CS 251 - Autumn 2021 Outlab 7 - Concurrency in Java

Scotland Yard! (100 points)

Scotland Yard is a very popular board game, the map being the city of London. The London Metropolitan Police Force is called Scotland Yard for real, too.

In this exercise, we're going to build a version of this game! All files are part of a package called `bobby` (that's slang for a London policeman). You can compile them with

```
javac bobby/*.java
```

and run the main method of a file `Foo.java` with

```
java bobby.Foo <args>
```

To run the server:

```
java bobby.ScotlandYard <port1> <port2> …
```

To run a client:

```
java bobby.Player <port>
```

Logistics

You have to fork this **repo**, which contains a partial implementation of the tool we want to build. Read the code, understand what's required, and fill in the details, following the guidelines. The code you have to write is short and reasonably accessible, but to get it right, **you have to know what you're doing.**

Feel free to star the repo, and follow guitarhero22 when you enjoy the outlab xD

You must not edit `Board.java, ManualPlayer.java` and `RandomPlayer.java`; it is sufficient to complete the sync logic in `ScotlandYard.java, Moderator.java, ServerThread.java` to get this tool running.

A lot of specific questions about the requirements will be answered by looking at the code, so in this document, we shall focus on the high level details, and the required background.

Submission Details:
You have to turn your code in on Moodle by **Saturday, October 16, 23:59**

For your submission, first make a directory called `<rollno1>-<rollno2>` where rollno1 is the lexicographically smaller roll number. Eg. `180070026-180070035`

The following should be the structure of your directory. Check with `tree -a`

```
.
|-- bobby
|   |-- Moderator.java
|   |-- ScotlandYard.java
|   |-- ServerThread.java
`-- references.txt
```

Compress this `<rollno1>-<rollno2>` directory, strictly with the command

```
tar -czvf.
```

For example,

```
tar -czvf 180070026-180070035.tar.gz 180070026-
180070035/
```

Grading Details
We will query your server with several RandomPlayers running in parallel. These will print the feedback they get, forming a trace. The score you earn will depend on how coherent your trace is (the exact grading scheme will be fairly reasonable and lenient). If something goes wrong, you can show us your code during the Project viva on 20th, and we can negotiate the awarded credit.

Please don't edit the print statements.

Motivation

Java is a very popular language in industry. You probably associate it with OOP (the very mention of it might be banal at this point). You'd also know that it's a game of pointers. While C/C++ requires you to be *meticulous* with pointers lest you allow a memory leak, Java lets you mess around with impunity, because something quite remarkable is happening under the hood: **garbage collection**. That's simply the process of freeing up memory that no longer has any reference to it.

Meaning, if you set the last pointer that points to an address to null, that content is gone. You don't have to bother, JVM has got your back. If only one could solve real world problems by denying their existence.

As for this Outlab, we shall learn some standard concurrency ideas with a neat synchronization primitive. We shall also acquaint ourselves a little with inter-process communication and socket programming, learning some exception handling along the way. **All very marketable knowledge**.

We encourage you to look through the given code: although you could earn full credit just fine without a full perusal, it has Easter Eggs you might find edifying.

No one:
Absolutely no one:
Not a single soul*:*
Mihir: GRE words go brrr

The Game

The city is an 8 x 8 grid, a Fugitive runs amok. In a single turn, the Fugitive can move like a chess Queen: horizontally, vertically, or diagonally. Upto 5 Detectives attempt to catch the Fugitive. Detectives move like chess Rooks: horizontally, or vertically. In this game, Players don't block each others' paths.

The Detectives win if one of them lands on the same square as the Fugitive, or if the Fugitive quits the game.

The Detectives lose if none of them can catch the Fugitive within 25 timesteps.

The Fugitive can see everyone, but is mostly invisible to the Detectives: the Fugitive only surfaces when 3 mod 5 timesteps have passed, i.e. at timesteps, 3, 8, … , 23, and finally 25, when the game is over.

But what is a timestep, and how do we simulate all this?

The Simulation

We assume that each agent decides their moves independently. Detectives have incomplete access to the information, the Fugitive has complete access at all times.

If this were played in real life, we'd do the following, akin to Mafia (except here we're interested in colliding with the Fugitive on a grid): the Detectives and the Fugitive would assemble. God would give them a starting square, and the game is played in rounds. In each round, the players submit their moves individually to God. Once God receives a response from all the players, he makes announcements: he tells the Fugitive where everyone is, and he tells the Detectives where they all are (only sometimes the Detectives are told the Fugitive's location).

Note that God's announcement is the only time someone gets information about other players.

If the announcement means that the game isn't over yet, the next round can begin only when everyone has received feedback for a round.

During a round, players can either make a move, or decide to quit. If the Fugitive quits, the Detectives in the round are told that they have won the game.

Detectives can freely quit and rejoin the game. New players, however, can only be registered *between* rounds, you can't just barge in. In real life too, you tell people to "play from the next round", don't you?

Note that when Detectives join, they must check if the game is going on: only then can they play, otherwise they must decide to quit.

A **timestep** (the number you see with Move in the output) corresponds to a round in which either:

1) At least one Detective entered and the Fugitive made a move on the board, or
2) The Fugitive quit

The Client-Server Model

How do we make computer programs play Scotland Yard? We've given you `ManualPlayer.java` and `RandomPlayer.java`. A process running the main method of either of these files is what we call a **Client**. The Client establishes a **socket connection** to the game **Server** (that's the process running the main method of `ScotlandYard.java`) that listens on a **port.**

Look at the client code. It basically listens for incoming input from the connection with a `BufferedReader`, and then decides on a move (either gets it from a human typing on the terminal, or generates it randomly), and sends that move across to the server with the `PrintWriter` associated with the socket connection.

The Server, on the other hand, caters to several clients, who are playing a single game. The Oracle Java Documentation is a good place to start learning about socket servers. For multiple clients, the way to go is to spawn threads, one for serving each client, and then continue listening again.

Our server is organised as follows: we have a few ports that we can specify on the command line. For each port, the server loops: spawning a `ScotlandYardGame` and letting it run to completion. It is the `ScotlandYardGame` that listens on a port for client connections.

For each client that connects via a socket, we create a `ServerThread` to talk to it. This `ServerThread` facilitates round by round play. `ServerThreads` playing the same game have shared access to the same `Board`, and need to access data consistently and run in synchrony.

Finally, we also have the `Moderator`, whose primary job is to allow `ServerThread`s into the next round.

Your job is to get this synchronization right, by completing `ScotlandYard.java, Moderator.java, ServerThread.java`

Processes and Threads

The computer runs several processes concurrently. However, they all use a common resource: time on the CPU. The **process scheduler** is a part of the Operating System that allocates CPU time to processes.

A process, at the lowest level, is a sequence of instructions loaded onto memory that the CPU hardware can execute. A process can have multiple **threads** that can execute concurrently**.** As far the scheduler is concerned, the **thread** is the smallest unit that contends for CPU time.

Look up the terms **Running** (the thread has been scheduled and its instructions are being executed), **Ready** (the thread is available to be scheduled) and **Blocked** (the thread is not ready to be scheduled). A thread can block for several reasons: it is waiting for some data to be fetched from disk (disk operations are abysmally slow with respect to memory), or it is waiting for some input from a socket buffer. In this lab, we will *block* threads on something known as a **“Semaphore permit”**. More on that later.

Blocked threads can be **signalled,** when whatever operation they were blocking on is completed: data from disk is ready, buffer is non-empty, etc. When this happens, they go to the **Ready** state.

Synchronization

We write concurrent programs that are agnostic to the Scheduling Policy used. You don’t want your results to be at the mercy of something as fickle as the *scheduler*. You want your multi-threaded program to be **correct**; you should thus somehow account for *every* possible scheduling of threads. Why, and how? Read on!

Threads of the same process share the same **program stack:** in English, this means that they can access common memory. They can write to and read from the same variables, and when that happens, you want your results to be consistent: we say that this is protecting the **integrity of the data.**

Code that accesses shared variables is known as a **critical section,** and for the correctness of our implementation, we want to ensure that only one thread is executing a critical section.

Why? Consider something as simple as *incrementing a shared variable.* In order to execute `x = x + 1`, you need to fetch `x` from memory into a register, perform the addition, and then write the result back to memory. Suppose threads A and B can access this section together. A possible scheduling is: A fetches 0, B fetches 0, A writes 1, B writes 1. If `x` was 0 initially, it **does not** end up being 2, as expected

Thus, at most one thread should be guaranteed to access `x` at a time.

We call this **mutual exclusion**. How mutual exclusion is *ensured* through hardware is a discussion best conducted in an OS course.

For coding purposes, we protect critical sections with synchronization primitives like **locks** (also known as **mutexes**, singular mutex, the most popular kind of lock), **conditional variables** and/or **semaphores**.

Java, as a programming language offers the somewhat magical `synchronized` keyword, that gives some guarantees, but from the documentation, we see that this control is way too coarse. For more fine grained control, we use synchronization primitives.

Using synchronization primitives doesn't magically make your code correct: they rely on the programmer's logic and discipline to work.

The humble lock is the simplest primitive that can protect a critical section. The code would look like

```
lock.acquire();
// do whatever is critical
lock.release();
```

You must be disciplined as a programmer! Use locks wherever there is a critical section. A thread can proceed only if it "acquires" the lock.

Just locks aren't very flexible, we can make program sync logic richer with conditional variables.

For the simplest case, assume a process has 2 threads, **A** and **B**. **A** multiplies the input by **x**, and **B** adds **x** to the input. You want your program to output **y*x + x**, where y is the input. If the scheduler schedules

B first and **A** later, then the output will be **(y+x)*x**, which is not what you intended. So, you need to somehow tell thread **B** to *wait* for **A** to finish its job, and that's where Conditional Variables are handy.

Locks make you wait, conditional variables tell you why.

Semaphores

Semaphores are a very neat synchronization primitive, and can be proven to be as powerful as the combination of locks and conditional variables. In this document, we will briefly describe how Semaphores work; you can read more, and also look up Conditional Variables if you're interested.

The Oracle Java Documentation is an excellent place to get an overview of what you can do with Semaphores in Java. To quote the explanation, a Semaphore is conceptually a count of **permits.**

Intuitively, when a thread wants to access a critical section, it will call **acquire** on the semaphore, and ask for a permit. If the count is greater than 0, the semaphore "grants the permit", i.e. decrements the count, and the thread can proceed.

If the count is 0, then it means there are no permits, and the thread **blocks**. Clearly, there has to be a mechanism to **release** permits.

Another thread, having done something to make the critical section safe to access, can issue a **release** on the semaphore. What this does is increments the count, and marks a thread that was blocked on the acquire as **ready**.

It is easy to see that if you initialise a Semaphore with a count of 1, you can use it just like a lock. However, there are several more integers than just 0 and 1, and it's not hard to imagine that a semaphore can do much more than locks.

Barriers, and the synchronization we need

Having discussed the technical preliminaries, we shift the discussion back to the problem at hand. We said that the game proceeds, round by round. Here's what a round looks like:

1) `ServerThread` gets its move from the client, reading a single line from the socket connection buffer.
2) `ServerThread` is allowed to play, and makes the move on the board
3) `ServerThread` then waits for all other active `ServerThreads` to make their move

4) `ServerThread` gets the feedback for the round, and relays it to the client via the socket buffer.
5) `ServerThread` then waits for all other active `ServerThreads` before proceeding to the next round.

This scenario, where several threads have to wait for each other before proceeding to the next step, is typical: we call the solution a barrier. Java does have an inbuilt barrier, but that only works for a fixed number of threads, once initialised.

In our game, the number of threads each round can vary. A player might quit, either gracefully by pressing "Q", or rage quit with Control+C. We need to deal with both gracefully. There can also be players waiting to join, and need to be inducted into the game as soon as possible.

It is the `Moderator`'s job to regulate all this between rounds, and set this flexible barrier up for the round. Have a look at the code and comments to see how this can be done. The idea is, the number of threads that we expect to trigger the barrier is `playingThreads.`

Points to note

A subtlety that needs to be handled is, what if the game is not being played (`boolean dead`)? Is it game over, or has it not been set up with a Fugitive yet? (`boolean embryo`)

It is `ScotlandYardGame`'s job to create `ServerThreads` and assign them ID's (we assume that the first thread to join is The Fugitive for the game, thereafter, only Detectives can join or leave without ending the game).

At the heart of all this logic is the `Board`. Every `Runnable` that is created has a pointer to the same `Board`, the stuff in the `Board` is shared. It is essential for you to go through `Board.java,` in particular, the variables and Semaphores we've defined to help with synchronization, and the data structures and methods we've defined to handle new players joining and leaving.

Do remember to acquire the appropriate permit before accessing anything!

So go ahead, code! The end result we expect is described in the next section. If there are any doubts, feel free to ask us on Piazza **by 14th**

October, and if there are doubts after that make reasonable assumptions and proceed.

Properties of Trace(s)

We plan to evaluate your output by querying your server with multiple RandomPlayers, run in parallel, for example:

```
java bobby.RandomPlayer 5000 > tr1.txt & java
bobby.RandomPlayer 5000 > tr2.txt & java
bobby.RandomPlayer 5000 > tr3.txt & java
bobby.RandomPlayer 5000 > tr4.txt & java
bobby.RandomPlayer 5000 > tr5.txt & java
bobby.RandomPlayer 5000 > tr6.txt
```

The printing is already implemented for you, so this output purely depends on how you synchronize the threads.

We will then look at the trace files, which must have the following properties (Move in the feedback denotes how many timesteps have elapsed):

1) The first line will identify what role the character is playing, and in what game. Games are uniquely identified by a pair (port:gamenumber), eg. 5000:0.
2) A Fugitive Trace is unique for a particular game.
3) In a Detective trace, moves must be in serial order, and no number is repeated. Eg, 3, 4, …, 19. The first move in a Detective trace is at least 1.
4) In a Fugitive trace, Move numbers start from 0 or 1, and are monotonically increasing, and the increment is at most 1.
5) In a Fugitive trace, if the Move number is 0 or the same as the previous line, it must be the case that all detectives are on -1 in that line
6) For every game G, Player P and Move M, there is at most one trace where P plays move M in Game G. This means that two clients cannot get the same character in the same game at the same time (character is Fugitive, or Detective i, i=0,1,2,3,4)
7) In Game G, if the last line is move M for the Fugitive trace, then:
    a) If M reports Play, then the highest move that can be found for any trace corresponding to Game G is M+1. This must report Victory.
    b) If M reports a result, then the highest move that can be found for any trace corresponding to Game G is M. This must report the opposite result for the Detectives

    c) These are the only occurrences of a result being reported
8) In every game G, all traces that include feedback for Move M **must** agree on the positions of all Players they report.
9) The sequence of squares observed in the feedback must correspond to valid moves.

We will use autograder, but also make it stronger to catch vacuous ways that may exist to bypass it. We could also scrutinise your code manually in the Project viva slot.

Python autograder provided to you along with repo. We believe this is correct, but like most disappointments, it is the product of a long night, so we can't guarantee it's 100% correct. At the end of the day, we're just a couple of older kids trying to help.

Reporting bugs in autograder is nice. There is a known bug that sometimes surfaces in `RandomPlayer.java` (the random number generator perhaps is sometimes called incorrectly). Feel free to let us know the fix on Piazza, if we don't get around to it.

In context of the updated hint, ServerThread, parts 6B and 6C:

`Part 6B`

~~The last thread to hit this must issue one~~
~~permit for the moderator to run~~

`Part 6C`

The last thread to hit this must issue one
permit for the moderator to run after (erasing player if necessary)