Concurrent and Distributed Systems

Assignment

Report

developed by
Gabriele Baris

Delivery: 13/01/2017

Table of Contents

Introduction

The entire project has been developed with Java version 8u111.
The project is structured according to the following folder tree:

- *assignment1*
  - *locks*: Java files for `FairLock` implementation
  - *test*: Java files for testing
- *assignment2*:
  - *clients*: Java files for clients implementation
  - *managers*: Java files for managers implementation
  - *test*: Java files for testing
- *assignment3*:
  - *implementation*: Java files for manager implementation
  - *models*: LTSA models
  - *test*: Java files for testing

Folders for Java files also represent their package.
Inheritance and interfaces were not strictly needed but were useful.

Assignment 1

Requirement 1

Implement a synchronization mechanism similar to the mechanism provided by Java within the `java.util.concurrent` package (explicit `Lock` and `Condition` variables) but whose behaviour is in accordance with the semantic ***"signal-and-urgent"***.
For the implementation of this mechanism you can use only the built-in synchronization constructs provided by Java (i.e. `synchronized blocks` or `synchronized methods`) and the methods `wait()`, `notify()` and `notifyAll()` provided by the class `Object`).
In particular:

- implement the class `FairLock` that provides the two methods `lock()` and `unlock()`, to be used to explicitly guarantee the mutual exclusion of critical sections. Your implementation must guarantee that threads waiting to acquire a `FairLock` are awakened in a **FIFO order**.
- implement also the class `Condition` that provides the two methods `await()` and `signal()` that are used, respectively, to block a thread on a Condition variable and to awake the first thread (if any) blocked on the `Condition` variable. In other words `Condition` variables must be implemented as **FIFO queues**. The semantics of the signal operation must be ***"signal-and-urgent"***. Remember that every instance of the class `Condition` must be intrinsically bound to a lock (instance of the class `FairLock`). For this reason, the class `FairLock` provides, in addition to methods `lock()` and `unlock()`, also the method `newCondition()` that returns a new `Condition` instance that is bound to this `FairLock` instance.

Implementation notes

In order to have each condition bounded to the `FairLock` instance, the `FairCondition` class has been developed as an inner class.
`FairLock` and `FairCondition` implement the interfaces provided by the `java.util.concurrent.locks` package in order to use them as the default `Lock` and `Condition` classes. All methods not explicitly requested have not been implemented and will return an `UnsupportedOperationException`.
To have more detailed information about what's happening inside the `Lock`, the FairLock must be compiled with the `D` flag set to `true`.

In order to implement the requested policy, the easiest approach would be the one involving all threads waiting on the same waitset, selecting the one to wake-up, awake all the waiting threads with a `notifyAll()` and put again all the unwanted threads to sleep, while the desired one would continue. This approach is correct, but leads to a great number of context switches, since if *n* threads are waiting, all of them are awakened and then *n-1* must come back to sleep.
Because of that it has been decided to use a more complex (but more efficient) approach: a list of event semaphores. Once a thread needs to be suspended for a certain condition to

become true, it is added to a queue and suspended on its event semaphore; once the condition becomes true, only one thread is awakened and continues, reducing the number of context switches. Using a queue (in Java, `LinkedList`) the FIFO order is easily implemented.

The owner of the lock is explicitly saved in order to correctly “passing the baton”. Since Java is implemented according to the “signal-and-continue” semantics, all the awakened threads returns in the entry queue and are possibly “mixed” with new incoming threads: the `owner` variable allows only the awakened thread to acquire le lock, while all the other threads would block.

Different `synchronized blocks` on different objects have been used, one after the other. Doing it in a *nested fashion* would have lead to deadlocks related to *Nested Monitor Calls*.

The implementation can be found in the class `FairLock`.

Assignment 2

Requirement 2.0

As a simple example of the use of the previous mechanism, implement a **manager of a single resource** that dynamically allocates the resource to three client threads: `ClientA1`, `ClientA2` and `ClientB`.
If the resource is in use by `ClientA1` or by `ClientA2`, when it is released and both `ClientB` the other `ClientA` are waiting for the resource, `ClientB` must be privileged.

Implementation notes

According to the specifications, clients of type A can suffer starvation. This scenario is not handled because it’s part of the priority rule itself and trying to manage it would violating the specifications.

The implementation is trivial, since the `FairLock` is implemented according to the *“signal-and-urgent”* semantics and so this is a very simple manager implemented with a monitor, as seen many times during the course.

The implementation can be found in the class `UrgentManager`.

Requirement 2.1

Provide also the implementation of the same manager but now by using the analogous mechanism provided by Java (`Lock` and `Condition` variables whose behaviour is in

accordance with the semantics *“signal-and-continue”* and point out the differences, if any, between this implementation and the previous one.

Implementation notes

To implement this manager, I needed to do “outside” what I did “inside” the `FairLock`: so the solution is a bit more complicated than the previous one, because Java primitives work with the *“signal-and-continue”* semantics. Because of that, two waiting queues are used (beyond condition variables) and also spurious wake-ups must be managed since the Java implementation allows them to happen.

The implementation can be found in the class `ContinueManager`.

Since the behavior is the same (as asked by the requirements), the main difference is in the implementation: since the FairLock is *“signal-and-urgent”* by itself, implementation at point 2.0 is very simple, while the one in 2.1 is more complicated since you have to explicitly define this behavior with synchronization primitives that are *“signal-and-continue”*.
Also, the differences are all the ones involving the two semantics: i.e., since in *“signal-and-urgent”* the awakened thread soon comes back executing, the precondition is true for sure, and the test can be done using an `if`; for *“signal-and-continue”* this is not true because the thread comes back to the entry queue and the test must be performed in a `while`.

Assignment 3

Requirement 3.0

By using the language FSP, provide the design model of the problem described at point 2.0.

Implementation notes

The design model is done according to what specified at point 2.0, so with 2 clients of type A and 1 client of type B. So the processes involved in the synchronization are 3: `CLIENT_A`, `CLIENT_B` and `MANAGER`.

Since in FSP there is no way to get the type of the calling thread, the manager provides two different actions to `request_*` the resource by the two tasks. The release is the same since all the threads must do the same.
Since actions in FSP are atomic, to model the possible suspension of the task, the action request is splitted in the actions `request_*` and `get_*` .

The implementation can be found in the file `design_model.ltsa`.

Requirement 3.1

From the design model described at point 3.0, derive the corresponding Java program implemented by using the `Lock` and `Condition` variables provided by java and whose behaviour is in accordance with the semantics ***"signal-and-continue"***.

Implementation notes

Using local processes, the design model becomes:

```
MANAGER = FREE,
FREE = (request_a->get_a->MANAGE[0][0] |
        request_b->get_b->MANAGE[0][0] |
        release->ERROR),
MANAGE[0][0] = (request_a->MANAGE[1][0] |
                request_b->MANAGE[0][1] |
                release->FREE),
MANAGE[1][0] = (request_a->MANAGE[2][0] |
                request_b->MANAGE[1][1] |
                release->PASS_A[1][0]),
MANAGE[2][0] = (request_a->ERROR |
                request_b->ERROR |
                release->PASS_A[2][0]),
MANAGE[0][1] = (request_a->MANAGE[1][1] |
                request_b->ERROR |
                release->PASS_B[0][1]),
MANAGE[1][1] = (request_a->ERROR |
                request_b->ERROR |
                release->PASS_B[1][1]),
PASS_B[0][1] = (get_b->MANAGE[0][0]),
PASS_B[1][1] = (get_b->MANAGE[1][0]),
PASS_A[1][0] = (get_a->MANAGE[0][0]),
PASS_A[2][0] = (get_a->MANAGE[1][0]).
```

This leads to the following table (note that some states are impossible because the number of clients is 3 and they can not be blocked all together or there would be a deadlock). Also, there are some transitions that lead to an error (for example, since there are 2 clients A, I cannot accept a `request_a` if already 2 of them are blocked).

State	Encoding	request_a	request_b	release
FREE	0	1	1	ERROR
MANAGE[0][0]	1	2	4	0
MANAGE[1][0]	2	3	5	8
MANAGE[2][0]	3	ERROR	ERROR	9
MANAGE[0][1]	4	5	ERROR	6
MANAGE[1][1]	5	ERROR	ERROR	7
MANAGE[2][1]	IMPOSSIBLE			
PASS_B[0][1]	6		ERROR	ERROR
PASS_B[1][1]	7		ERROR	ERROR
PASS_B[2][1]	IMPOSSIBLE			
PASS_A[1][0]	8			ERROR
PASS_A[2][0]	9	ERROR		ERROR

The next states for `get_a` and `get_b` have not been inserted in the table because they can be easily derived from the previous FSP code. Also, in the Java code `request_*` and `get_*` will be merged inside a single method.

At this point, the code in the class `ModelManager` has been written. Even if some code is redundant it has been decided to leave it because this way it's more readable and each piece can be compared to the model itself. It has also been decided to leave the methods `request_a` and `request_b` even if in Java there is a way to know which is the calling thread.

Requirement 3.2

By modeling this implementation with the FSP language, verify that it satisfies the problem's specification.

Implementation notes

According to what we've seen during the course, the content of the file `implementation_model.ltsa` has been provided starting from the code at point 3.1. The implementation model has then been tested against the property built using the design model and no property violations were found.