Worksheet 3: Multithreading Design Structures

Updated: 9th August, 2019

1. The GUI Thread

What is the following code (probably) trying to do, and why is it problematic?

```
public void displayImage(Future<Image> imageFuture, ImageDisplayer display)
{
    Platform.runLater(new Runnable()
    {
        @Override
        public void run()
        {
            try
            {
                display.setImage(imageFuture.get());
            }
            catch(InterruptedException | ExecutionException e)
            {
                display.setImage(DEFAULT_IMAGE);
            }
        }
    });
}
```

How would you fix this, without altering the method signature?

(Assume that `ImageDisplayer.setImage()` does exactly what it sounds like, and simply places an image on the screen.)

2. Understanding Blocking Queues

Say you would like to develop an artificial-intelligence technical support application, to run on a web server. You want users, potentially hundreds at a time, to write queries to it over the web, and have the AI answer them individually (as best it can).

(a) In the first stage, we still need humans to help train the AI. It must monitor conversations between users and human technical support personnel, so it can learn what it should (eventually) do.

You will have the following key classes:

`UserSession`. This will have many instances, each one responsible for a particular user (taking their query and displaying responses).

`HelperSession`. Similar, but for the technical support person. At any given time, each `UserSession` object will be associated with zero-or-one `HelperSession` object(s), and the two will exchange `String` messages.

`AITrainer`. This will have a *single* instance (across the whole application) which must receive all the messages exchanged between users and helpers. It processes and stores them.

Objects of all three classes are each responsible for their own thread. That is, each user and helper has a thread, and the AI trainer does too. Given that we'd like to use blocking queues to manage all communication between these threads, answer the following:

- Where *could* we put blocking queues?
- What should go inside them?
- What *kind* of queues should they be?

Some further considerations:

- `AITrainer` needs to receive whole conversations at once, OR it needs to know which conversation each message belongs to. Otherwise it won't be able to see what's going on.
- The single `AITrainer` object should not cause the conversation between user and helper to block, or they will get annoyed!

(b) Once the AI has been sufficiently trained, we introduce a different design in which it will take the place of human helpers.

Now we'll have the classes `UserSession` and `AIHelper`. The first (as before) has one instance and one thread per user. However, a single `AIHelper` instance will be able to help any number of users, so we don't need more than one of it.

What blocking queue arrangement(s) could we have now?

3. Understanding Thread Pools

Continuing the AI scenario from the previous question, let's consider how we could use thread pools to improve the application's efficiency.

Specifically:

(a) Can we improve the efficiency of `AIHelper`?

(b) Can we improve the efficiency of `UserSession`?

4. File Search

Obtain a copy of either file_searcher_javafx.zip or file_searcher_swing.zip. These contains a small, working Java GUI application that searches for files by name. That is, it takes a path and search term from the user and reporting a list of all matching files. This application is currently single-threaded, and so the GUI will freeze during the search process.

It doesn't matter whether you choose the JavaFX or the Swing version. If you can successfully run one version, unmodified, then use that one. They will both give you the same learning experience. Also note that you're *not* being asked to actually develop the GUI – that's already been done. In either case, you can run the `gradlew` script to build and run the application:

```
[user@pc]$ ./gradlew run
```

Note: Swing and JavaFX have subtle differences. In particular, they have different names for the class/method that lets you schedule a task on the GUI thread:

- In JavaFX, it's `Platform.runLater(...)`.
- In Swing, it's `SwingUtilities.invokeLater(...)`.

They both do the same thing, but you have to use the right one for the GUI library you're using.

Your task[1] is to introduce *two* new threads that will communicate via a blocking queue: one thread to find *all* files, and another to filter them by name. The finder thread should *enqueue* (put into the queue) each file it encounters. The filter thread should *dequeue* each one, check if it matches the search term, and if so, add it to the result list.

However, both operations should be done *in the same class* (to avoid otherwise messy class coupling). It is recommended that you create a new "`FSFilter`" class for this purpose.

How will you know when it works? If you execute a search that takes some time, and you can see the results *incrementally* coming in, then you know the GUI is being updated while the search is progressing; i.e., it hasn't frozen.

5. File Content Search

Modify the search application so that it looks for matches in file *contents*, rather than filenames. To to do this[2], have the filter thread work with an `ExecutorService` object and perform each individual file check in its own separate thread.

Implementation notes:

- You'll probably find that you won't need your blocking queue anymore, and in fact your final code here (apart from the file reading) may turn out simpler than in the previous question.
- Use `Executors.newFixedThreadPool()` to obtain an `ExecutorService`.
- How many threads should your thread pool have? This is an open question (since searching file contents involves both I/O and CPU processing). For the purposes of this exercise, you can just hard-code a number. However, if you'd like, you can try to identify the optimal number of threads by experimentation (on your current machine at least).

[1] There are certainly other design options to achieve the same effect, but we need an excuse to try out blocking queues.

[2] Again, there are of course alternative design options, but we need an excuse to use executors.

- `ExecutorService.submit()` returns a `Future`, but you can ignore it and simply have each filter task communicate its own result to the window.
- `IOException` may happen when trying to read a file. If it happens in this case, treat it as if the file simply didn't match the search term.

6. (Optional Extra) Modify Cron to use a Blocking Queue and an Executor

In the previous worksheet, you used lower-level threading mechanisms to execute cron jobs and log their results. In practice, we could replace these by higher-level mechanisms. Specifically, you could:

(a) Have `Logger` use a blocking queue to accept log messages, instead of a monitor.

(b) Have `Scheduler` use an executor / thread pool to run its jobs, instead of directly launching a new thread.

(c) Have `Scheduler` use a `CompletableFuture` to run its jobs, and take responsibility for passing each job's output to `Logger`. We would need to change `Job` to implement `Supplier<String>` instead of `Runnable`.

These changes should actually result in simpler code!

End of Worksheet