CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #13, Concurrency, Interference, and Synchronization

John Ridgway

March 12, 2015

Concurrency and Threads

- Computers are capable of doing more than one thing at the same time.
- Some computer hardware can actually have two processes going on simultaneously using different processors.
- Other computers use **timesharing**, a system where a single processor moves among more than one task, making progress on each in turn and maintaining the context of each.

Concurrency and Threads

- The more programs interact with the real world, in such ways as monitoring sensors or awaiting commands, the more important concurrency becomes.
- In this lecture we'll take a brief look at some of the major issues with concurrency, and the facilities Java offers to deal with them.
- You'll learn much more about concurrency in CMPSCI 230 and 377.

`Runnable` and `Thread`

- The `Runnable` interface requires a single public method called `run`.
- The `Thread` class has a constructor that takes one argument of type `Runnable`.
- If we create a `Thread` object using a `Runnable`, we can then `start` the thread, at which point the `Runnable`'s `run` method starts processing in parallel with the main thread. We can later interrupt the thread.

A Runnable Example

```
public class Counter {
  private int count;
  public Counter() { count = 0; }
  public void increment() { count += 1; }
  public String toString() {
    return "Count is:\t" + count; } }

public class Increase implements Runnable {
  private Counter c;
  private int amount;
  public Increase(Counter c, int amount) {
    this.c = c;
    this.amount = amount; }
  public void run() {
    for (int i = 1; i <= amount; i += 1) {
      c.increment(); } } }
```

A Runnable Example

This is the simplest way to run an `Increase` operation, in the main thread just as we would run anything else. The `count` field of `c` is increased to 10000 and the output is thus `"Count is: 10000"`.

```
public class Demo01 {
  public static void main(String[] args) {
    Counter c = new Counter();
    Runnable r = new Increase(c, 10000);
    r.run();
    System.out.println(c); } }
```

Clicker Question #1

What is the output of this application?

```
public class Clicker1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    Runnable r1 = new Increase(c, 10000);
    Runnable r2 = new Increase(c, -5000);
    r1.run();
    r2.run();
    System.out.println(c); } }
```

A. an exception

B. `"Count is: 0"`

C. `"Count is: 5000"`

D. `"Count is: 10000"`

Using Multiple Threads

```
public class Demo02 {
  public static void main(String[] args) {
    Counter c = new Counter();
    Runnable r = new Increase(c, 10000);
    Thread t = new Thread(r);
    t.start();
    System.out.println(c); } } //note DJW typos
```

- You might think we get 10000 again, but on various trials John got numbers ranging from 91 to 603. (The behavior is *non-deterministic*!)
- The issue is that the `Increase` operation goes on *in parallel with* the main thread, and it only gets so far before the main thread prints `c`.

Joining Two Threads

```
public class Demo03 {
  public static void main(String[] args)
    throws InterruptedException
  {
    Counter c = new Counter();
    Runnable r = new Increase(c, 10000);
    Thread t = new Thread(r);
    t.start();
    t.join();
    System.out.println(c); } }
```

This time we do get 10000, because the `join` method suspends the main thread until the new thread is finished. So we only reach the print statement after the `Increase` is finished.

Interference

```
public class Demo04 {
  public static void main(String[] args)
      throws InterruptedException {
    Counter c = new Counter();
    Runnable r1 = new Increase(c, 5000);
    Runnable r2 = new Increase(c, 5000);
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start();   t2.start();   t1.join();   t2.join();
    System.out.println(c); } }
```

Increasing `c` by 5000 twice ought to give us 10000 again. But on John's first five trials the results were 7426, 7438, 6898, 7923, and 6595. Why?

Interference

- The problem here is **interference** between the threads. Both threads are trying to access the same variable `count`, by reading it, increasing it by 1, and writing it back.

- Suppose that `t1` reads the value 2317, then `t2` reads 2317 before `t1` writes back. Then `t1` writes back 2318, and `t2` also writes 2318.
- There were two attempts to increment the variable, but it only went up by one.

Clicker Question #2

In the previous example, what happens if several `increments` by `t1` happen between the read step of a `t2` `increment` and its write step?

A. All of them work, but the `t2` one doesn't.

B. None of the `t1` increments have any effect.

C. Only the last `t1` increment works.

D. An exception is thrown.

Interference

- When does this happen? It depends on the exact timing of the read and write operations of the two threads.
- These are governed by the operating system, which is timesharing the processor or processors among the threads.
- Exactly what happens may well depend on the other loads on the processor, which is why we get the nondeterministic behavior.

Synchronization

- We can't have this sort of thing happening whenever two different processes are sharing a resource.
- The basic idea to avoid the problem is to make sure that only one process has permission to access the resource at one time.
- In a hardware system, we might have a physical token that a process needs to have to get at the resource, and pass the token among the processes.

Synchronization

- Fortunately Java lets us coordinate the processes at a higher level of abstraction, through a mechanism called **method-level synchronization**.
- We create a new class `SyncCounter` that is identical to `Counter` except for one thing: The method `increment` is declared to be `synchronized`. We then make a new class `SyncIncrease` that uses SyncCounter but has no other changes.

Synchronized Code

```
public class SyncCounter {
  private int count;
  public SyncCounter() { count = 0; }
  public synchronized void increment() { count += 1; }
  public String toString() {
    return "Count is:\t" + count; } }

public class SyncIncrease implements Runnable {
  private SyncCounter c;
  private int amount;
  public SyncIncrease(SyncCounter c, int amount) {
    this.c = c;
    this.amount = amount; }
  public void run() {
    for (int i = 1; i <= amount; i++) {
      c.increment(); } } }
```

Synchronization

```
public class Demo05 {
  public static void main(String[] args) throws InterruptedException {
    SyncCounter c = new SyncCounter();
    Runnable r1 = new SyncIncrease(c, 5000);
    Runnable r2 = new SyncIncrease(c, 5000);
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start();   t2.start();
    t1.join();    t2.join();
    System.out.println(c); } }
```

This version consistently prints 10000. What's different is that the `increment` method now plays nicely with other copies of itself, waiting for them to finish before going on.

Synchronized Queues

- Queues are often shared among multiple processes. For example, client processes might put their requests on a queue for server processes to take off and satisfy.
- We would hate for an element to be put on the queue and never come off, or to come off twice. Interference among processes, like we have just seen, could easily cause this to happen, or cause the queue to throw an exception.

Clicker Question #3

Suppose `q` is the array of an array-based queue, and one thread tries to `enqueue(x)` at about the same time as another thread tries to `enqueue(y)`. The rear index is originally `r`. Which of these is not possible?

A. `x` goes in position `r+1`, `y` in `r+2`

B. position `r+1` is `null`, `x` goes in `r+2`

C. `y` goes in `r+1`, position `r+2` is `null`

D. all three of the results above are possible

Synchronized Queue Example

- DJW have an example on pages 348-352 using the `SyncCounter` class. They have a new version of `Increase` that takes a `Queue<Integer>` as a parameter, and runs by taking integers off the queue and adding them to the counter, until the queue is empty.
- Their `Demo06` class fills a queue with the numbers from 1 to 100, then creates two of these `Increase` processes to together empty the queue and add its members to a common `SyncCounter`.

Synchronized Queue Example

- This works most but not all of the time. In their trials they got the correct answer of 5050 the first ten times, but the eleventh time got 4980 and the sixteenth time threw a `NullPointerException`.
- They fix this by writing a new class called `SyncArrayBndQueue` that has the word `synchronized` on its methods. All the methods then play nicely with one another.

Queues in `java.util`

- Remember that `Queue` in the `Collections` package is an interface, implemented by nine different classes.
- Some of these classes are **thread-safe**, meaning that its operations are synchronized to prevent interference among them.
- The older Java classes like `Vector` are thread-safe, but modern Java offers cheaper, simpler unsynchronized classes like `ArrayList`, that may be used in the many situations where there aren't competing threads, as well as specific thread-safe classes.

Average Waiting Time

- Queues are perhaps most commonly used to **simulate** real-world waiting situations. DJW offer a case study that allows a user to compare the average waiting times for **customers** when varying numbers of **servers** are available, each with its own queue.
- More servers cost more money, but customers are happier with shorter waiting times.

- Exploring the **tradeoffs** in simulation is cheaper than running real-world experiments.
- But the simulation is only as good as its **model** of customer and server behavior.

Average Waiting Time

- Different customers take different amounts of time to be served, and arrive for service with different intervals of time between them.
- An easy way to get such a variation is to use a `Random` object to generate numbers uniformly, within given ranges, for the service times and the intervals. The ranges might be based on actual experimental data – DJW's simulator allows the user to provide the maximum and minimum service times and interval times.

The Randomized Simulation

- The `Simulation` class will create the customers and queues, run the experiment, and report the results.
- A `Customer` object has an arrival time and a service time on its creation, and will have a finish time assigned during the simulation. We can calculate how long the `Customer` was waiting in the queue – our output will be the average waiting time for the set of customers.

The Randomized Simulation (continued)

- We'll need a `CustomerGenerator` class to generate these objects using a `Random` object.
- A `Queue` will correspond to each server. When a customer arrives, she will enter the shortest available queue (the one with the fewest customers in it).

The Randomized Simulation (continued)

- The server will dequeue a customer when it finishes with the previous one, unless its queue is empty.
- The dequeued customer gets a finish time, computed by adding her service time to the time she is dequeued.
- But our existing queue classes actually won't suffice to model the queues we want.

The GlassQueue Class

```
public class GlassQueue<T>
  extends LinkedUnbndQueue<T> {
  public int size() { return numElements; }
  public T peekFront() { return head.getData(); }
  public T peekRear() { return tail.getData(); } }
```

- This is an example of the use of inheritance.
- In the case study, we want a queue that is able to report its size and to give pointers to its front and rear elements. (DJW call this a "glass queue" because it is transparent.)
- Rather than write a new class repeating the old code, we can extend the existing class.

Clicker Question #4

```
public class GlassQueue<T>
  extends LinkedUnbndQueue<T> {
  public int size() { return numElements; }
  public T peekFront() { return head.getData(); }
  public T peekRear() { return tail.getData(); } }
```

Why can this class's code access the fields `numElements`, `head`, and `tail`?

A. they are public fields

B. they are protected fields of the superclass

C. it can't because they are private fields of the superclass – this code won't compile

D. they are private fields but it doesn't matter