CMPSCI 187: Programming With Data Structures

Lecture #20: Concurrency and a Case Study
David Mix Barrington
24 October 2012

Concurrency and a Case Study

- Concurrency and Threads
- Example: `Counter`, `Increase`, and `Runnable`
- Interference of Threads and Synchronization
- Synchronized Queues and the `java.util` Queues
- The Average Waiting Time Case Study
- A Simulation With Randomized Customers
- The `GlassQueue` Class

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.

- 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.

Example: `Counter`, `Increase`, and `Runnable`

- The `Runnable` interface requires a class to have a public method named `run`, which can then be put into a `Thread` object. This allows it to be executed in another **thread** of the Java execution besides the **main thread**.

```
public class Counter {
   private int count;
   public Counter( ) {count = 0;}
   public void increment( ) {count++;}
   public void 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++) c.increment;}}
```

- The `Thread` created below should eventually increase the value stored in `c` to 1000. But DJW report that when they ran this, they got varying answers under 100. The reason is that they start running the thread and then *in parallel* ask the main thread to print the value stored in c. They find out how far the thread t has gotten by the time the main thread asks c its value.

- Note that `Thread` objects can throw `InterruptedException` objects, which are checked and thus must be caught or thrown beyond the `main` method.

```
public class Demo02 {
   public static void main (String[ ] args)
         throws InterruptedException {
      Counter c = new Counter( );
      Runnable r = new Increase(c, 1000);
      Thread t = new Thread(r);
      t.start( );
      System.out.println(c);}} // note typo on p. 343
```

Interference of Threads and Synchronization

- DJW continue with more similar examples. In `Demo03`, they start the thread `t` and then say `t.join( )`, which causes the main thread to wait until `t` is finished. In this case they get the value of 1000 stored in `c`.

- In `Demo04` they create two threads, `t1` and `t2`, each of which should increase `c` by 5000. They then start both and ask both to join. This *should* set `c` to 10000, but they get varying values in the 9000's.

- The problem is **interference** between the threads. Each thread is reading the value in c, increasing it by 1, and writing it back. Suppose `t1` writes its value *between* `t2`'s read and write operations? Its value is overwritten by `t2`'s, and the increase `t1` wanted to make is never made.

- In Demo05 they replace the Counter with a SyncCounter in which the increment method is `public synchronized void increment`. Then only one thread can have access to the method at a time, and we get 10000.

Synchronized Queues and the `java.util` Queues

- DJW's `Demo06` creates an `ArrayBndQueue<Integer>` object, loads it with the numbers from 1 to 100, then creates two threads with variants of the `Increase` class that are each to take numbers from the queue and add their value into the `SyncCounter`.

- Though the counter is synchronized, the queue is not. They find that *intermittently*, a number will not be added in or an exception will occur. This apparently was because the two threads tried to dequeue at about the same time. They then build a `SyncArrayBndQueue` class that is synchronized.

- The Java library contains both synchronized (**thread-safe**) and unsynchronized classes among the nine implementations of the `Queue` interface. 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.

The Average Waiting Time Case Study

- 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** with varying numbers of **servers** 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 probably cheaper than running real-world experiments. But the simulation is only as accurate as its **model** of customer and server behavior.

- 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 max and min service times and interval times.

A Simulation With Randomized Customers

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

- We have here a nice 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.

- Note that the new constructor calls the old one using the word `super`. This process repeats through the inheritance hierarchy all the way to the `Object` constructor. The zero-parameter constructor exists even if not declared.

```
public class GlassQueue<T> extends ArrayUnbndQueue<T> {
   public GlassQueue( ) {super( );}
   public GlassQueue(int origCap) {super(origCap);}
   public int size( ) {return numElements;}
   public T peekFront( ) {return queue[front];}
   public T peekRear( ) {return queue[rear];}}
```