Lecture 25: Java Threads, Java synchronized statement

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
One possible solution to Worksheet #24

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

1. // Start of thread t0 (main program)
2. sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
3. // Compute sum1 (lower half) and sum2 (upper half) in parallel
4. final int len = X.length;
5. Thread t1 = new Thread(() -> {
6.     for(int i=0 ; i < len/2 ; i++) sum1+=X[i];
7.     t1.start();
8.     Thread t2 = new Thread(() -> {
9.         for(int i=len/2 ; i < len ; i++) sum2+=X[i];
10.        t2.start();
11.        int sum = sum1 + sum2; //data race between t0 & t1, and t0 & t2
12.        t1.join(); t2.join();


2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements.

1. // Start of thread t0 (main program)
2. sum = 0; // static int field
3. Object a = new ... ;
4. Object b = new ... ;
5. Thread t1 = new Thread(() ->
6. { synchronized(a) { sum++; } });
7. Thread t2 = new Thread(() ->
8. { synchronized(b) { sum++; } });
9. t1.start();
10. t2.start(); // data race between t1 & t2
11. t1.join(); t2.join();
Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program’s main() method.

Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```java
public class Thread extends Object implements Runnable {
    // Creates a new Thread
    Thread() { ... } // Creates a new Thread with Runnable object r
    void run() { ... } // Code to be executed by the thread
    // Case 1: If this thread was created by another thread
    // Case 2: If this class is subclassed, the
    // in the subclass is called
    void start() { ... } // Causes this thread to start executing
    void join() { ... } // Wait for this thread to die
    void join(long m) // Wait at most m milliseconds for thread to die
    static Thread currentThread() // Returns currently executing thread
    ...
}
```

A lambda can be passed as a Runnable.
Every Java object has an associated lock acquired via:

- **synchronized statements**
  - `synchronized( foo ) { // acquire foo’s lock
    // execute code while holding foo’s lock
  } // release foo’s lock

- **synchronized methods**
  - `public synchronized void op1() { // acquire ‘this’ lock
    // execute method while holding ‘this’ lock
  } // release ‘this’ lock`

Java language does not enforce any relationship between object used for locking and objects accessed in isolated code

- If same object is used for locking and data access, then the object behaves like a monitor

Locking and unlocking are **automatic**

- Locks are released when a synchronized block exits
  - By normal means: end of block reached, `return`, `break`
  - When an exception is thrown and not caught
Implementation of Java synchronized statements/methods

• Every object has an associated lock
• “synchronized” is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  – monitorenter requests “ownership” of the object’s lock
  – monitorexit releases “ownership” of the object’s lock
• If a thread performing monitorenter does not gain ownership of the lock (because another thread already owns it), it is placed in an unordered “entry set” for the object’s lock
Monitors – a Diagrammatic summary

Figure 20-1. A Java monitor.

Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif
What if you want to wait for shared state to satisfy a desired property? (Bounded Buffer Example)

1. public synchronized void insert(Object item) { // producer
2.     // TODO: wait till count < BUFFER SIZE
3.     ++count;
4.     buffer[in] = item;
5.     in = (in + 1) % BUFFER SIZE;
6.     // TODO: notify consumers that an insert has been performed
7. }

9. public synchronized Object remove() { // consumer
10.    Object item;
11.    // TODO: wait till count > 0
12.    --count;
13.    item = buffer[out];
14.    out = (out + 1) % BUFFER SIZE;
15.    // TODO: notify producers that a remove() has been performed
16.    return item;
17. }
The Java wait() Method

- A thread can perform a `wait()` method on an object that it owns:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set

- Causes thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

- Since interrupts and spurious wake-ups are possible, this method should always be used in a loop e.g.,

  ```java
  synchronized (obj) {
    while (<condition does not hold>)
      obj.wait();
    ... // Perform action appropriate to condition
  }
  ```

- Java’s `wait-notify` is related to “condition variables” in POSIX threads
Entry and Wait Sets
The notify() Method

When a thread calls `notify()`, the following occurs:

1. selects an arbitrary thread $T$ from the wait set
2. moves $T$ to the entry set
3. sets $T$ to Runnable

$T$ can now compete for the object's lock again
Multiple Notifications

• `notify()` selects an arbitrary thread from the wait set.
  – This may not be the thread that you want to be selected.
  – Java does not allow you to specify the thread to be selected

• `notifyAll()` removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.

• `notifyAll()` is a conservative strategy that works best when multiple threads may be in the wait set
public synchronized void insert(Object item) {
    while (count == BUFFER SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) {
        }
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER SIZE;
        notify();
    }
}
public synchronized Object remove() {
    Object item;
    while (count == 0) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    notify();
    return item;
}
Complete Bounded Buffer using Java Synchronization

```java
public class BoundedBuffer implements Buffer {

    private static final int BUFFER_SIZE = 5;
    private int count, in, out;
    private Object[] buffer;

    public BoundedBuffer() { // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }

    public synchronized void insert(Object item) { // See previous slides
    }

    public synchronized Object remove() { // See previous slides
    }

}
```
Announcements

• The entire written + programming (Checkpoint #2) is due by Monday, April 5th at 11:59pm
• Lab 6 is due Tuesday, April 6th at 12pm (noon)
Consider the case when multiple threads call `insert()` and `remove()` methods concurrently for a single `BoundedBuffer` instance with `SIZE >= 1`.

1) Can you provide an example in which the wait set includes a thread waiting at line 2 in `insert()` and a thread waiting at line 11 in `remove()`, in slide 8? If not, why not?

2) How would the code behave if all wait/notify calls (lines 2, 6, 11, 15) were removed from the `insert()` and `remove()` methods in slide 8?