Object Oriented Programming

Week 10 Part 2
Multi-threading: Synchronizing Threads

Lecture

- Race Conditions
- Synchronization
- Method Synchronization
- Locks
- Statement Synchronization
- Deadlock

Race Conditions

The Problem: Race Conditions

- A *Race Condition* occurs when one thread overwrites the result of a second thread stored in a shared variable while the first thread was sleeping. I.E.
  1) A thread is preempted between the time it reads a value and the time it writes the variable
  2) The second thread updates the variable while the first thread is preempted, overwriting the value
  3) The second thread's update is lost.
- It is called a race condition because the two threads race to see which one updates the variable
- Race conditions are *extremely* hard to debug because they occur sporadically.

Race Condition Example

- Thread 1 read x
- Thread 1 preempted
- Thread 2 read x
- Thread 2 add x + 1
- Thread 2 write x
- Thread 1 restarts
- Thread 1 add x + 1
- Thread 1 write x

- X = 1
- X = 1
- X = 1
- X = 1
- X = 2
- X = 2 (Thread 1 has 1)
- X = 2
- X = 2 (should be 3)

RC Example Discussion

- When Thread 1 is preempted, the value of x is 1
- When Thread 1 restarts, it still thinks the value of x is 1 even though Thread 2 has changed it to 2
- Thread 1, not knowing that Thread 2 has run, updates the value writing over the value that Thread 2 wrote

Race Condition: `SimpleThreads`

Shared variable: sharedVar

```
public class SimpleThreads {

    static long startTime = 0;
    public static int sharedVar = 0;

    public static long getStartTime() {
        return startTime;
    }

    public static void printMessage(String message) {
        System.out.format("%s, running: %d ms, %s%n",
                Thread.currentThread().getName(),
                System.currentTimeMillis() - startTime,
                message);
    }

    public static void main(String args[])
            throws InterruptedException {
        startTime = System.currentTimeMillis();
        printMessage("Starting main()");

        Thread t1 = new Thread(new RunnableThread(1000), "Thread 1");
        t1.start();
        printMessage("Thread 1 started");

        Thread t2 = new Thread(new RunnableThread(500), "Thread 2");
        t2.start();
        printMessage("Thread 2 started");

        t1.join();
        printMessage("Ending main()");
    }
}
```

Race Condition: `RunnableThread`

Make calculation using shared variable

Enter blocked state by sleeping

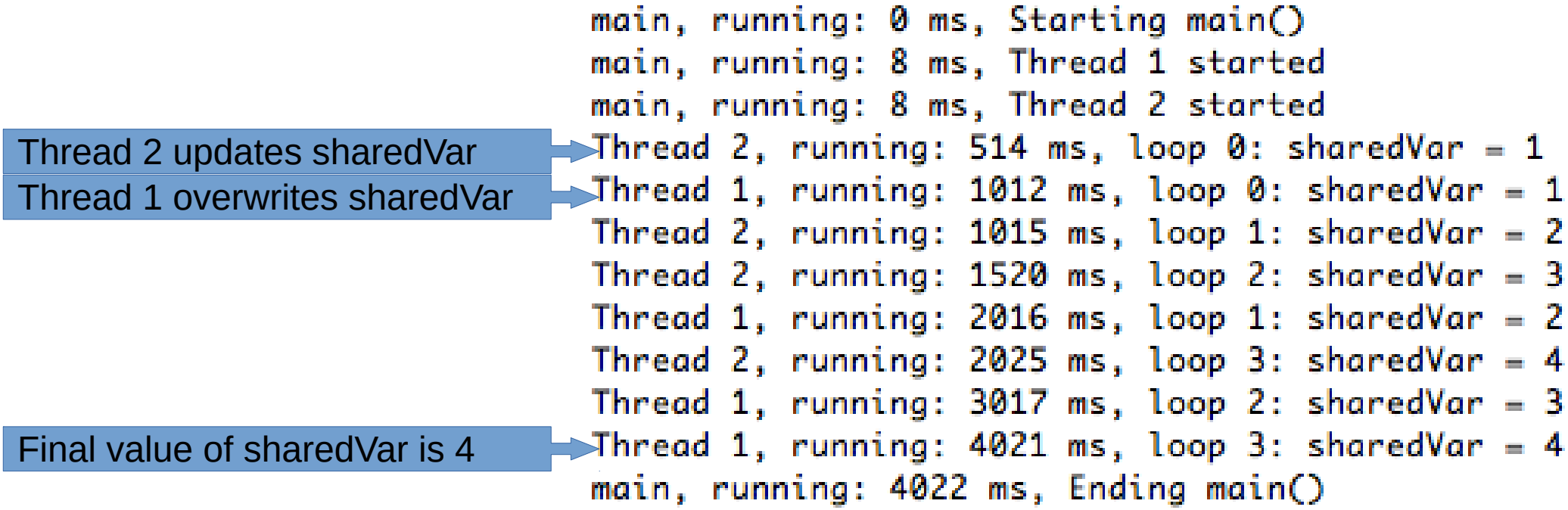
Update the shared variable

```
public class RunnableThread implements Runnable {

    long sleepTime = 0;
    String msg = new String();
    int tempVar;

    RunnableThread(long sleepTime) {
        this.sleepTime = sleepTime;
    }

    @Override
    public void run() {
        for (int i = 0; i < 4; i++) {
            tempVar = SimpleThreads.sharedVar + 1;
            try {
                Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
                SimpleThreads.printMessage("Interrupted: "
                                           + e.getMessage());
            }
            SimpleThreads.sharedVar = tempVar;
            msg = String.format("loop %s: sharedVar = %d",
                    i, SimpleThreads.sharedVar);
            SimpleThreads.printMessage(msg);
        }
    }
}
```

Race Condition: Expectations

- Each thread runs updates the shared variable four times adding one each time
- There are two threads running
- We would expect the value of variable to be 8 (2 * 4) when the program terminates

Race Condition: Output



Synchronization

- *Synchronization* allows threads to use the same variables
- Threads signal to each other using a *semaphore* or *monitor*
  - A monitor is an object that locks a sequence of code so only one thread can use it at a time
    - Only one thread at a time can run that code
  - If another thread tries to enter a monitor it is blocked

Synchronization

Java Synchronization

- Java provides ways to synchronize threads
  - Synchronized methods
  - Synchronized statements

Synchronized Methods

Synchronized Methods

- *Synchronized methods* allow only one thread to execute a method at a time
- If a second thread tries to execute the method, it is blocked until the first method finishes.
- Synchronized methods are declared in java using the keyword `synchronized` to the declaration of the method

Example: Synchronized Method

- First, notices that we have a design error in our SimpleThreads class
  - The variable sharedVar is declared to be public giving the SharedThreads class no control over its access
  - We will fix that by adding the increment method
- We can then make that method synchronized

Synchronized : SimpleThreads

Make sharedVar private

Access provided by sharedIncrement

Getter need not be synchronized

```
public class SimpleThreads {

    private static long startTime = 0;
    private static int sharedVar = 0;

    public static synchronized void sharedIncrement() {
        sharedVar = sharedVar + 1;
    }

    public static int getSharedVar () {
        return sharedVar;
    }

    public static long getStartTime() {
        return startTime;
    }
// The rest is unchanged
```

The getter need not be synchronized, because we only risk overwriting another thread's work if we write to the variable. Reading is safe.

Synchronized: `RunnableThread`

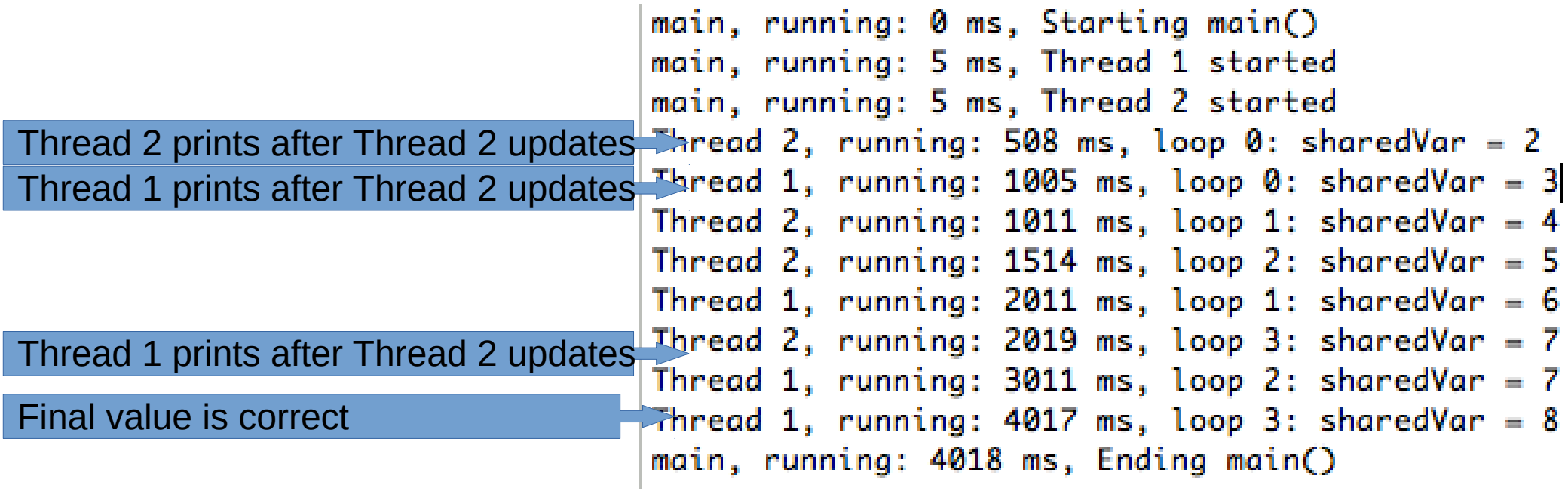
```
public class RunnableThread implements Runnable {

    long sleepTime = 0;
    String msg = new String();
    int tempVar;

    RunnableThread(long sleepTime) {
        this.sleepTime = sleepTime;
    }

    @Override
    public void run() {
        for (int i = 0; i < 4; i++) {
            SimpleThreads.sharedIncrement();
            try {
                Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
                SimpleThreads.printMessage("Interrupted: "
                                           + e.getMessage());
            }
            msg = String.format("loop %s: sharedVar = %d",
                    i, SimpleThreads.getSharedVar());
            SimpleThreads.printMessage(msg);
        }
    }
}
```

Update using SimpleThreads method

Enter blocked state by sleeping

Get the value using SimpleThread method

Synchronized: Output



- The anomaly is now cause by the distance in time between the update and the print.
- We can fix this by synchronizing both the increment and the print

Synchronized incrementAndPrint: `SimpleThreads`

Need to pass in loop counter

Print message in synchronize method

```
public static synchronized void incrementAndPrint(int i) {
    sharedVar = sharedVar + 1;
    printMessage(String.format("loop %d: sharedVar = %d",
            i, sharedVar));
}
```

Updating and printing now occur atomically

Synchronized incrementAndPrint: `RunnableThread`

```
public void run() {
    for (int i = 0; i < 4; i++) {
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
            SimpleThreads.printMessage("Interrupted: "
                                       + e.getMessage());
        }
        SimpleThreads.incrementAndPrint(i);
    }
}
```

Call incrementAndPrint with loop variable

Synchronized incrementAndPrint: Output

```
main, running: 0 ms, Starting main()
main, running: 6 ms, Thread 1 started
main, running: 6 ms, Thread 2 started
Thread 2, running: 512 ms, loop 0: sharedVar = 1
Thread 1, running: 1011 ms, loop 0: sharedVar = 2
Thread 2, running: 1015 ms, loop 1: sharedVar = 3
Thread 2, running: 1519 ms, loop 2: sharedVar = 4
Thread 1, running: 2017 ms, loop 1: sharedVar = 5
Thread 2, running: 2025 ms, loop 3: sharedVar = 6
Thread 1, running: 3019 ms, loop 2: sharedVar = 7
Thread 1, running: 4021 ms, loop 3: sharedVar = 8
main, running: 4022 ms, Ending main()
```

- The anomaly is fixed

Locks

Locks

- *Locks* are objects that insure that only one thread uses a method
- Each object and class has an *intrinsic lock* associated with it
  - Methods declared to be static used the class's lock
  - Non-static methods use the objects lock
- To use the object or classes lock, call the the synchronized() method.

Synchronized incrementAndPrint: `SimpleThreads`

Synchronized on class

```
public static void incrementAndPrint(int i) {
    synchronized(SimpleThreads.class) {
        sharedVar = sharedVar + 1;
        printMessage(String.format("loop %d: sharedVar = %d",
                i, sharedVar));
    }
}
```

Because the method is static, we need to increment on SimpleThread's class object.

Synchronized on this

```
public void incrementAndPrint(int i) {
    synchronized(this) {
        sharedVar = sharedVar + 1;
        printMessage(String.format("loop %d: sharedVar = %d",
                i, sharedVar));
    }
}
```

If the method were not static, we could synchronize on the object's lock. However, then we would need to pass the object to all of the threads it created so the threads could synchronize on the particular object's lock.

Synchronized Statements

Synchronized Statements

- *Synchronized statements* are blocks of statements synchronized using locks.
- We can increase the granularity of the locking by using synchronize statements
  - We create multiple locks, then lock blocks of statements with different locks
  - One thread may do one block while another does a different blocks
  - Locks are created whenever we create an object.

Synchronized Statements: `SimpleThreads`

Create two Objects

Inc synchronized on Object lock1

Print synchronized on Object lock2

```
static Object lock1 = new Object();
static Object lock2 = new Object();

public static void incrementAndPrint(int i) {
    synchronized(lock1) {
        sharedVar = sharedVar + 1;
    }
    synchronized(lock2) {
        printMessage(String.format("loop %d: sharedVar = %d",
                i, sharedVar));
    }
}
```

Because the method is static, we need to use static Objects.

Deadlock

New Problem: Deadlock

- *Deadlock* occurs when one thread waits on a lock owned by another thread, but the first thread is waiting on a lock owned by the second thread
  - More generally, there is a circular wait, in which each thread is waiting on another.
  - That is, the deadlock may involve more than two threads.

Deadlock: Example

- By adding the second lock, we have introduced the possibility of deadlock.
  - Thread 1 could increment then start printing acquiring lock2
  - Thread 2, following close behind starts incrementing, then starts to print, but is blocked by Thread 1.
  - Thread 1 continues on, then starts to increment again. It is blocked because Thread 2 owns lock 1
  - Neither thread progresses and the program stalls