MASTER DEGREE in EMBEDDED COMPUTING SYSTEMS
A.Y. 2016 - 2017

Report of the Project
Concurrent and Distributed Systems

Full Professor
Paolo Ancilotti

Silvio Bacci

Index

Chapter 1
Assignments

1.1 Assignment 1

Implement a synchronization mechanism similar to the mechanism provided by Java within the java.util.concurrent packages (Explicit Locks 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).

1.1.1 FairLock

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.

1.1.2 Condition

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.

1.2 Assignment 2

1.2.1 Resource Manager

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.

1.2.2 Lock and Condition provided by Java

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.

1.3 Assignment 3

1.3.1 Design Model

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

1.3.2 Implementation in Java

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*.

1.3.3 Implementation Model

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

Chapter 2
Assignment 1: FairLock and Condition

2.1 EventSemaphore

In order to implement a FairLock what we need is a queue of elements to save the order of execution of the method lock(). Besides we need an Object on which we can block the current thread. So the idea is to create a Class called EventSemaphore that has the following synchronized methods:

- block(): if the semaphore is "red" the current thread blocks itself.
- unblock(): we change the state of the semaphore and we send a notify to the blocked thread (we use notifyAll() only to be more general, in this case it is useless).

We could think at this semaphore as a "one shot" semaphore (used only once and then removed) but we developed the semaphore in a generic way in order to be more general.
This semaphore is called "Event Semaphore" because what we need is a sort of synchronization between the blocked thread and the signaller. In fact if the unblock operation is executed before the block method, when the (not yet) blocked thread will execute the block operation it will not block itself. This could be dangerous (someone could enter between unblock and block operations and in that case we could have 2 threads in critical section) but, as we will see in the next paragraph, we use the "passing the baton" technique and so we are sure that no one can enter in between.
In order to avoid spurious wake up we use a cyclic scheme in which we test every time the colour of the semaphore.
Each EventSemaphore not referred anymore will be destroyed by the Garbage Collector.
All the exceptions are thrown to the high level (we don't care about them).

2.2 FairLock

The idea is to use an ArrayList of type EventSemaphore in which we can insert the semaphore on which the thread will block itself. The insert operation in the queue and the block operation are not executed in mutual exclusion and so there is the possibility to insert the semaphore in the queue but not to block the thread. If an unlock operation is executed before blocking the thread, the notify operation is executed and the thread not blocked yet can continue without blocking itself.
There is no problem because we use the "passing the baton" technique and so no other threads can acquire the lock after a notify operation (only the awakened thread can continue by acquiring the lock).
In the unlock operation every time we check if an urgent thread is ready (it has the maximum precedence), otherwise we check if a new "entry" thread is arrived and finally we release the lock if no one is arrived.
The lock operation is not a synchronized method because we want to block the thread on a different monitor (otherwise we have nested monitor calls and so deadlock). We decided to call a synchronized method (isBusy()) that tests the condition and eventually:

- Insert a semaphore in the ArrayList if the current thread should be blocked
- Change the state of the variable busy in order to say that the lock is not available

We used some methods in order to manage the ArrayLists and these methods are not synchronized because they are always executed when we have the lock of the Monitor.
The FIFO order is guaranteed when we execute the isBusy() operation. In fact there is the possibility that a thread calls the operation lock but before blocking itself another thread executes the same operation and enters in the critical section of the method isBusy().

So, even if the other thread is arrived first we take the order only of the execution of the synchronized method. This problem is impossible to solve if we have to use only the low mechanisms of Java.

2.3 Condition

In order to implement a signal-and-urgent behaviour we need to use 2 different queues: urgent and condition queue. The urgent queue is shared between all the conditions variable associated with the same instance of the Class FairLock (the queue is created in the FairLock and passed to the class Condition at the creation of a Condition's instance), while the condition queue is one for each condition variable.

The code was developed using the structure of the semantic signal-and-urgent that is:

- ***enter-the-monitor*:**

```
if( <the monitor is locked> ) {
      <add the executing thread to the entry queue>;
      <block the thread >; }
else
      <lock the monitor>;
```

- ***leave-the-monitor*:**

```
if( < there is at least a thread in the urgent queue > )
      <select and remove one thread from the urgent queue and restart it>;
            //this thread will occupy the monitor next
else if( < there is at least a thread in the entry queue > )
      <select and remove one thread from the urgent queue and restart it>;
            //this thread will occupy the monitor next
else
      <unlock the monitor>;
      //the monitor will become unoccupied
```

- **wait(c):**

```
<add the executing thread to the queue of the condition c>;
leave-the-monitor ;
<block the thread >;
```

- **signal(c):**

```
if( < there is at least a thread in the queue of the condition c > ) {
      <select and remove one thread P from the queue of the condition c>;
      <add the executing thread to the urgent queue>;
      <restart thread P>; // so P will occupy the monitor next
      <block the executing thread >;
}
```

The enter-the-monitor operations are exactly the operations executed by the lock method of the class FairLock. Also the leave-the-monitor operations are the same of the unlock method. So what we need in the ResourceManager will be a lock at the beginning of every method and an unlock at the end.
Also in this case the queues are ArrayList of type EventSemaphore and so the spurious wake up are avoided directly by using the cyclic scheme of the semaphore.

Chapter 3
Assignment 2: Resource Manager

3.1 Resource Manager with FairLock and Condition

In order to implement it we need to use lock and unlock methods, respectively at the beginning and at the end of the request and release operations.
In the request operation if the resource is not available we need to block the thread on its own Condition variable (one Condition for all threads of type A and one for threads of type B).
In the release operation we check if a Thread of type B is blocked and in that case we need to wake up it, otherwise if a thread of type A is blocked we wake up it and finally, if there are no blocked threads we release the resource simply.
In this case we use the "passing the baton" technique and so in the request operation we need only an if (no one can enter in between and so we don't check the condition again).

3.2 Resource Manager with Lock and Condition provided by Java

In order to implement it we need to use lock and unlock methods, respectively at the beginning and at the end of the request and release operations.
In this case we implement the Lock as a ReentrantLock with fairness (FIFO order is guaranteed both for lock acquire and for awakened threads from condition variables).
The FIFO order is already guaranteed but in general we need to save the blocked threads if we want to avoid spurious wake up. In fact if a spurious wake up arrives the thread could continue and so we must check if the condition is already true or not (a cyclic scheme is sufficient). But what happens if we block the thread again because of a spurious wake up? Now the thread is in the last position of the queue and this is a problem because we lost FIFO order. So we need an array of conditions (one for each thread) on which we can block the threads and 2 ArrayLists (one for each type of threads) in order to save the arriving order. Of course, for the thread B this queue is useless but it was used in order to be more general.
We use the same approach of the previous manager (the structure of the methods is the same, by using the "passing the baton" technique) but in this case we need to save the arriving order and every time we need to check if we are or not in the queue. In fact in the release operation we remove the thread we want to wake up and so the awaken thread should be check if it is still in the queue or not. If it is in the queue a spurious wake up was happened and we must block the thread again.

Chapter 4
Assignment 3: Design Model, Implementation in Java and Implementation Model

4.1 Design Model

According to the specification of the point 1.2.1 we developed a design model. In order to develop it we used 5 operations/transitions:

- acquire_a : the thread is of type A and it wants to acquire the resource.
- acquire_b : the thread is of type B and it wants to acquire the resource.
- endacquire_a : this operation is executed when an A thread finishes the acquire operation (no one can execute between an acquire and an endacquire of the same thread, unless the thread must be blocked).
- endacquire_b : this operation is executed when an B thread finishes the acquire operation (no one can execute between an acquire and an endacquire of the same thread, unless the thread must be blocked).
- Release: releases the resource and wakes up eventually a thread (B has the precedence).

4.2 From the Design Model to the implementation in Java

We need to translate the model using not indexed states (each local process identifies an internal state of the monitor) where each local process is defined by a single action prefix (a single transition) or a set of alternative action prefixes.
So the translation is:

```
RESOURCE_MANAGER = (acquire_a -> ENDACQUIRE_A[0] | acquire_b -> ENDACQUIRE_B[0] | release -> RESOURCE_MANAGER),

ENDACQUIRE_A[0] = (endacquire_a -> ACQUIRE_A[0]),
ENDACQUIRE_A[1] = (endacquire_a -> ACQUIRE_A[1]),
ENDACQUIRE_B[0] = (endacquire_b -> ACQUIRE_B[0]),
ENDACQUIRE_B[1] = (endacquire_b -> ACQUIRE_B[1]),

ACQUIRE_A[0] = (acquire_a -> ACQUIRE_A[1] | acquire_b -> ACQUIRE_A[2] | release -> RESOURCE_MANAGER),
ACQUIRE_A[1] = (acquire_b -> ACQUIRE_A[3] | release -> ENDACQUIRE_A[0]),
ACQUIRE_A[2] = (acquire_a -> ACQUIRE_A[3] | release -> ENDACQUIRE_B[0]),
ACQUIRE_A[3] = (release -> ENDACQUIRE_B[1]),

ACQUIRE_B[0] = (acquire_a -> ACQUIRE_B[1] | release -> RESOURCE_MANAGER),
ACQUIRE_B[1] = (acquire_a -> ACQUIRE_B[2] | release -> ENDACQUIRE_A[0]),
ACQUIRE_B[2] = (release -> ENDACQUIRE_A[1]).
```

Now we need to map all the 12 states into an integer values (from 0 to 11):

State	Encoding
RESOURCE_MANAGER	0
ENDACQUIRE_A[0]	1
ENDACQUIRE_A[1]	2
ENDACQUIRE_B[0]	3
ENDACQUIRE_B[1]	4
ACQUIRE_A[0]	5
ACQUIRE_A[1]	6
ACQUIRE_A[2]	7
ACQUIRE_A[3]	8
ACQUIRE_B[0]	9
ACQUIRE_B[1]	10
ACQUIRE_B[2]	11

Then we need to underline for each state which is the next state when we execute a certain transition.

States	acquire_a	acquire_b	endacquire_a	endacquire_b	release
0	1	3	-1	-1	0
1	-1	-1	5	-1	-1
2	-1	-1	6	-1	-1
3	-1	-1	-1	9	-1
4	-1	-1	-1	10	-1
5	6	7	-1	-1	0
6	-1	8	-1	-1	1
7	8	-1	-1	-1	3
8	-1	-1	-1	-1	4
9	10	-1	-1	-1	0
10	11	-1	-1	-1	1
11	-1	-1	-1	-1	2

According to these values we implemented it in Java by ignoring spurious wake up.

4.3 Implementation Model

By the implementation in Java we derived the implementation model and finally we check the specification of the problem, directly obtained by the design model.

Chapter 5
Testing

5.1 Test Code

Every time we added a new component we also provided a test code. These test codes usually create a certain number of threads (3) that share some objects (locks, conditions, resource manager and so on). To test the managers we also created a Resource that is an integer value incremented by each thread. Inside the code there are some “print” that are useful to debug the same code. After a certain number of proofs we don't know if the program is correct but we can say that for lots of test the code works well in accordance with the specifications.