**CA4006**

**Concurrent and Distributed Programming**

**Assessment N°1:**

Martin FERRAND - 22104615

Clovis GILLES- 22103376

1. **what aspects of the project you got working / not working / partly working:**

Everything listed below is working:

* A delivery of 10 books is made with a probability of 0.01 every tick (configurable).
* Bookshelves have a maximum capacity (configurable, if set to 0 then unlimited).
* 10 assistants by defaults (configurable, minimum 1).
* 6 sections by default (configurable, minimum 1, possibility to generate sections).
* Assistants prioritize empty sections when delivering books.
* Real time data reporting.
* Assistants take breaks. By default an assistant can start taking a break after 200 ticks and must take a break after 300 ticks. Each tick when a break is possible the assistant has a 0.01% chance of taking a break. Configurable.
* A lot of parameters are configurable. Launch the program with ‘-h’ option when launching to learn about all of them.
* The mapping between ticks and time is configurable. By default a tick last 10 milliseconds.
* Primitive data printing in the terminal.

1. **How the work was divided**

Most of the work was done side by side. We can however say that some part were mostly done by one or the other as seen below:

Martin FERRAND:

* TimeScheduler.java
* SynchronizedThread.java
* Main.java
* StatsManager.java

Clovis GILLES:

* Assistant.java
* Bookstore.java
* Book.java
* Section.java
* Logger.java
* Makefile

1. **How to compile and run the code:**

To compile the program with make:

**Make**

A build directory should have been created.

To run the program with make:

**make run**

The program requires at least one argument, to add arguments do:

**make run arg= ' -d '**

For exemple if you want to use 100 assistants, set the tick length to 1 millisecond and set the spawn rate of client to 1 every 5 ticks on average you would write:

**make run arg='-an 100 -tv 1 -csr 5'**

If make is not an option you can use javac:

**javac -d ./build ./\*.java ./Entities/\*.java ./Objects/\*.java ./Tools/\*.java**

A build directory should have been created.

To run the program with java:

**java –c ./build Main**

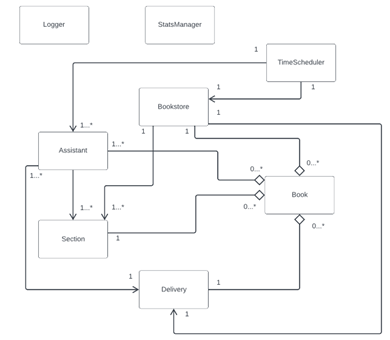
Or with argument, this time just add them at the end without single quotes:

**java -cp ./build Main –d**

The default argument is ‘-d’, the program will use the default values as described by the assignment.

To stop the program, enter ‘quit’ or use **CTRL + C**.

1. **What are the considered tasks/dependencies in your program**



* The purpose of the **Timescheduler** class is to manage ticks, the *Assistants* and *Bookstore* classes and their synchronization. It exists only once in the program and has a list of *SynchronizedThread* in which Assistants are stored whose number is between 1 and a predefined number and an instance of Bookstore which is unique in the program.
* The purpose of **Bookstore** is to manage book deliveries in the *Delivery* class as well as to create and assign Customers to sections. She also creates *Books* and assigns them a section. It has a *Delivery* class and it can create a predefined number of books in *Delivery*. It can be found in the *SynchronizedThread* list of the *TimeScheduler* Class as a single copy.
* The **Delivery** class represents the delivery arrival area of *Books*. It has a list of Books whose are waiting to be delivered. This list is filled by itself when the *Bookstore* ask it to. You can find this class in the program’s main, then it is referenced in *Bookstore* and in the *Assistant* class.
* The **Book** class represents the library’s books. We can find its instances in *Section* and in *Assistant* and in *Delivery*. They are created by the class Delivery.
* The **Section** class aims to represent the bookshelves. It has a name, a list of *Book* of 0 to a number predefined. The Sections are stored in a list in the main but we can find a reference to this list in the Assistant class and in the Bookstore class.
* The **Assistant** class is the representation of a human assistant in a bookstore. It will take the books from the delivery area, carry them to their sections, put them in it, and it will take breaks. It has a list of Books. A reference to the list of the *Section*s and another of the *Delivery*. The *Assistant*s are created and stored in the *SynchronizedThread* list of *TimeScheduler*.
* The **Logger** class aims to manage all the operations that are related to the logging of the information. It is a static class that can be called everywhere in the code without being instanced.
* The **StatsManager** class aims to manage all the operations that are related to the statistic’s calculations and the display of them. It is a Singleton. It also manage the simple interface.
* The **Main** class is not represented on the previous graph. It aims to initialise the program and to parse the arguments of the latter.

1. **what patterns/strategies did you use to manage concurrency**

Nearly all objects inherit of *SynchronizedThread* with work in tandem with *TimeScheduler* to provide synchronisation between every component. The objective is to make sure that the tick count is consistent between every thread and to make sure that the threads are well handled. This mean every thread have been started (even if new threads are added later on while the program is running), that no thread runs much faster than any other by waiting for all to finish what they were doing (we are aware that we are wasting CPU time of faster thread in this configuration). We are using *java.util.concurrent.CyclicBarrier* to achieve the synchronisation.

*SynchronizedThread* is used to create any objects that need to run in a thread that needs to be synchronized (to get correct tick count). Therefore, it is an abstract class that inherit from Thread and implementing the run function of Thread that must not be overridden by child class. The function doWork() is abstract and therefore must be implemented by child class, the function must do all the work the object must perform during a single tick. The rest of the methods are utilities to setup the object, exit the thread and other miscellaneous functions.

The rest of concurrency issues are usually dealt with by using the "synchronized" keyword with provide an easy-to-use monitor when common resource is modified. It is not used everywhere because sometimes it doesn’t matter if the resource is modified by multiple thread at the same time.

1. **How your solution addresses issues like fairness, prevention of starvation etc.**

To address all issues such as fairness, prevention of starvation, we decided to use the key word "synchronised" that acts as a monitor, on the functions of the Delivery as well as those of the Section object that allow to take and drop a book. This is function are very short and can not do any blocking which allows that no thread remains blocked in the critical area which is protected by synchronised and therefore all the threads wanting it can access this critical area provided to wait that the latter is liberated, what happens in a short time.

However, the "synchronised" keyword is only used five times, two times in functions to take books in sections or delivery, once to add a book in a *Section*, once to write to the log file (to make sure the lines are not mixed up), and once to get the instance to the *StatsManager* to make sure only one instance exists.

The two times the "synchronised" keyword is used to take a book, it is useful because we don’t want a book to be taken multiple times or any other issues related to the integrity of the lists of books in Sections or in the Delivery, the same is true for the action of putting a book in a section.

The "synchronized" keyword will give access to the resources in an First In First Out (FIFO) manner. Starvation should not happen, but since every thread should in average take the same amount of time and none have priority over another one it does not seem possible to improve the issue of starvation.

Deadlocks should not be possible, every function using "synchronised" will finish without issues and can’t loop indefinitely.