Java Concurrency

Handouts comes to you by way of Julie Zelenski.

In Java, by default, execution usually proceeds from the main method in a standard linear, sequential fashion with just one thread of execution. However, Java was built from the ground up to be multi-thread-friendly. The language contains facilities that make it is quite easy to start up new threads to execute in parallel. For example, spawning a new thread for each object in order to model the actions of entities that execute and interact simultaneously is a pretty natural combination of the object-oriented and concurrent paradigms.

The `Thread` class

The `Thread` class in the `java.lang` package encapsulates a thread. The `Thread` object responds to many messages that allow you to control thread execution, change the scheduling priority, suspend/resume the thread, and more. The methods we are most interested in are:

```
Thread(String threadName)  // constructor for a Thread
void start()               // spawns thread, calls run method
void run()                 // intended to be overridden by subclasses
boolean isAlive()          // is the thread started and hasn't yet exited
void sleep(long numMillis) // sleeps current thread for at least millisecs
```

In order to use threads, you create a new `Thread` subclass and override the `run` method to include the code you would like the new `Thread` to execute.[1]

For example, here is a simple `Thread` subclass that is designed to loop, sleeping for a second and then printing out a message about the number of seconds that it has counted so far. It loops until it hits the target number of seconds specified when the `Timer` was created:

```
public class Timer extends Thread
{
   private int target;  // number of seconds to count up to

   public Timer(int countTo)
   {
      super("Timer thread");
      target = countTo;
   }
```

[1] There is also another way that doesn't involve subclassing Thread, but we're going to stick with the simple way in CS107. If you're curious, see the `Runnable` interface in the JavaSoft documentation.

```
    public void pause(int numMilliseconds)
    {
       try { sleep(numMilliseconds); }        // sleep for amount of time
       catch (InterruptedException e) {}      // ignore exception if it comes
    }

    public void run()
    {
       for (int i = 0; i < target; i++) {
          pause(1000);                              // wait 1 second
          System.out.println(i + " seconds have passed");
       }
    }
}
```

When you call new to create a `Thread` object, the new thread is created in a suspended state. In order to start it executing, you need to send it the `start` message, which sets up the thread stack and then calls the `run` method to do the real work. The thread exits when it finishes the body of the `run` method, or earlier if it receives a `stop` message.

Here is an excerpt of the code that would create a `Timer` thread and start it running:

```
{
   Timer timer;

   timer = new Timer(60);  // count 60 seconds
   // at this point, timer is created, but is not yet running...

   timer.start(); // sets up stack, calls run, now thread is active

   for (int i = 0;i < 1000; i++) // dumb loop, just for illustration
      System.out.println("Hi, I'm the main thread");
```

After you have called `start`, the thread can be scheduled concurrently with the main thread and any other currently active threads. From the above code, we would likely see lots of lots of "I'm the `main` thread" messages interspersed with the few and regular "N seconds have passed" as the timer thread wakes up to count and print.

Java threads will be subject to the same scheduling issues as we found in C: you have no guarantee who gets the processor and for how long, so lots of variation is possible. And most importantly, if the threads simultaneously access shared state, you need to work out control using the Java synchronization facilities.

Synchronization

Because the Java libraries were designed from the ground up to be thread-safe, most objects in the system packages already take precautions to avoid problems caused by simultaneous access. For example, the `println` message of the `PrintStream` class doesn't allow a print request to get interrupted midstream, something we might have to do manually for a non-safe version of `printf` (the latest Solaris version of `printf` is thread-safe, so we didn't have to deal with this, but on many other systems, it is not). In the

above program, even though we took no special precaution, the messages of the two threads will not be garbled in the output. Each message gets printed in its entirety before the next message because the `PrintStream` class has an internal lock that it is using to control access. Another thread-safe example is the `Vector` class, that ensures methods like `addElement` and `removeElement` are scheduled sequentially so that changing the contents of a Vector simultaneously from two different threads does not cause havoc. Having libraries already designed to be thread-safe is a big help toward writing a correct concurrent program.

The same facility used to protect system classes is also available for use in your own classes. Rather than control critical regions with a low-level semaphore primitive, Java provides a control primitive called a *monitor*, an idea from the Mesa language developed by Xerox. A monitor allows you to designate a critical region at a higher level and in a less error-prone way than raw semaphores. A monitor is basically a binary lock that allows you to serialize access to a resource.

In Java, you declare that a method should only be entered by one thread at a time by marking it with the `synchronized` qualifier. When one thread enters a synchronized method, Java guarantees that thread will finish and exit the method before another thread can execute any other synchronized method on the same object. For example, a `Stack` object that might be simultaneously accessed by multiple threads would want to protect the `pop` method so that only one thread can be in the middle of popping a value:

```
public synchronized Object pop()
{
   Object topValue = list.elementAt(0); // would be trouble if swapped out
   list.removeElementAt(0);             // between these two lines
   return topValue;
}
```

If a thread is currently executing a `pop`, any other requests to pop (for this particular Stack object) will be queued up outside the method until the first pop completes in its entirety. Note that a pop message sent to a different Stack object is not affected, since it is not operating on the same data.

Similarly, we might want to protect the `push` method:

```
public synchronized void push(Object o)
{
   list.addElement(o);
}
```

If we look up `addElement` in the `Vector` class, we will discover it is `synchronized` to avoid conflicts with the other methods that change the `Vector` contents, so the `Vector's` data is already protected. We still may want to `synchronize push` with other `Stack` methods, because we don't want a push interfering with a `pop`, for example.

You can check with the on-line reference to learn which methods in the standard system classes are `synchronized`. You'll find most of the system classes are already thread-safe, so in many cases, you may not need to take many additional precautions of your own.

Where locking granularity on the entire method is too coarse for your needs, you can also create a smaller critical region by applying the `synchronized` keyword to a piece of data in a smaller block:

```
public Object pop()
{
   Object topValue;

   synchronized (list) {   // acquire lock on the list object
      topValue = list.elementAt(0);
      list.removeElementAt(0);
   }
   ... other code here that can be outside critical section...
   return topValue;
}
```

Synchronized methods are like a single binary lock serializing object access. All synchronized methods in a class use the same lock, even if they are synchronized for different reasons. Synchronized monitors don't eliminate the possibility of deadlock or starvation, but they do at least help you to handle the ordinary cases of binary locking at a higher level with less ugly code and less chance for error than semaphores.

`notify` and `wait`

Although the binary lock for mutual exclusion is perhaps the most common type of synchronization needed in concurrent programs, it doesn't cover all the uses we had for semaphores. In order to achieve the "`rendezvous`" pattern we created with semaphores, Java provides the `notify` and `wait` methods.

When a thread needs to wait for an action taken by another thread, it uses the `wait` method (or the optional "`wait` with timeout" version) which efficiently suspends the thread, similar the `SemaphoreWait` function. When ready, another thread can wake it up by sending a `notify` message to wake up and resume a previously suspended thread, sort of like a `SemaphoreSignal` does. Both of these methods can only be called from within a `synchronized` block.