
  - If the lock is unavailable the thread is blocked
- Locking and unlocking are **automatic**
  - Can't forget to release a lock
  - Locks are released when a block goes out of scope
    - By normal means: end of block reached, `return`, `break`
    - When an exception is thrown and not caught
- `Thread.holdsLock(obj)` - return true if current thread holds lock of `obj`
  - Useful for assertions and debugging

# Locks are Reentrant

---

- Locks are **granted** on a **per-thread** basis
  - Called *reentrant* or *recursive* locks
  - Promotes object-oriented concurrent code
- A synchronized block means *execution of this code requires the current thread to hold this lock*
  - If it does – fine
  - If it doesn't – then acquire the lock

Lock is only released by the block that actually acquired it

- Reentrancy means that recursive methods, invocation of **super** methods, or local callbacks, don't deadlock

```
public class Widget {  
    public synchronized void doSomething() { ... }  
}  
  
public class LoggingWidget extends Widget {  
    public synchronized void doSomething() {  
        Logger.log(this + ": calling doSomething()");  
        super.doSomething(); // Doesn't deadlock!  
    }  
}
```

# Count 3s Problem

---

- Problem statement: count the number of 3's in a given int[] array
- Sequential code:  
`count = 0;`  
`for (int i=0; i<array.length; i++)`  
 `if (array[i]==3) count++`

## Figure 6.27 Count 3s solution in Java

---

```
1  import java.util.*;
2  import java.util.concurrent.*;
3
4  public class CountThrees implements Runnable
5  {
6      private static final int ARRAY_LENGTH=1000000;
7      private static final int MAX_THREADS=10;
8      private static final int MAX_RANGE=100;
9      private static final Random random=new Random();
10     private static int count=0;
11     private static Object lock=new Object();
12     private static int[] array;
13     private static Thread[] t;
14
15     public static void main(String[] args)
16     {
17         array=new int[ARRAY_LENGTH];
18
19         //initialize the elements in the array
20         for(int i=0; i<array.length; i++)
21         {
22             array[i]=random.nextInt(MAX_RANGE);
23         }
```

# Figure 6.27 Count 3s solution in Java. (contd.)

```
25 //create the threads
26 CountThrees[] counters=new CountThrees[MAX_THREADS];
27 int lengthPerThread=ARRAY_LENGTH/MAX_THREADS;
28
29 for(int i=0; i<counters.length; i++)
30 {
31     counters[i]=new CountThrees(i*lengthPerThread,
32                                 lengthPerThread);
33 }
34
35 //run the threads
36 for(int i=0; i<counters.length; i++)
37 {
38     t[i]=new Thread(counters[i]);
39     t[i].start();
40 }
41 for(int i=0; i<counters.length; i++)
42 {
43     try
44     {
45         t[i].join();
46     }
47     catch(InterruptedException e)
48     { /*do nothing*/ }
49 }
50
51 //print the number of threes
52 System.out.println("Number of threes: " + count);
53 }
54
55 private int startIndex;
56 private int elements;
57 private int myCount=0;
58
59 public CountThrees(int start, int elem)
60 {
61     startIndex=start;
62     elements=elem;
63 }
```

```
65 //Overload of run method in the Thread class
66 public void run()
67 {
68     //count the number of threes
69     for(int i=0; i<elements; i++)
70     {
71         if(array[startIndex+i]==3)
72         {
73             myCount++;
74         }
75     }
76     synchronized(lock)
77     {
78         count+=myCount;
79     }
80 }
81 }
```

# Count 3s solution in HJ

---

```
1. public class CountThrees {
2.     private static final int ARRAY_LENGTH=1000000;
3.     private static final int MAX_THREADS=10;
4.     private static final int MAX_RANGE=100;
5.     private static final Random random=new Random();
6.     private static int count=0;
7.     private static final int[] array = new int[ARRAY_LENGTH];
8.     public static void main(String[] args){
9.         for (int i=0; i<array.length; i++) array[i]=random.nextInt(MAX_RANGE);
10.        forall (point [i]:[0:MAX_THREADS-1]) {
11.            int myCount = 0;
12.            for (int j=0; j<lengthPerThread; j++)
13.                if (array[i*lengthPerThread+j]==3) myCount++;
14.            isolated count += myCount;
15.        } // forall
16.        System.out.println("Number of threes: " + count);
17.    } // main()
18.} // CountThrees
```

---

# Use of wait/notify with locks

---

- **Waiting for a condition to hold:**

```
synchronized (obj) { // obj protects the mutable state
    while (!condition) {
        try { obj.wait(); } // like pthread_cond_wait()
        catch (InterruptedException ex) { ... }
    }
    // make use of condition while obj still locked
}
```

- **Changing a condition:**

```
synchronized (obj) { // obj protects the mutable state
    condition = true;
    obj.notifyAll(); // like pthread_cond_broadcast()
    // or obj.notify() // like pthread_cond_signal()
}
```

- **Golden rule: Always** test a condition in a loop
  - Change of state may not be what you need
  - Condition may have changed again
    - No built-in protection from **'barging'**
  - Spurious wakeups are permitted - and can occur

# Waiting and Notification

---

- The `wait` methods will
  - Atomically release the lock and block the current thread
  - Reacquire lock before returning
- `notify()` means wake up *one* waiting thread
- `notifyAll()` means wake up *all* waiting threads

```
public class TrafficSignal {
    public enum Color { GREEN, YELLOW, RED };
    @GuardedBy("this") private Color color;
    public synchronized void setColor(Color color) {
        this.color = color;
        notifyAll();
    }
    public synchronized void awaitGreen() throws InterruptedException
    {
        while (color != Color.GREEN) wait();
    }
}
```

# Visibility

---

- Locking also has another consequence - *visibility*
  - Relates to “Java Memory Model” (to be discussed later)
- In the absence of synchronization, a thread reading a variable that has been modified by another thread can see a *stale value*
  - This is allowed to enable hardware caching and compiler optimization
  - In the absence of synchronization, different threads may see different values of the same variable

And this is perfectly OK!
- As long as you follow the rule for synchronization, you need never worry about this
- When two threads synchronize on the same lock, all variable values written by the releasing thread are visible to the next thread that acquires the lock
- The details are complicated, but the rule is simple
  - All threads that access a shared, mutable variable must synchronize using a common lock

# Java Memory Model

---

- **Programmers view:**
  - Everything happens in the order I indicate through the code statements that I write
- **Reality ( JVM/compiler & processor):**
  - Everything happens in whatever order the implementation chooses, so long as the program(mer) can't tell the difference
- **For single-threaded systems:**
  - Program order can't be distinguished from actual order
- **For multi-threaded systems:**
  - Without correct use of synchronization different threads can see different actions in memory
    - At different times
    - In different orders
- **The Memory Model defines the rules**

# The Java Memory Model (JMM)

---

- Conceptually simple:
  - Every time a variable is written, the value so written is added to the set of all values the variable has had
  - A read of a variable is allowed to return **ANY** value from the set of written values
- The JMM defines the rules by which values in the set are removed
  - **Synchronization actions** and **happens-before** relationship
- Programmers goal: through proper use of synchronization
  - Ensure the written set consists of only one value—that most recently written by some thread
- Basic safety guarantee: No “out-of-thin-air” values
  - A read always returns a value written by some thread, some time
  - Reads and writes of all basic data types are **atomic**  
Except for **long** and **double**

# Synchronization Actions

---

- **Program order**: The order in which statements appear in a program, as executed by a **single** thread
- **Synchronization order**: The order in which synchronization actions are executed
  - Always consistent with program order
  - Determines a partial order across actions in different threads
- **Example synchronization actions**:
  - Starting a thread; joining on a thread
  - Acquiring a lock; releasing a lock
- Some actions **synchronize-with** another action. Eg:
  - Starting a thread **synchronizes-with** the first action in that thread
  - Releasing a lock **synchronizes-with** all subsequent acquires of that lock

# Happens-Before Relationship

---

- Formal relationship between reads and writes of variables
  - Controls the possible values that a read of a variable may return
- For a given variable:
  - If a write of the value  $v_1$  happens-before the write of a value  $v_2$ , and the write of  $v_2$  happens-before a read, then that read may not return  $v_1$
  - Properly ordered reads and writes ensure a read can only return the most recently written value
- If an action **A synchronizes-with** an action **B** then **A happens-before B**
  - So correct use of synchronization ensures a read can only return the most recently written value

# Example: Unsynchronized Data Holder

---

```
- class DataHolder {  
    int data = 0;  
    boolean dataReady = false;  
    boolean isReady() { return dataReady; }  
    void setData(int newData) {  
        data = newData;  
        dataReady = true;  
    }  
    int getData() { return data; }  
}
```

```
    DataHolder h = ...;
```

```
// ... Thread-1           // Thread-2  
while (!h.isReady())      h.setData(42);  
    doOtherWork();  
int goodData = getData();
```

- No synchronization actions between threads, so no happens-before
  - Thread-1 need never have `isReady()` return true
  - Even if `isReady()` returns true, `getData()` may return 0 not 42

# Example: Synchronized Data Holder

---

```
- class SyncDataHolder {
    int data = 0;
    boolean dataReady = false;
    synchronized boolean isReady() {
        return dataReady; }
    synchronized void setData(int newData) {
        data = newData;
        dataReady = true;
    }
    synchronized int getData() { return data; }
}

        SyncDataHolder h = ...;

// ... Thread-1                // Thread-2
while (!h.isReady())           h.setData(42);
    doOtherWork();
int goodData = getData();
```

- Synchronization ensures if `isReady` is true then `getData` returns 42
  - Note: synchronization on `getData` not needed iff `isReady` always invoked first

# Volatile Variables

- Java also provides a “light” form of synchronization: **volatile** variables
- Volatile variables guarantee *visibility*
  - A read always sees the most recent value written to that variable
- Volatile variables can be used to provide thread-safety if and only if
  - The variable does not participate in invariants with other variables
  - The variable's value does not depend on the previous value
- Incrementing a volatile variable (++v) is not thread-safe
  - Increment operation looks atomic, but isn't
- Volatile variables are best suited for flags that have no dependencies
  - **while (!asleep) ++sheep;**
  - In the above, **asleep** should be **volatile** otherwise, the compiler can (HotSpot -server does) hoist it out of the loop, and transform the code:

```
if (!asleep)
    while (true)
        ++sheep;
```

# volatile Variables

---

- Reads/writes of variables declared **volatile** are synchronization actions
  - A write to a **volatile** variable synchronizes-with all subsequent reads of that **volatile** variable
  - So **volatile** variables provide ordering and visibility guarantees for themselves and any data they “protect”
- Reads/writes of variables declared **volatile** are also atomic
  - Extends the basic atomicity guarantee from the 32-bit types to the 64-bit types: **long** and **double**
  - NOTE: compound actions are NOT atomic, Eg:
    - ++ requires: read, increment, write

# Alternatives to Lock Based Synchronization

---

- Locking can be expensive
  - Overhead, contention, context-switching, memory effects
  - But not usually a bottleneck in well-designed code
- Java provides a few constructs to reduce need for locks
  - **final** fields do not require locks to read
  - **volatile** fields do not require locks to read or write
    - But updates like `++aVolatile` are **NOT** atomic
    - Correct use relies on Java 5.0 Memory Model
  - `java.util.concurrent.atomic` package

Atomic variables: get, set, increment, decrement

- Synchronization without use of locks requires detailed understanding of Java Memory Model (to be discussed later in class)
-

# Atomic Variables

---

- Holder classes for scalars, references and fields
    - `java.util.concurrent.atomic`
  - Support atomic operations
    - Get, set and arithmetic operations (where applicable)  
Increment, decrement operations
    - Compare-and-set (CAS)  
Used for lock-free algorithms - advanced topic
  - Abstraction of `volatile` variables
    - Similar memory model properties
    - But with atomicity properties of locks
  - Nine main classes:
    - { int, long, reference } X { value, field, array }
  - E.g. `AtomicInteger` useful for counters, sequence numbers, statistics gathering
-

# AtomicInteger Example

---

- Construction counter for monitoring/management

- Replace this: 

```
class Service {
    static int services;
    public Service() {
        synchronized(Service.class) {
            services++;
        }
    }
    // ...
}
```

- With this: 

```
class Service {
    static AtomicInteger services =
        new AtomicInteger();
    public Service() {
        services.getAndIncrement();
    }
    // ...
}
```

# Code Spec 6.20 Examples of Java's atomic objects.

---

## Sample Operations for AtomicInteger

```
boolean compareAndSet(expectedValue, updateValue);
```

- Atomically sets the value to `updateValue` if the current value is the same as `expectedValue`.

```
int getAndIncrement();
```

- Atomically reads the current value and increments the current value by one.

# Summary of Today's Lecture

---

- Java threads
- Java locks
- Java memory model
- Volatile and Atomic variables in Java