SPL 181 Assignment 2

Java 8 and Concurrent Programming

December 03, 2017

- TAs in charge of the assignment:
  - Hussien: hussien@post.bgu.ac.il
  - Matan Sar Israel: matansar@post.bgu.ac.il
- Submission due: 26/12/2017
- Unit test due: 07/12/2017 (see section 3.6)

1 Before You Start

- The goal of following assignment is to practice concurrent programming on Java 8 environment. This assignment requires a good understanding of Java Threads, Java Synchronization, Lambdas, and Callbacks. Make sure you go over the respective practical sessions which cover these topics.
- The Q&A of this assignment will take place at the course forum only. Critical updates about the assignment will be published in the assignment page on the course's website. These updates are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:
  - Read previous Q&A carefully before asking the question; repeated questions will probably go without answers.
  - Be polite, remember that course staff does this as a service for the students
  - You are NOT allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour.
- Only Majeed, the TA in charge of the course, can authorize extensions. In case you need an extension, contact him directly.

Figure 1: Good Luck Guys!

2 Introduction

In the following assignment you are required to implement an Actor ThreadPool, and to use it to build a University Management System. In Actor thread pool, each actor has a queue of actions. One can submit new actions for actors and the pool makes sure that each actor will run its actions in the order they were received, while not blocking other threads. The threads in the pool are assigned dynamically to the actors. As an SPL181 team you will implement an Actor Thread Pool.

It is very important to read the entire work before starting. Do not be lazy here, the work will be much easier if you read and understand the entire work in advance.

3 Part 1: Actor Thread Pool

3.1 Detailed Description

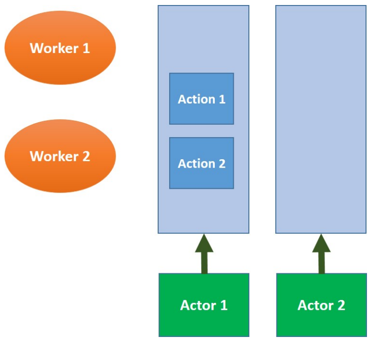
In actor thread pool, each actor has a queue of actions. One can submit a new action to the actor. The threads in the actor thread pool are assigned dynamically to the actors, in the following way. Each thread searches for an action to execute in all the actors' queues. Once the thread found such an action, it will prevent any other thread from fetching actions from that queue. Once it finished executing the action, it will allow other threads to fetch actions from this queue. And that thread will try to find another action to execute. Note, although a thread prevents other threads from processing actions from the queue which it is executing an action from it, the other threads are not blocked. The threads have to search for an action from other queues, and only if all the queues are empty or not available (threads work on them) then the threads will go sleep and should wake up once an action from an available queue is ready to be fetched.
An important observation, in Actor Thread Pool, the amount of actors can be significantly **greater** than the amount of threads. See figure 2.

3.1.1 Design Pattern: Event Loop

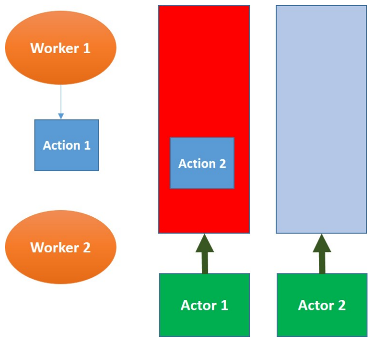
In this assignment you will implement the Event Loop design pattern. In such pattern, each thread in the Thread Pool has a loop. In each iteration of the loop, the thread tries to fetch an action and execute it. An important point in such design pattern is not to block the thread for a long time, unless there is no work for it. Blocking the thread for a long time while there is a work for it will have a bad effect in your implementation, since threads are idle although there is work for them. See figure 3.

3.2 Actors and Actions

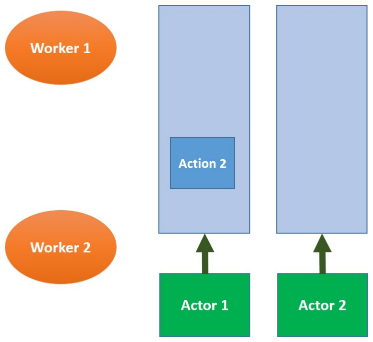
An action is a computational task. An actor is a computational entity that in response to actions it receives, an actor can: make local decisions, create more actors, send message to other actors. Actors may modify private state, but can only affect each other through messages.
In this assignment, the messages between actors are actions. An actor can submit an action to another actor's queue. A thread will fetch this action from the receiver queue and execute it. Note, that actors submit actions to another actor's queue, when that action may affect the private state of the receiver's queue, and by that avoiding locks since only one thread can access the private state of an actor at the same time.
You must not synchronize on the state of the actor, but you can use the state for the implementation of your Actor Thread Pool.



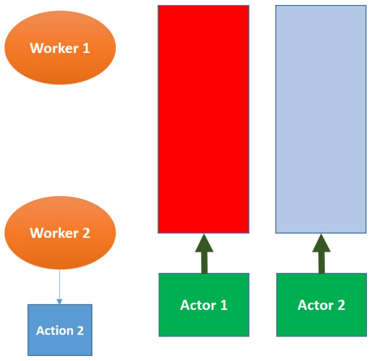
(a) Two Actors and Two Threads (Threads), actor 1 has two actions in his queue



(b) Thread 1 fetches action 1 from actor 1 and executes it. Thread 2 can not fetch actions from actor 1 queue



(c) Action 1 has been completed. Therefore, the queue of actor 1 is available for both threads



(d) Thread 2 fetches action 2 from actor 1 and executes it

Figure 2: Actor Thread Pool Description

3.3 Dependency Between Actions and Using Promises

In some applications the actions have interdependence constraints. Your thread pool should fulfill these constraints. The actions execution should be suspended until the actions it depends on are completed. Suspending an action should not suspend the thread or the actor. When an action is suspended, the thread should continue with another action, and the actor's queue should be available for fetching actions by any thread. The suspended action should be eventually continued only when all actions it waits for them are done. The approach to handle this is to enqueue the continuation of the resumed action on the same actor's queue whenever it is ready to be continued. In some point, one thread will fetch the continuation and execute it.

3.3.1 Promise Design Pattern

In order to enable the mentioned above. We use the promise design pattern (Using the Promise class you will implement in this assignment). A Promise is used for deferred computations and represents the result of an operation that has not been completed yet. Each action has a Promise Object which hold its result, whenever an action needs a result of another action, it passes to the action's Promise object a callback. This callback will be executed once the event is completed. Here is a motivation for using Promises. Suppose you want to execute an action which takes time, such as multiplication of a two matrices, A, B. You may write this code statement (assume you have Mult function):

```
result = Mult(A,B);
```

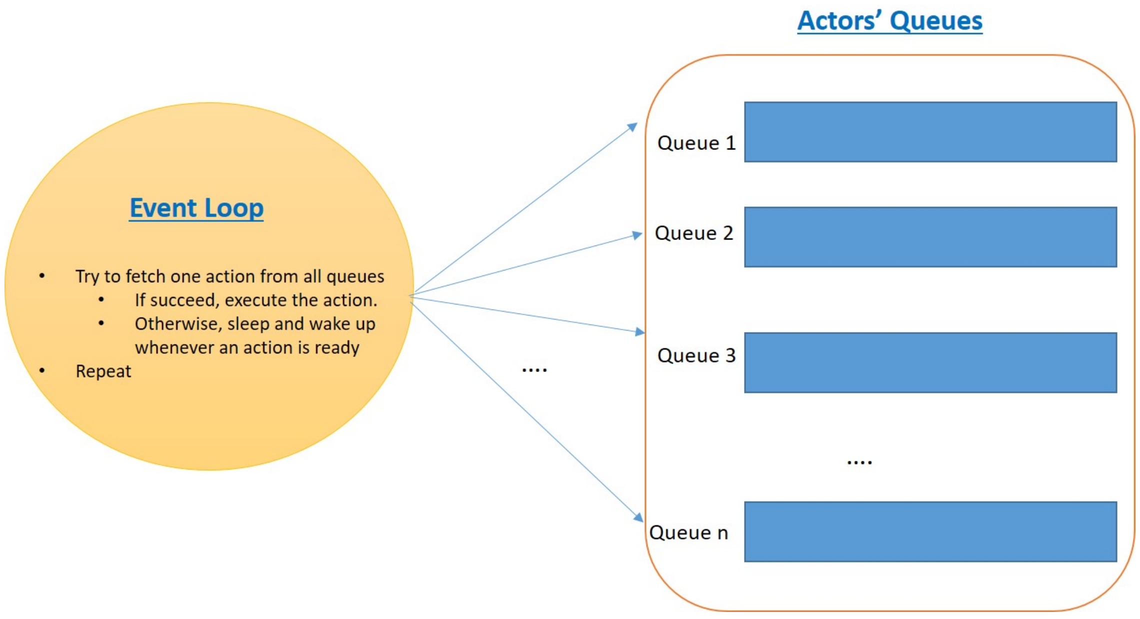

Actors' Queues
Event Loop
Try to fetch one action from all queues
If succeed, execute the action.
Otherwise, sleep and wake up whenever an action is ready
Repeat
....
Queue 1
Queue 2
Queue 3
....
Queue n

Figure 3: Event Loop

As Mult is a heavy operation, assume it takes 10 seconds to complete. Meanwhile, your thread is blocked from doing another tasks. One might want that the threads do other work while waiting for the result of Mult. This is exactly the goal of Promise. With Promise you can write something like this:

```
promise = Mult(A,B);
then(promise, ()->{print(result)});
int simple = 9-5;
```

In this example, instead of the actual result of Mult, you get a promise that you will get the actual result once Mult is completed. Now, Mult does not block your thread, the thread simply gets the promise and gets back. After the thread received the promise, it specify what to do with the result when it is ready (by passing a callback to `then` method). And continue working on other work. When the result is ready, the callback passed to `then` is executed.

3.4 Example of Actor Thread Pool

In order to explain how Actor Thread Pool works, assume that we want to write a program to manage banks. Each bank has a list of customers, where customers can transfer money to customers in different accounts. The goal is that banks do not block each other.
In actor thread pool, we can think about each one of the banks as actor. We submit actions for each bank. The threads in the thread pool will search for an available action in one of the banks and execute it while preventing other threads from executing an action of the same bank, that ensures that actions of the same bank are executed in the order the were received. Lets consider the action of transferring money from a customer A in bank 1 to customer B in bank 2. This action involves the authorization of both banks, updating the customers records and a receipt for both customers. Therefore, in the implementation of such action, bank 1 inserts a new action for bank 2 asking for confirmation for the transferring and updating customer B records. Bank 1 gets a promise from bank 2 to return him an answer. Meanwhile, bank 1 will continue handling other transactions for his customers. Once the action sent by bank 1 to bank 2 is completed, bank 2 will resolve the promise given to bank 1. Once the promise is resolved, the main action will be continued by inserting it again the queue of bank 1. In some point later, the action is fetched again, however, during the second execution the action should not be handled from scratch, and only a continuation of the action should be executed. The continuation will be executed and the money will be transferred from customer A to B. (see section 3.7 for code example)

3.5 Implementation Of Actor Thread Pool

In this part of work you are supplied with 4 classes that you will have to implement in order to construct a working actor thread pool. All of the classes are stored in the bgu.spl.a2 package. You are advised to download and read the javadoc of the supplied interfaces as a complementary material to this section. The rest of this section describes the given classes and the way they should be implemented.

3.5.1 The Actor Thread Pool Components: Actors and Threads

To this end, in our framework, each Actor has its own action's queue. Each actor has own **id** which is a String, and a `PrivateState` object. The Actor Thread Pool holds all the queues of all the actors in our system. The **submit** method of the the Actor thread pool, receives as parameter, an action (or list of actions) along with corresponding actor. If the actor is present in the thread pool, it enqueues the action to the queue. Otherwise, if is not present, it will create the actor's queue and submit the action to it.
The actor thread pool maintains also a set of threads. The thread mission is to execute the actions submitted to the actor. An important observation in this matter, is that the thread number can be too smaller from the number of actors. Therefore, one can not assign a thread per each actor. Rather, the assigning of actors to thread is done dynamically. That is, a thread searches for an action to execute in all the actors queue. Once the thread found such an action, it will prevent any other thread from executing tasks of the same actor.

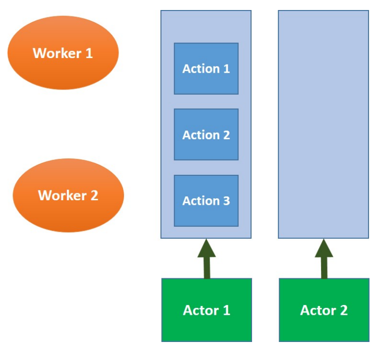
3.5.2 Actions and Promises

The queue of each actor holds objects of type Action. The Action is an abstract class, where each sub class has to define the method `start` only, which describes the behavior of the action. Each Action holds an object of type Promise which represents its deferred result. That is, this object will hold eventually the result of the action when it completes its computation. A Promise object holds a list of callbacks that will be executed when the Promise object is resolved (.i.e when it holds the actual result).
The flow is as follows, a thread fetches an action from a queue. The thread calls the `handle` method of the action, which in turn calls either the start method of the action or a callback we refer as `continuation` to continue the execution of the actor after it gets resumed.
As we mentioned, some actions might need the help of other actors in order to continue their computation. For this matter, that action will send a message to the other actor by `sendMessage` method. This methods gets as parameter the message which is of type Action, and the receiver actor. It will simply enqueue the message (action) to the other actor's queue (and if it is not present, will create it). By sending the message, the actor receives a Promise object from the receiver, this Promise will hold the result of the sent action in some point later. After the actor has sent all the messages, the action will be suspended for waiting on the sent actions. The actor proceeds with other actions on its queue. When all the actions it waits on are completed, the suspended action will be inserted again to the actor's queue. At some point, the thread will execute the suspended action again, and it actually will execute the continuation of the action.

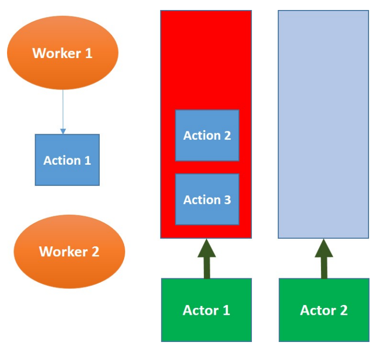
3.5.3 Summary and Additional Clarifications

The following is a summary and additional clarifications about the implementation of different parts of the framework.

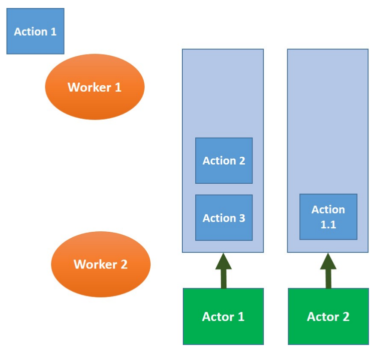
IMPORTANT: It is mandatory to read the all javadocs of the interfaces carefully, as these provides necessary hints of the implementation. You can add methods and fields to the classes. But you are NOT allowed to remove or change any method signature that we provided you with. Otherwise you will fail our automatic tests. Notice that you still can add the "synchronized" keyword to some methods if needed. However, remember that In concurrent programming, you should try to find good ways to avoid blocking threads as much as possible. This also applies to this assignment.



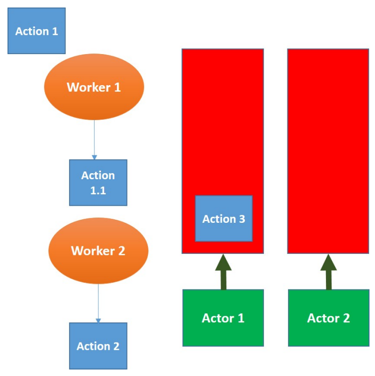
(a) Two actors, and two Threads. Actor 1 has three actions



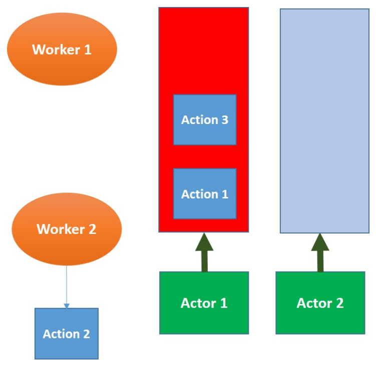
(b) Thread 1 fetches Action 1 from actor 1 and executes it. Thread 2 is prevented from fetching action from actor 1



(c) Action 1 sent a message Action 1.1 to Actor 2, and waits for its result by Action 1.1 promise object. The Threads can fetch actions from both actors.



(d) Thread 1 fetches Action 1.1 from Actor 2, and Thread 2 fetches action 2 from Actor 1



(e) Thread 1 completes Action 1.1. The Promise object of Action 1.1 is resolved, and a callback sent by Action 1 is called. Therefore, Action 1 is inserted again to actor's 1 queue

Figure 4: Task "life cycle" - a sample of class interaction within the Actor Thread Pool .

- **Action**: an abstract class that represents an action. An action is an object which holds the required information to handle an action in the system. The action also holds a Promise<R> object which will hold its result. The main action methods are:
  - start: This is the abstract method, that should be implemented for each action type, it implements the action behavior.
  - sendMessage: This methods submit an action (message) to other actor.
  - then: add a callback to be executed once all the given action are completed.
  - complete: resolves the internal result - should be called by the action derivative once it is done.
  - getResult: returns the task promised result.
- **Promise**: this class represents a promise result i.e., an object that eventually will be resolved to hold a result of some operation, the class allows for getting the result once it is available and registering a callback that will be called once the result is available. Promise includes these methods:
  - get: return the resolved value if such exists.
  - resolve: called upon completing the operation, it sets the result of the operation to a new value, and trigger all the subscribed callbacks.
  - subscribe: add a callback to be called when this object is resolved if while calling this method the object is already resolved - the callback should be called immediately.

  - isResolved: return true if this object has been resolved.

- **VersionMonitor**: Describes a monitor that supports the concept of versioning - its idea is simple, the monitor has a version number which you can receive via the method getVersion() once you have a version number, you can call await() with this version number in order to wait until this version number changes. You can also increment the version number by one using the inc() method.
- **ActorThreadPool**: manages all the the actor and threads in our system. The constructor of ActorThreadPool creates an ActorThreadPool which has n threads. The ActorThreadPool includes these methods:
  - submit: Enqueues an action to an actor's queue. If the actor is not present in the system, it will create it.
  - start: start the threads belongs to this thread pool.
  - shutdown: closes the thread pool - this method interrupts all the threads and wait for them to stop - it returns only when there are no live threads in the queue. After calling this method, one should not use the queues anymore

3.6 JUnit Tests for The Actor Thread Pool

When building a framework, one should change the way they think. Instead of thinking like programmer which writes software for end users, they should now think like a programmer writing a software for other programmers. Those other programmers will use this framework in order to build their own applications. For this part of the assignment you will build a framework (write code for other programmers), the programmer which will use your code in order to develop its application will be the future you while he works on the second part of this assignment.
Therefore, Before your class implementations are used in a production environment, you will have to verify that they work as expected. In order to do so, you must write unit tests for your classes. These unit tests should be based on the documentation of the classes, without being related on the implementation. After you implement the classes, you can run the unit test and check that your implementation is correct. An example of unit test is for `sendMessage` method in the Action class. This method should submit an action to other actor, but it must not wait for it (if the action needs to wait, the waiting should be done in the `start` method). You have to submit the unit tests alongside your project. Take care to write good tests.

Listing 1: Example of a Unit Test for resolve method in Promise

```java
@Test
public void testResolve(){
    try{
        Promise<Integer> p = new Promise<>();
        p.resolve(5);
        try{
            p.resolve(6);
            Assert.fail();
        }
        catch(IllegalStateException ex){
            int x = p.get();
            assertEquals(x,5);
        }
        catch(Exception ex){
            Assert.fail();
        }
    }
    catch(Exception ex){
        Assert.fail();
    }
}
```

In Unit Tests we also check that the method throws the expected exception. For example, `resolve` throws `IllegalStateException` if we try to resolve an already resolved promise.

You need to write Unit Tests for Promise and VersionMonitor classes and submit them by 07.12.17.
Here are some instructions. Notice: In Maven the unit tests are placed in src/test/java - this considered the same package as: bgu.spl.a2 These are the steps:

1. Download the interfaces
2. Extract them
3. Import Maven Project in Eclipse
4. To add a test for class X, right click on the class file X, add -> new -> JUNIT Test Case, and place it in src/test/java

Submit all your package, with all the classes.
The package hierarchy as specified in section 5.1. Make sure you compile with Maven.

3.7 A Code Example

The following simple example clarifies the usage of the framework to implement a system of banks. Each bank has an actor. The Transmission action is an action submitted for the actor of the sender bank in order to transfer an amount of money from its customer to another customer in different bank. In the start method, the bank sends a message to the second banks asking for confirmation, since the confirmation involves access to the private state of the customer in the other bank. Therefore, it is more efficient to ask the other bank to do this work (so we do not need locks, why?). The message is actually submitting an action to actor of the second bank. In `then` method, we wait until we receive an answer, and then call complete to resolve the result.
Note: You MUST NOT wait for the actions in sendMessage, since it is not necessary that the action waits for the actions it sends to the other actors.
In the Confirmation action we actually do some checks and decide on the answer.

First, in the main method we start the thread pool and submit actions. The following is an example of a simple main method which submits one action to the Thread Pool,

3.7.1 Simple Main Method

Listing 2: Submitting Transmission action to the Actor Thread Pool

```
1   ActorThreadPool pool = new ActorThreadPool (8);
2   Action<String> trans = new Transmission (100 , "A","B" , "bank2", "bank1");
3   pool.start();
4   pool.submit(trans, "bank1", new BankStates());
5   CountDownLatch l = new CountDownLatch(1);
6   trans.getResult().subscribe(() -> {
7   l.countDown();
8   });
9   l.await();
10  pool.shutdown();
```

1. In line 1 we create a thread pool with 8 threads.
2. In line 2 we create an action of type Transmission. Transferring 100 from "A" in "bank1" to "B" in "bank2".
3. In line 3 we start the thread pool.

4. In line 4 we submit the action to the thread pool. We pass the id of the actor of bank1. Also, in order for the thread pool to maintain a Private State object for "bank1" actor, we send an object of Private State (here we send BankState which extends PrivateState). We send this object as indicator of the Thread Pool of the the type of the actor. If the actor is already in the Pool, this object should be ignored. That is, the thread pool should hold only one PrivateState object per actor which is the one received when we submit an action for the actor for the first time. (Although we send an object each time we submit an action to the actor, these redundant objects are to be ignored.).
5. Note that you need to wait for the action to complete before existing the program. To achieve this we use CountDownLatch. Thus, we exit only when are sure that that action is done.

Next is a possible implementation of the Transmission action.

3.7.2 Transmission Action - Basic Implementation

```
1  public class Transmission extends Action<String>{
2  int amount;
3  String sender;
4  String receiver;
5  String receiverBank;
6
7  public Transmission(int amount, String receiver, String sender,
8  String receiverBank, String senderBank) {
9  this.senderBank = senderBank;
10 this.sender = sender;
11 this.receiver = receiver;
12 this.amount = amount;
13 this.receiverBank = receiverBank;
14 }
15
16 @Override
17 protected void start() {
18 List<Action<Boolean>> actions = new ArrayList<>();
19 Action<Boolean> confAction = new Confirmation(sender,receiver,
20 receiverBank, new BankStates());
21 actions.add(confAction);
22 sendMessage(confAction , receiverBank, new BankStates());
23 then(actions, () ->{
24 Boolean result = actions.get(0).getResult().get();
25 if (result == true){
26 complete("transmission succeed");
27 System.out.println("transmission succeed");
28 }
29 else{
30 complete("transmission failed");
31 System.out.println("transmission failed");
32 }
33
34 });
35 }
36 }
```

1. line 1. By Action<String> we define the result type of this action to be String.
2. line 17. We override the abstract start method. Here we place the behavior of the action.
3. line 19. We create another action of type Confirmation (we do not show in this document). This action will be submitted for "bank2"'s actor, since it requires access to its private state.

4. line 22. In sendMessage we put the action (which is the message) in "bank2"'s actor queue.
5. line 23. We announce that we are interested in each result of the actions in the list to proceed.
6. line 24-32. This is the continuation. When the result of the action is resolved, the action will be inserted again to the "bank1"'s actor, this continuation will be run in some point.

3.8 Testing the Actor Thread Pool

Before you hurry into implementing the University Management, you are advised to test your implementation of the Actor Thread Pool. To achieve this you might complete the implementation of the Banks System. Complete the Confirmation action so it does some non trivial work. Submit actions and check it does the work.

4 Part 2: University Management System

In this section you will simulate a University Management System using your Actor Thread Pool framework from part 1. The university has a set of departments, each department offers a set of courses to students (of all the departments) and each student can register for courses if he meets the prerequisites and there is an available space for him. The register/unregister request are considered until the end of the registration period. In this System we define Actors and their Private States. It is obligatory to comply to our definitions without any change.
In this section you are not allowed to use any type of synchronization on the Actors private state. You have to plan your implementation of the actions in such way that you will not need synchronization.
Synchronization is possible in SuspendingMutex class only. It can be implemented without synchronization, though.

4.1 Program Flow

The program is divided into three phases. Once a phase is completed, you will proceed to the next phase.

- Phase 1: All the open course actions appear in Phase 1. There might appear other actions as well.
- Phase 2: Any action can appear
- Phase 3: Any action can appear

At first you open all the suggested courses of all the departments. Once the second phase is completed, it is possible for the students to register for the courses.

Note: Students can be added while the registration is run.
Note: Students can register for courses in different departments.

4.2 Actors

In order to simulate the university, you should create three types of actors:

- An actor per student.
- An actor per course.
- An actor per department's secretary.

4.3 Private States Of Actors

Each actor should maintain in its private state a log of all the action it has preformed. In addition, the private states include:

– Department: A department's private state includes `list of courses` and `list of students in all the department`.

– Course: A course's private state includes `number of available spaces` and `list of students in the course`, `number of registered students`, and `prerequisites`.

– Student: A student's private state includes `grades sheet`, and `department's signature`. In the grades sheet appear all the courses the students learnt along with his grades.

Important: You are not allowed to use concurrent data structures to maintain the the private states of the actors.
Important: You are not allowed to extend the private state beyond what is described above, e.g, a department can not hold a list of students enrolled in each course, such data is part of the private state of the course only. Also, the department can not hold the grades sheet of the students, such is part of the private state of the student only.

4.3.1 Logging Actions

In its private state, the actor maintains a list of all the actions it has executed. The list holds only the description of the action as it is shown in the JSON file in section 4.9.2. Example: "Open Course", "Add Student", etc.

4.4 Actions

Following is a list of actions that you should enable.

Important: In each action you might create new actors, create new actions and submit them to other actors, etc. You must implement the action in the better way that ensures no synchronization on the private states of the actors.

1. Open A New Course:
   – Behavior: This action opens a new course in a specified department. The course has an initially available spaces and a list of prerequisites.
   – Actor: Must be initially submitted to the Department's actor.
2. Add Student:
   – Behavior: This action adds a new student to a specified department.
   – Actor: Must be initially submitted to the Department's actor.
3. Participating In Course:
   – Behavior: This action should try to register the student in the course, if it succeeds, should add the course to the grades sheet of the student, and give him a grade if supplied. See the input example.
   – Actor: Must be initially submitted to the course's actor.
4. Unregister:
   – Behavior: If the student is enrolled in the course, this action should unregister him (update the list of students of course, remove the course from the grades sheet of the student and increases the number of available spaces).
   – Actor: Must be initially submitted to the course's actor.
5. Close A Course:

  - Behavior: This action should close a course. Should unregister all the registered students in the course and remove the course from the department courses' list and from the grade sheets of the students. The number of available spaces of the closed course should be updated to -1. DO NOT remove its actor. After closing the course, all the request for registration should be denied.
  - Actor: Must be initially submitted to the department's actor.

6. Opening New places In a Course:
   - Behavior: This action should increase the number of available spaces for the course.
   - Actor: Must be initially submitted to the course's actor.
7. Check Administrative Obligations:
   - Behavior: The department's secretary have to allocate one of the computers available in the warehouse, and check for each student if he meets some administrative obligations. The computer generates a signature and save it in the private state of the students.
   - Actor: Must be initially submitted to the department's actor.
8. Announce about the end of registration period:
   - Behavior: From this moment, reject any further changes in registration. And, close courses with number of students less than 5.
   - Actor: Must be initially submitted to the department's actor.

4.5 Feature : Support Preferences List

As you will experience, when it comes to interesting (or easy) elective courses it is not a trivial mission to find a space for you. Therefore, we wish to support Preferences List, in which the student supply a list of courses he is interested in them, and wish to register for ONLY one course of them. The courses are ordered by preference. That is, if he succeed to register for the course with the highest preference it will not try to register for the rest, otherwise it will try to register for the second one, and so on. At the end, he will register for at most one course.

4.6 Warehouse

The warehouse class holds a finite amount of computers. Each computer has a Suspending Mutex (defined next section). When the department wants to acquire a computer, it should lock its mutex if it is free. And release it once it finished the work with the computer. If the computer is not free, the department should not be blocked. It should get a promise which will be resolved later when the computer becomes available.

4.6.1 Computer

A computer has a method of checkAndSign. This method gets a list of the grades of the students and a list of courses he needs to pass (grade above 56). The method checks if the student passed all these courses.
Each computer has two different signatures. One signatures for sign that the student meets the requirement and the other to sign the he does not.

4.7 Suspending Mutex

Holds a flag which indicates if the computer is free or not, and has a queue of promises to be resolved once the Mutex is available. In the Suspending Mutex there are two methods:

- Up: Release the mutex.

– Down: Acquire the mutex, If the mutex is not free, the thread should no be blocked. It should get a promise which will be resolved later when the printer becomes available.

Note: The Suspending Mutex can be implemented without any synchronization. However, using synchronization will be accepted as long as the implementation is blocking free.

4.8 Simulator

The simulator class is tasked with running the simulation. We may replace your ActorThreadPool implementation with our own during testing, so be sure to implement all simulation functionality in the Simulator class, not in ActorThreadPool!.
Once constructed, calling the simulators start() function will perform the following:

– Parse the Json Files.

– Submit actions to the thread pool passed to the method `attachActorThreadPool`.
DO NOT create an ActorThreadPool in start. You need to attach the ActorThreadPool in the main method, and then call start.

Calling simulator.end() will preform the following:

– shut down the simulation.

– returns a HashMap containing all the private states of the actors as serialized object to the file "result.ser".
You may do so using this code, or similar:

Listing 3: A Snippet Code For Writing The Output

```
    HashMap<String, PrivateState> SimulationResult;
    SimulationResult = SimulatorImpl.end();
    FileOutputStream fout = new FileOutputStream("result.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fout);
    oos.writeObject(SimulationResult);
```

– Your output filename MUST BE result.ser

4.9 Input Format

4.9.1 The JSON Format

All your input files for this assignment will be given as JSON files. You can read about JSONs syntax: http://www.json.org. In Java, there are a number of different options for parsing JSON files. Our recommendation is to use the library Gson. See the Gson User Guide (https://github.com/google/gson/blob/master/UserGuide.md) and APIs to see how to work with Gson. There are a lot of informative examples.

4.9.2 Input File

Defined below is a json input file that describes a single execution of our university. The simulation JSON file will be provided to your program as the first command line argument.
The following is an example of that file.
Important Note: Assume that the input is legal. That is, student does not try to register for course which is not exist, etc.

```
{
"threads": 8,
"Computers" : [
{
"Type":"A",
"Sig Success": "1234666",
"Sig  Fail": "999283"
},
{
"Type":"B",
"Sig Success": "4424232",
"Sig  Fail": "5555353"
}
],
"Phase 1" : [
{
"Action":"Open Course",
"Department": "CS",
"Course": "SPL",
"Space": "400",
"Prerequisites" : ["Data Structures", "Intro to CS"]
},
{
"Action":"Open Course",
"Department": "CS",
"Course": "Data Bases",
"Space": "30",
"Prerequisites" :  ["SPL"]
},
{
"Action": "Add Student",
"Department": "CS",
"Student": "123456789"
}
],
"Phase 2" : [
{
"Action": "Add Student",
"Department": "Math",
"Student": "132424353"
},

{
"Action": "Participate In Course",
"Student": "123456789",
"Course": "SPL",
"Grade": ["98"]
},
{
"Action": "Add Student",
"Department": "CS",
"Student": "5959595959"
},
{
"Action": "Add Spaces",
"Course": "SPL",
"Number": "100"
},
{
"Action": "Participate In Course",
"Student": "123456789",
"Course": "Data Bases",
"Grade": ["-"]
},
{
"Action": "Register With Preferences",
"Student": "5959595959",
"Preferences": ["Data Bases","SPL"],
"Grade": ["98","56"]
},
{
"Action": "Unregister",
"Student": "123456789",
"Course": "Data Bases"
},
{
"Action": "Close Course",
"Department": "CS",
"Course": "Data Bases"
},
{
"Action" : "End Registeration"
},
{
"Action" : "Administrative Check",
"Department": "CS",
"Students": ["123456789","5959595959"],
"Computer": "A",
"Conditions" : ["SPL", "Data Bases"]
}
],
"Phase 3": [
{
"Action" : "Administrative Check",
"Department": "CS",
"Students": ["123456789","5959595959"],
"Computer": "A",
"Conditions" : ["SPL", "Data Bases"]
}
]




}
```

The file holds a json object which contains the following fields:

- `threads` - an integer defining the amount of threads in the `Actor Thread Pool`.
- Computers - an array of computers in the warehouse, each computer has two signatures.
- Phase 1 - An array of all the open courses actions, and some other action might appear. All the actions in Phase 1 should be completed before proceeding to Phase 2.
- Phase 2 - An array of different actions. All the actions in Phase 2 should be completed before proceeding to Phase 3.
- Phase 3 - An array of different actions.

Note: the number and order of the actions might be different. You must insert the actions in the order they

appear in the json.
Participate In Course: In this action, you should try to register the student in the course (if there is available space and he meets the prerequisites of the course). If the mission is succeed, then if the grade is not "-", you should give him a grade.
Register With Preferences: In Grade, as in Participate In Course. Here we give a list of grades, each grade for a course. Remember: the student can register for one course at most from the supplied list.

Important: The simulation file should be given as argument in the main function. That is, you can't use a predefined name or location. The run command is:
mvn exec:java -Dexec.mainClass="bgu.spl.a2.sim.Simulator" -Dexec.args="myFile.json".
In Dexec.args we pass arguments to the main method - in our case this argument represents the simulation file (the file name is NOT necessary myFile.json).

5 Format and Submission Instructions

5.1 Packages

In order for your implementation to be properly testable, you must conform to the following package structure.

```
package structure
└bgu
  └spl
    └a2
      ├─Promise
      ├─Action
      ├─VersionMonitor
      ├─ActorThreadPool
      ├─PrivateState
      └sim
        ├─Warehouse
        ├─SuspendingMutex
        ├─Simulator
        ├privateStates
        │ ├─StudentPrivateState
        │ ├─CoursePrivateState
        │ └─DepartmentPrivateState
        └actions
```

You may add classes as you see fit. Your application should run correctly if we replace the your implementation of the first part with our own implementation.
Important: If you do not conform to the above structure, automated tests may fail, resulting in a reduction of your assignment grade.
Important: All the actions which extend the Action should be placed in bgu.spl.a2.sim.actions.

5.2 Building the Application: Maven

In this assignment you are going to use maven as your build tool. Maven is the de-facto java build tool. In fact, maven is much more than a simple build tool, it described by its authors as a software project management and comprehension tool. You should read a little about how to use it, you can find a short tutorial here: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html.
IDEs such as eclipse, netbeans or intellij all have native support for maven and may simplify interaction with it - but you should learn yourself how.
Use the following command to install maven in the lab:
/home/studies/course/spl161/java/install-maven
Please do not attempt to install it in your lab any other way.

In this assignment you required to work with java 8 (1.8), You will need to have JDK8 installed and to configure your maven build to use java 8 - you should findout how to acheive that yourself.

5.3 Deadlocks, Waiting, Liveness and Completion

Deadlocks You should identify possible deadlock scenarios and prevent them from happening.
Waiting You should understand where wait cases may happen in your program, and how you have solved these issues.
Liveness Locking and Synchronization are required for your program to work properly. Make sure not to lock too much so that you don't damage the liveness of the program. Remember, the faster your program completes its tasks, the better.
Completion As in every multi-threaded design, special attention must be given to the shut down of the system. When the ActorThreadPool receives a "shutdown" command, the program needs to terminate. This needs to be done gracefully, not abruptly.

5.4 Documentation and Coding Style

There are many programming practices that **must** be followed when writing any piece of code.

- Follow Java Programming Style Guidelines. A part of your grade will be checking if you have followed these mandatory guidelines or not. It is important to understand that programming is a team oriented job. If your code is hard to read, then you are failing to be a productive part of any team, in academics research groups, or at work. For this, you must follow these guidelines which make your code easier to read and understand by your fellow programmers as well as your graders.
- Do not be afraid to use long variable and method names as long as they increase clarity.
- Avoid code repetition - In case where you see yourself writing same code block in two different places, it means this code must be put in a private function, so both these places may use it.
- Full documentation of the different classes and their public and protected methods, by follow- ing Javadoc style of documentation. You may, if you wish, also document your local methods but they are not mandatory.
- Add comments for code blocks where understanding them without, may be difficult.
- Your files must not include commented out code. This is garbage. So once you finish coding, make sure to clean your code.
- Long functions are frowned upon (30+ lines). Long functions mean that your function is doing too many tasks, which means you can move these tasks to their own place in private functions, and use them as appropriate. This is an important step toward increased readability of your code. This is to be done even in cases where the code is used once.
- Magic numbers are bad! Why? Your application must not have numbers throughout the code that need deciphering.

Your implementation must follow these steps in order to keep the code ordered, clean, and easy to read.

5.5 Submission Instructions

- Submit all the package as specified in section 5.1.
- You need to submit the Unit Tests as well, therefore your POM should include a dependency for the JUNIT.
- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
- You must submit one .tar.gz file with all your code. The file should be named "assignment2.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.
- The submitted zip should contain the src folder and the pom.xml file only! no other files are needed.
- Extension requests are to be sent to majeek at cs. Your request email must include the following information:
  * Your name and your partner's name.
  * Your id and your partner's id.
  * Explanation regarding the reason of the extension request.
  * Official certification. Request without a compelling reason will not be accepted.