Lab 1 - Basic Concurrency in Java

September 9, 2023

- Group 3
- de Sotto, Diego and Zapata, Francisco

1 Task 1 : Creating and joining threads

Source files:

Case A `task1/MainA.java` (Spawns a thread that prints "Hello world").

Case B `task1/MainB.java` (Spawns five threads that prints "Hello world". Prints "Goodbye" once all are done).

Case C `task1/MainC.java` (Same as MainB, but identifying each thread with an id).

To compile and execute (Inside the task1 folder in the terminal):

Case A
```
javac -d .  MainA.java
java task1.MainA
```

Case B
```
javac -d .  MainB.java
java task1.MainB
```

Case C
```
javac -d .  MainC.java
java task1.MainC
```

2 Task 2 : Simple Synchronization

2.1 Race conditions

Source files:

- `task2/MainA.java` (Spawns n, arbitrary integer of threads that increase an integer by 1 000 000) Expectation is for the value of the integer to be different than 1 000 000 (as memory read / writes might overlap like seen in lectures).

To compile and execute (Inside the task2 folder in the terminal):

```
javac -d .  MainA.java
java task2.MainA
```

2.2 Synchronized keyword

Source files:

- `task2/MainB.java` (Same objective as 2a, but using synchronized keyword) Now, the expectations are different. By using the synchronized keyword, threads are blocked from modifying the value of the integer simultaneously. Hence, the value should be 1 000 000 * n

To compile and execute (Inside the task2 folder in the terminal):

```
javac -d .  MainB.java
java task2.MainB
```

2.3 Synchronization performance

Source files:

- `task2/MainC.java` (Record average execution times for 2b with different values of n) Results were then plotted in a graph to be compared with results from DARDEL. Data is stored in two different *.dat files, depending on where the code was executed. A isPDC boolean flag is required when compiled code is executed to distinguish.

To compile and execute (Inside the task2 folder in the terminal):

```
javac -d .  MainC.java
```

`java task2.MainC <isPDC>`, where isPDC should be true if executed in PDC, false if it is done in a local machine.

3 Task 3 : Guarded blocks using wait()/notify()

Source files:

Case A `task3/MainA.java` (Implementation of asynchronous sender-receiver). Many values were printed, most around the halfway point of 500 000. However, not a single time was 0 or 1 000 000 printed, which suggests no sequentiality.

Case B `task3/MainB.java` (Implementation of busy-waiting receiver). Here, the problem was solved and the program consistently printed out the correct answer of 1 000 000.

Case C `task3/MainC.java` (Implementation of a waiting with guarded block).Again, the problem was solved and the program consistently printed out the correct answer of 1 000 000. Issue in first version was fixed by changing the location of the synchronized blocks and reintroducing the done flag. Now, the wait() method is kept inside the loop waiting for the done flag to be set to true.

Case D `task3/MainD.java` (Explore the effects of guarded block on performance). While exploring the performance, we faced the issue of thread executions overlapping. Solved when changing the code of MainC to solve the errors there

To compile and execute (Inside the task3 folder in the terminal):

Case A
```
javac -d .  MainA.java
java task3.MainA
```

Case B
```
javac -d .  MainB.java
java task3.MainB
```

Case C
```
javac -d .  MainC.java
java task3.MainC
```

Case D
```
javac -d .  MainD.java
java task3.MainD
```

4 Task 4 : Producer-Consumer Buffer using Condition Variables

Source files:

Case A `task4/Buffer.java` (Implementation of a producer-consumer buffer class with limited capacity N. A FIFO protocol was used for handling data.) We decided to implement it using a ReentrantLock and Conditions, as the resultant code was perceived as cleaner and more intuitively related to the "queue" idea we had of threads waiting to access a specific piece of code. In the second submission, an if condition looking for a case in which the buffer is closed and empty and a call to remove an integer is made. A ClosedException is now thrown in that scenario.

Case B `task4/ClosedException.java` (Exception thrown if an attempt to close the Buffer is made when it is already closed).

Case C `task4/Main.java` (Testing of the buffer class of 4a). Takes the storage space of the buffer as an argument and spawns two threads that add / remove integers to it.

To compile and execute (Inside the task4 folder in the terminal):

```
javac -d .  *.java
javac -d .  Main.java
java task4.Main <N>
```
where N is the number of spaces Buffer is started with.

5 Task 5 : Counting Semaphore

Source files:

Case A `task5/CountingSemaphore.java` (Implementation of counting semaphore) Once again, we decided to go with the ReentrantLock/Condition pair as it appeared more intuitive.

Case B `task5/Main.java` (Tests for CountingSemaphore class) !!!! WHILE LOOP USED TO TEST !!! The program spawns 10 threads that constantly compete for the resources (default of 5 available) with random waiting times. Program

To compile and execute (Inside the task5 folder in the terminal):

```
javac -d .  *.java
java task5.Main
```

6 Task 6 : Dining Philosophers

6.1 Modelling

Source files:

- `task6/MainA.java` It was modelled implementing a ForkSemaphore class, which is a subclass of the CountingSemaphore in task5. As expected, the program deadlocks after a few eating breaks.

To compile and execute (Inside the task6 folder in the terminal):

```
javac -d .  *.java
java MainA
```

6.2 Solution

Source files:

- `task6/MainB.java` To solve the deadlock, the last philosopher switches the order in which he picks up the forks. This time, a synchronized block approach was followed.

To compile and execute (Inside the task6 folder in the terminal):

```
javac -d .  *.java
java MainC
```

7 Task 7 : Deadlock Debugging

Source files:

- task6/Main.java (Creates a deadlock of two fighters trying to fight with two different swords) For deadlock debugging, some IDE's offer a "Detect deadlock" option - e.g Eclipse with the Sun JVM). There are also methods of the Thread class that allow to check for mutual exclusion. In this lab, I used the VSCode debugger to visualize my code execution, but literature siggests using a profiler (like JProfiler), which connects to the JVM and works through your code in a low lever way.