SIENA*college*
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2020

Topic Notes: More Threads and Animation

Repaint Frequency

A few of you have noticed that we are sometimes creating a large number of threads in our programs. Fortunately, Java threads are a pretty lightweight constructs and we can reasonably create a large number. This is especially true for the kinds of threads we have used so far, where threads spend most of their time in a sleep call.

One thing those threads often do in our graphics programs with animation is to call `paint-Component`. If we have dozens or hundreds of threads, each calling `repaint` periodically, it could end up resulting in many more calls to `paintComponent` than are necessary for a smooth animation. Since we can only see 24 frames per second, there is no need to refresh the window at a rate much higher than that.

During class, we will modify the `SnowScene` program in two ways:

1. Threads responsible for individual animated objects (in this case, the `FallingSnow` objects) will update the positions of those objects, but not make a call to `repaint`.
2. An additional `Thread` object will be created that simply calls `repaint` about 30 times per second. Since we never need to interact with this object other than to call its `start` method, it can be constructed as an anonymous class.

Think about the circumstances where this version will result in fewer `paintComponent` calls and circumstances where it will result in more. Can we make any changes to limit the latter?

A Bouncing Ball Animation

In our next example

`https://github.com/SienaCSISAdvancedProgramming/BallTosser`

we see yet another animated ball object, with this one having a few new features:

- By dragging the mouse, a new ball can be launched on different trajectories and speeds.
- The ball bounces off the walls and floor, losing a bit of its energy each time it does so.
- The ball is subject to gravity.

- The ball disappears when its motion gets close to 0.

The launching of the ball is done similarly to how you launched the ball in Ski Ball. The differences in the x and y coordinates of the press point and the release point, subject to a scaling factor, are used to determine its initial speed in each direction.

The motion in y is subject to a gravitational pull, like in some earlier work.

When the ball hits any side of the window, it bounces off. We simulate it losing some energy on bounces by multiplying the speeds by a dampening factor any time a bounce occurs.

We decide the ball is done moving when it is on the "floor" and its motion in both dimensions is close to 0.

Another Look at Thread Safety

We have been working with animations for a while, and have only briefly talked about the dangers of concurrency. Those going on to take Operating Systems in the fall will study these issues in much more detail. Our concern here is to recognize when *thread safety* is a potential issue, and to learn how to ensure that we don't run into problems.

We want to look for situations where the same data might be used by two threads at the same time, and at least one of those threads might be modifying that data. We saw that can happen even with very simple data when we looked at the two threads that concurrently modify a `counter` variable with increment and decrement operations.

Along similar lines, consider an example of a singly-linked list data structure, where a reference to the first node in the list is stored in an instance variable `head`. Our program creates a list `x`:

```
SimpleLinkedList<Integer> x = new SimpleLinkedList<Integer>();
```

The program then creates two threads, each of which tries to add a value to this (so far) empty list:

Thread A:

```
x.add(1);
```

Thread B:

```
x.add(2);
```

We would expect that after these two statements execute, our list would contain two values, 1 and 2, in either order. Whichever thread executed its `add` method later would place its value at the start of the list.

Recall that the code for the `add` method of such a list has a case for adding at position 0, which is what the method calls above are trying to do:

```
    if (pos == 0) {
       head = new SimpleListNode<E>(obj, head);
       return;
   }
```

Suppose both `add` calls happen at almost exactly the same time. Both calls go into the `if` statement with the value of `head` being `null`. Both call the `SimpleListNode` constructor, with that `null` reference as the second parameter. Both constructor calls return a new node with a `null` value for their `next` reference, and both assign a reference to the just-constructed node to the `head` instance variable of the list.

What's in the list? We'd end up with just one node, and it would contain the value added by whichever thread assigned the `head` variable last. This is another example of a race condition.

We can avoid this kind of problem by declaring the methods of the list with the `synchronized` keyword, so at most one thread can be executing any method of the list class at any given time. This would work in this case to ensure thread safety. However, there are potential problems remaining, both related to the *granularity* of synchronization we are using.

- Placing the `synchronized` keyword on the method headers will serialize **all** access to the methods with that keyword, not just in situations where the instance variables of the list and its nodes are being modified. This reduces our concurrency and limits the ability of our threaded implementation to execute efficiently, especially when trying to take advantage of multiple processing cores. Further, the underlying implementation that ensures proper behavior of `synchronized` methods comes with computational costs. Every Java object has what is called an *intrinsic lock* that is used to support exclusive access for synchronization.
- This does not handle cases where the problem arises from unfortunate thread interactions that span multiple method calls. That is, the problem arises from the interleavings of the method calls themselves, not the code within a method.

Note: as has been mentioned a few times this semester, the first item above is the difference between `java.util.Vector` and `java.util.ArrayList`. All methods of `Vector` are `synchronized`, while `ArrayList`'s are not. So when doing single-threaded programming, we tend to use the `ArrayList` to avoid the overhead.

Java has a mechanism to take any class that implements the `List` interface and wrap it up so that all of the methods are `synchronized`. The `static` method `Collections.synchronizedList` is provided for this purpose. There are similar methods for other Java API collection interfaces including `Collection`, `Map`, and `Set`.

Ultimately, we want to make the `synchronized` code segments as small as possible so as not to limit concurrency, but large enough to ensure thread safety in the given situation.

The latter case from above can be demonstrated again with a linked list (though this would work just as well for something like an `ArrayList`).

We have a list with one element:

```
SimpleLinkedList<Integer> x = new SimpleLinkedList<Integer>();
x.add(1);
```

Then two threads each try to remove a value to make use of it in some way:

```
if (!x.isEmpty()) {
  int val = x.get(0);
  x.remove(0);
  // do something
}
```

What is the intended behavior here? What could go wrong? (in-class exercise)

This is the kind of problem that can occur in many of our animation examples, so we'll use one of those, `BallTosser`, to demonstrate a way to handle it.

What we need to do is to ensure *mutual exclusion* on the parts of the code that cannot safely execute concurrently. These parts of the code are called *critical sections*.

We introduce a new instance variable of type `Object`, which will be used as an *explicit lock*. We place the code in any method that forms our critical sections, for which we want to ensure mutual exclusion inside a block that looks like this, for an explicit lock named "`lock`":

```
synchronized (lock) {
   // critical section code
}
```

In the `BallTosser`, we create an explicit lock as an instance variable, then need to place two segments of code in the `synchronized` blocks. These are the two parts of the code that can modify our list: the loop in `paintComponent`, and the call to `add` in `mouseReleased`.

The modified version can be found in

`https://github.com/SienaCSISAdvancedProgramming/SynchronizedBallTosser`