Worksheet 2: Object Oriented Multithreading

Updated: 31st July, 2019

1. Thread Safety

Each of the following classes is *not* thread safe. For each case, explain why.

(a)

```
public class Library
{
    private List<Book> bookList = new ArrayList<>();

    public void displayBooks()
    {
        for(Book b : bookList)
        {
            System.out.println(b.toString());
        }
    }

    public void addBook(String title, String author, String isbn)
    {
        bookList.add(new Book(title, author, isbn));
    }
}
```

(b)

```
public class Point
{
    private int x, y;
    private Object lock = new Object();

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void add(Point other)
    {
        synchronized(lock)
        {
            x += other.x;
            y += other.y;
        }
    }

    @Override
    public synchronized boolean equals(Object other)
    {
        return (other instanceof Point)
            && ((Point)other).x == x
            && ((Point)other).y == y;
    }
}
```

(c)

```
public class MyData
{
    private List<Double> data1 =
        Collections.synchronizedList(new ArrayList<>());

    private List<Double> data2 =
        Collections.synchronizedList(new ArrayList<>());

    public double dotProduct()
    {
        double result = 0.0;
        for(int i = 0; i < data1.size(); i++)
        {
            result += data1.get(i) * data2.get(i);
        }
        return result;
    }

    public addData(double d1, double d2)
    {
        data1.addAll(d1);
        data2.addAll(d2);
    }
}
```

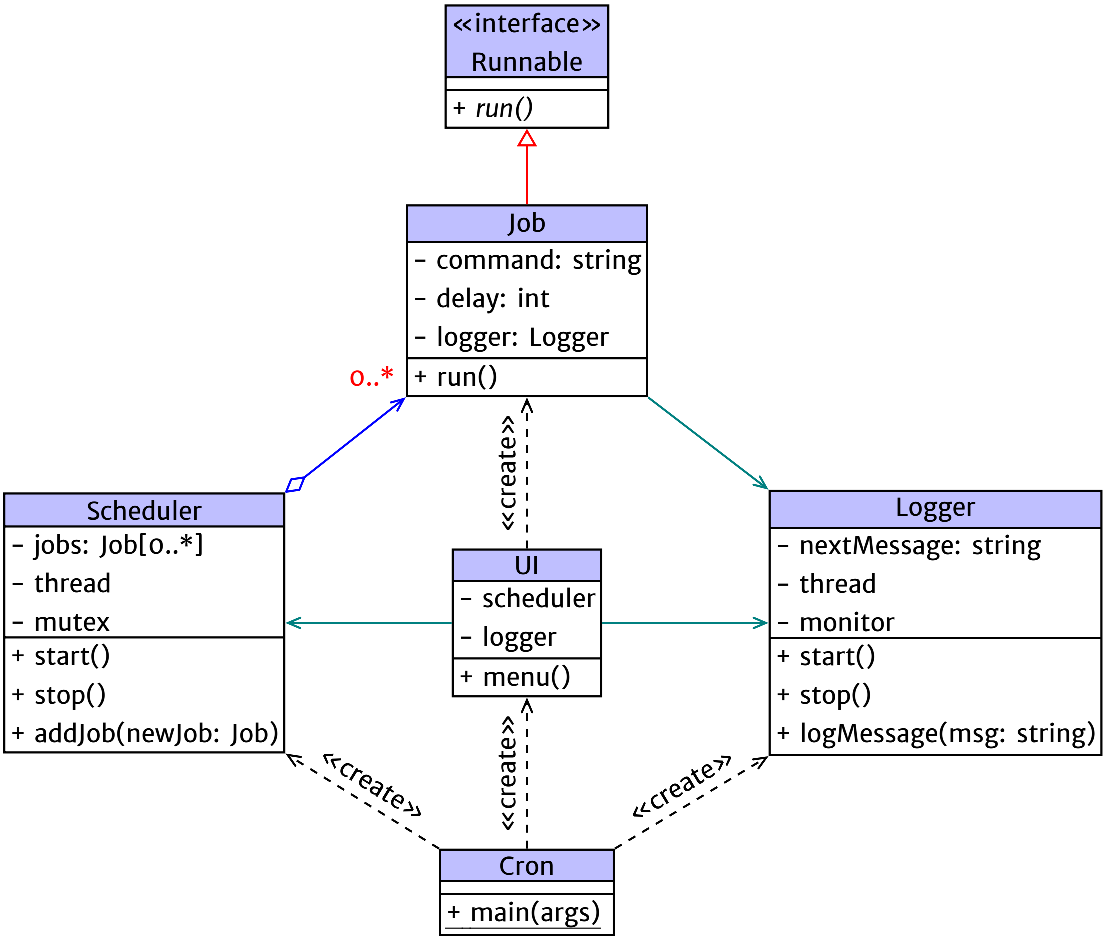
2. Multithreaded Cron

Cron is a UNIX utility for running particular commands ("jobs") periodically. Your task is to implement a highly simplified (and a bit contrived) version of cron in Java, using multithreading.

In reality, cron can schedule jobs in a highly customisable way, but generally not more often than once per minute. It works by waking up once per minute, checking what jobs need to be run right now (if any), starting them, and capturing their output.

Here, for simplicity, we'll say that jobs actually need to be run *every n seconds* (not minutes), where n is a positive integer that could be different for each job.

Here's a overview of the class structure we'll develop (where `Runnable` is the standard Java interface):



To get you started, find and download `cron-stub.zip` on Blackboard. The `Cron` and `UI` classes are almost complete, but we'll need to fill in all the code inside `Scheduler`, `Job` and `Logger`.

But let's look at the role of each of these classes:

`Cron` – the class containing `main()`. It creates instances of `UI`, `Scheduler` and `Logger`, and tells the scheduler and logger to start their threads (see below).

`UI` – the user interface. This asks the user (via the terminal) for details of new "jobs" to create and run, and adds them to `Scheduler` as requested.

`Job` – represents a particular command to be run at periodic intervals. Each command must be run in its own separate thread, in order to progressively capture its output. (Otherwise the IO operation to read the command output would prevent us doing anything else.)

`Scheduler` – controls the execution of jobs. The scheduler has a list of `Job`s, and its own thread (to be started then the program launches, and stopped when the user wants to quit).

`Logger` – receives the output of each completed job, and appends it to a log file "cron.log". All actual file writing is done inside a separate logger thread.

We'll attack this problem in several steps:

(a) The Job Class

First, we'll create the `Job` class, and launch it in a new thread.

(i) Create the class and fill in the details shown on the UML diagram.

(ii) What goes inside the `run()` method? The actual execution of external commands in Java (and capturing of their output) is not something we've covered, so here's the code to do that:

```
// Assume 'command' is a string containing the command the
// execute. Then we initially run it as follows:
Process proc = Runtime.getRuntime().exec(command);

// Arrange to capture the command's output, line by line.
StringBuilder output = new StringBuilder();
BufferedReader reader = new BufferedReader(
    new InputStreamReader(proc.getInputStream()));
String line = reader.readLine();
while(line != null)
{
    output.append(line);
    output.append('\n');
    line = reader.readLine();
}

// We've now reached the end of the command's output, which
// generally means the command has finished.
System.out.println(command + ": " + output.toString());
```

You will also need to handle `IOException` here, and (when we make a particular change later on) `InterruptedException`.

(iii) Modify the `UI` class to create a new `Job` object and launch it in a new thread.

See the source code comment "`// Add new job to the scheduler`" for the appropriate place to do this. This is the point at which the UI knows about all the things that go into making a `Job` object.

Warning: Do not call the `run()` method. The point of the exercise is to start up a new thread, and when you do this, the new thread will automatically call `run()`. However, calling `run()` directly yourself will not start a thread.

(iv) Compile, run and test your code. If you run your code on a Linux machine, and you enter the command "ls", you should see it displayed on the screen.

(b) The Scheduler Class

Now let's create the main cron functionality.

`Scheduler` itself *does not* implement `Runnable`. Instead, it must create a thread for itself. Also, within `Scheduler`, you'll need to create a lambda or anonymous class to define the task to be run on that thread. Inside that task, you'll create a loop that:

- Runs through the list of jobs, checks if any need to be started right now, and launches them in their own new threads.
- Waits one second. (Use `Thread.sleep(1000L)`.)
- Repeats until an `InterruptedException` occurs.

How does the scheduler know exactly when to start a job? It's a simple calculation: each `Job` object knows *how frequently* it should be run (its "delay"), and the `Scheduler` can add up all the seconds it's been waiting. Then, just check whether that seconds counter is divisible by "delay".

> **Note:** There is an alternate (and somewhat superior) approach to this, using the `Timer` class and the `scheduleAtFixedRate()` method. This will produce slightly more accurate timing than `sleep()`, because the latter doesn't take into account time spent by other things in the loop.

That's the concept. Here are the specific things you need to do:

(i) Fill in the `addJob()` method. This is just going to add a `Job` to an internal list. However, this will effectively pass information from the UI thread to the scheduler thread, so you need to lock a mutex.

(ii) Fill in the `start()` method, and define the scheduler task. This will implement the algorithm briefly described above. The task will also need to lock the mutex when it accesses the list of jobs.

(iii) Fill in the `stop()` method. This is called from the `UI` in order to shut down the `Scheduler`, so it needs to interrupt the scheduler thread. This means you need to store a reference to the `Thread` object when you create it. Also, did you handle the `InterruptedException` inside the task?

(iv) Modify the `UI` class to add `Job` objects to the `Scheduler`, and delete the code from before that directly starts them.

(v) Again, compile, run and test your code. Try adding a job that runs the command "echo Hello World" once every five seconds and another (simultaneously) that runs "ls" once per second. You'll find that the user interface gets disrupted from the command output, but it will still accept keyboard input (even if you have to type blindly).

(c) The Logger Class

`Logger` will take the command output away from the screen (so it doesn't disrupt the user interface), and append it to a file instead. Some aspects of `Logger` are very similar to Scheduler, specifically the idea of the start and stop methods.

The task still contains a loop, and still exits on InterruptedException. Obviously what it actually does is different. Here's how you can append text to a file:

```
try(PrintWriter writer =
    new PrintWriter(new FileWriter("cron.log", true)))
{                                       append (instead of overwriting)
    writer.println("Hello world");
}
```

However, where as Scheduler simply locks the job list when one thread is accessing it, Logger will use a *monitor* to coordinate its activities. The idea is as follows:

- Apart from the monitor, Logger will need another field to temporarily store a *single* log message (not a list).
- Logger.logMessage() will wait for the existing log message (if any) to be dealt with, set a new one, then notify the internal thread.
- The internal thread will run a loop that waits for a message to be provided, then appends it to a file, then notifies any other thread waiting in logMessage().

To finish it off, you just need to have each Job make a call to Logger.logMessage() instead of System.out.println().

End of Worksheet