CS6868 - Concurrent Programming

Java Concurrency

V. Krishna Nandivada

IIT Madras

Java Threads Vs Processes

- Typically each instance of JVM creates a single process.
- Each process creates one or more threads.
  - Main thread creates the others.

Java Threads

- Each Java thread is an object - instance of the Java Thread class.
- An application that creates an instance of Thread must provide the code that will run in that thread.
  - `implement Runnable` interface.
  - Provide an implementation of the `run` method.

  or
  - `extend Thread` class
  - Provide an overridden implementation of the `run` method.

  Adv/Disadv??
  - (Hint) Java allows single inheritance.
  - Extending thread class ⇒ cannot extend any other class.

What can a thread do?

- Start executing the thread body specified in the `run` method.
- Sleep - `Thread.sleep(..)`
- Wait for child threads to finish: `ch.join()`
- Communicate with other threads.

Communication via synchronized methods

- `synchronized` statements and methods. Making a methods synchronized has two effects:
  - It is not possible for two invocations of synchronized methods on the same object to interleave.
  - When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object.
    - Guarantees that changes to the state of the object are visible to all threads.
- Constructors cannot be synchronized. Why? Only the object creating threads should call the constructor.
- synchronized methods ensure that there is no thread interference.
- Q: Too many synchronized methods. Disadv?

How do synchronized methods/statements work?

- Each object has an associated intrinsic lock.
- When a thread invokes a synchronized method,
  - automatically acquires the intrinsic lock for that method's object
  - releases the lock when the method returns (normal or via exception).
- What if a thread invokes a synchronized method recursively?

Updating shared variables

Java guarantees that following actions would be atomic.

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).
- Any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.
- We can also declare a variable (of some types) as atomic.

Atomic variables

- The `java.util.concurrent.atomic` package defines classes supporting atomic operations on single variables.
- All classes of `get` and `set` operations.
- Like `volatile` variables' write operation, the `set` operation has an happens-before relation with the corresponding `get` operation.

Atomic variables - Example

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
                // guarantees thread non-interference.
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Atomic variables (contd)

- Supported classes:

AtomicBoolean	AtomicInteger	AtomicIntegerArray
...	AtomicLong	AtomicLongArray
...	LongAccumulator	LongAdder

- All support: `boolean compareAndSet(expectedValue, updateValue)`
- CAS operation: How to use it to realize synchronization?
- Building blocks for implementing 'non-blocking' data structures.

Happens before relation

- Sequential order: Each action in a thread happens-before every action in that thread that comes later in the program's order.
- Unlock $\rightarrow$ Lock: An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor.
- Volatile writes: A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
- A call to start on a thread happens-before any action in the started thread.
- All actions in a thread happen-before any other thread successfully returns from a join on that thread.

Happens-before relation is transitive.

Deadlock, Livelock and Starvation

- Deadlock: two or more threads are blocked forever, waiting for each other.
- Starvation: a thread is unable to gain regular access to shared resources and is unable to make progress.
- Livelock: threads are not blocked, but are not making any progress.

Guarded blocks

- A way to coordinate with others.
- A guarded block
  - polls a condition that has to be true to proceed.
  - other threads set that condition

```
public void guardedEntry(){
    while (!flag) ;
   // flag is set. Inefficient.
}
```

Guarded blocks (contd)

```
public synchronized void guardedEntry() {
    // Check once and "wait"
    while(!flag) { // Loop is needed.
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // flag is set. Efficient.
}
```

- Q: Why synchronized?
- `notify` vs `notifyAll`

Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Helps write reliable code.
- cannot be corrupted by thread interference or observed in an inconsistent state.
- Creating many immutable objects Vs updating existing objects.
  - Cost of object creation, GC
  - Code needed to protect mutable objects from corruption.

Example

```
public class SynchronizedRGB {
  private int red;   // between 0 - 255.
  private int green; // between 0 - 255.
  private int blue;  // between 0 - 255.
  private String name;

  public synchronized void set(int r, int g, int b,
                               String n) {..}
  public synchronized int getRGB() {
      return ((red << 16) | (green << 8) | blue);
  }
  public synchronized String getName() { return name; }
  public synchronized void invert() {
      red=255-red; green=255-green; blue=255-blue;
      name="Inverse of " + name;
  } }
```

Example (contd)

```
...
SynchronizedRGB color =
    new SynchronizedRGB(0, 0, 0, "Black");

int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

What if another threads updates the color object after Statement 1?

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

Such issues do not arise with immutable objects

Mutable to Immutable

General guidelines:

- Don't provide 'setter' methods.
- Make all fields final and private.
- Don't allow subclasses to override methods or provide 'setter' methods.
  - declare the class as final.
  - make constructor final and provide a factory method.
- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects.
  - Don't share references to the mutable objects. Copy and share if required.

Mutable to Immutable (Example)

```
final public class ImmutableRGB {
  final private int red;   // between 0 - 255.
  final private int green; // between 0 - 255.
  final private int blue;  // between 0 - 255.
  final private String name;

  public /*synchronized*/ int getRGB() {
        return ((red << 16) | (green << 8) | blue); }
  public /*synchronized*/ String getName()
      { return name; }
  public /*synchronized*/ ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                        255 - green, 255 - blue,
                        "Inverse of " + name);
    } }
```

No synchronized methods required!

Deadlocks in Locks

```
public class Deadlock {
 class Friend {
  private final String name;
  public Friend(String name){this.name = name; }
  public String getName() {return this.name; }
  public synchronized void bow(Friend bower) {
   System.out.format("%s: %s" +" has bowed to me!",
     this.name, bower.getName());
   bower.bowBack(this); }
  public synchronized void bowBack(Friend bower) {
   System.out.format("%s:%s"+" has bowed back!",
    this.name, bower.getName()); } }
```

Deadlocks in Locks (contd)

```
public static void main(String[] args) {
    final Friend alpha = new Friend("Alpha");
    final Friend beta = new Friend("Beta");
    new Thread(new Runnable() {
        public void run() { alpha.bow(beta); }
    }).start();
    new Thread(new Runnable() {
        public void run() { beta.bow(alpha); }
    }).start();
} }
```

Avoid deadlocks in Locks

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) { this.name=name;}

        public String getName() { return this.name;}
```

Avoid deadlocks in Locks (cont.)

```
    public boolean impendingBow(Friend bower) {
        Boolean myLock = false;
        Boolean yourLock = false;
        try {
            myLock = lock.tryLock();
            yourLock = bower.lock.tryLock();
        } finally {
            if (! (myLock && yourLock)) {
                if (myLock) {
                    lock.unlock();
                }
                if (yourLock) {
                    bower.lock.unlock();
                } } }
        return myLock && yourLock;
    }
```

Avoid deadlocks in Locks (cont.)

```
public void bow(Friend bower) {
    if (impendingBow(bower)) {
        try {
            System.out.format("%s: %s has"
                + " bowed to me!",
                this.name, bower.getName());
            bower.bowBack(this);
        } finally {
            lock.unlock();
            bower.lock.unlock(); }
    } else {
        System.out.format("%s: %s started"
            + " to bow to me, but saw that"
            + " I was already bowing to him.",
            this.name, bower.getName());
    } }
```

Avoid deadlocks in Locks (cont.)

```
    public void bowBack(Friend bower) {
        System.out.format("%s: %s has" +
            " bowed back to me!%n",
            this.name, bower.getName());
    }
}
```

Avoid deadlocks in Locks (cont.)

```
class BowLoop implements Runnable {
    private Friend bower;
    private Friend bowee;

    public BowLoop(Friend bower, Friend bowee) {
        this.bower = bower; this.bowee = bowee;}
    public void run() {
        Random random = new Random();
        for (;;) {
            try {
                Thread.sleep(random.nextInt(10));
            } catch (InterruptedException e) {}
            bowee.bow(bower);
        } } }
```

Avoid deadlocks in Locks (cont.)

```
    public static void main(String[] args) {
        final Friend alpha = new Friend("Alpha");
        final Friend beta = new Friend("Beta");
        new Thread(new BowLoop(alpha, beta)).start();
        new Thread(new BowLoop(beta, alpha)).start();
    }
}
```

Not covered

- Executors.
- Threadpools
- Fork/Join framework
- Concurrent Collections.

References

Java online resources
Oracle man pages.

¡++¿