Report

Matteo Rotundo

2015/2016

Contents

1 FairSem

This is Part1 of the assignment. It will be shown all the techniques and test adopted.

1.1 Assignment Details

The class FairSem must implement a FIFO semaphore. For the implementation of this class, can be used only the java low-level synchronization features (synchronized blocks, synchronized methods and the Object's methods wait(), notify() and notifyAll(). The solution should take into account, and work correctly, even in the case of spurious wake-up.

1.2 Implementation

In order to guarantee the fairness (i.e. access in FIFO fashion) among the threads that try to access to a critical section protected by the FairSem, mainly are used two strategies. The first is to use the passing the baton technique. This guarantees that eventually, a new thread that tries to get the lock, could not surpass blocked threads. The second strategy adopted, is to handle blocked thread in a FIFO queue.

FairSem have the following private data structures:

- *semvalue:* It's the value of the semaphore;
- *blocked: It's* the FIFO queue of blocked threads;
- *awakened:* It's the FIFO queue of the awakened threads (soon it will be clear why it is useful);
- *test_mode:* it's a boolean that indicates if the FairSem is in test mode;
- *entry_P and exit_P*: Both are FIFO queue and are useful for the the test;

Besides *FairSem* provides the constructor and the methods P() , V() and state().

1.2.1 Constructor

It has the following declaration:

```
FairSem(int value, int Nthreads, boolean val);
```

Thus:

1. The semaphore will be initialized to value;
2. FairSem will handle Nthreads;
3. val indicates if handling FairSem in test mode;

1.2.2 P()

This operation is for requesting the access to the critical section.

In this method it's checked if the value of the semaphore is green (i.e. > 0), otherwise the thread is blocked in the *blocked* FIFO queue. The thread enters in a loop , where the condition to go out is that the tid of the current thread must be equal to the tid of the first thread in the awakened queue, and is suspended, so that condition will be checked each time thread will woken up. This avoids wrong situations due to spurious wake-up.

The *awakened* queue is inserted for handling this critical situation:

Let suppose that the semaphore is initialized to n >1) . Let suppose that there a n threads in the critical section and other threads are blocked. If there are some V of the n threads and some blocked threads get the lock before the selected thread, every threads return in the wait set even if they are awakened with some V after the first. (i.e. some V have no effects). In order to avoid this situation it's needed a queue to remember the awakened threads and a notifyAll after the wait (if the awakened queue is not empty of course).

1.2.2 V()

This operation signals the exit of a thread from the critical section. If some thread is blocked, then it's extracted from the blocked queue and inserted in the awakened queue passing the baton. It's done a notifyAll because it's not possibile to wake up a specific thread in the wait set. Otherwise it's increased the value of the semaphore.

1.2.3 State()

It returns true if the semaphore is green, otherwise returns false. The reason because it's useful will be clear with the implementation of the ArrayPort.

1.3 Test

The purpose of the testing phase is to verify that the order of threads entering P() is the same of the threads leaving P(). So, enterings and leavings are recorded in two queues. The number of threads and the value of the semaphore can be inserted by the user after the run.

Running:

```
> make Part1

> java TestFairSem
```

2 SynchPort

This is Part2 of the assignment. It will be shown all the techniques and test adopted.

2.1 Assignment Details

The class SynchPort<T> simulates a generic synchronous port that provides the two methods:

1. **pr.send(Message<T>m)** called by a process to send the message m to the port pr;
2. **pr.receive()** called by a process to receive a message from the port pr. The received message is returned by the function;

According to the concept of port, given a port pr, more processes can send messages to pr while a single process can receive messages from pr. (pr must be declared as a public field in the receiving process).

Message<T> is the type of the parameter passed to the function send and the type of the object returned by the function receive. An object of this type must contain at least two fields: a value of type T, which represents the information sent by the sending process, and a field which allows the receiving process, when it is needed, to return a response message to the sending process.

2.2 Implementation

SynchPort port provides a *contructor* and the methods: *send(Message<T>), receive(), statePort().*

The private variables of this class are:

- *data*: a buffer on which exchanged messages are temporarily stored;
- *empty:* the semaphore associated to the emptiness of m;
- *full*: the semaphore associated to the fullness of m;
- *waitR*: the semaphore thats guarantees the *send()* to be synchronous;

The techniques adopted are a *split-binary semaphore* and a *rendez-vous* at the end of the operations. The split binary semaphore guarantees:

1. no overflow and no underflow;
2. no deadlock for the producer and the consumer;
3. mutual exclusion;

The receive operation will be blocking if no message is sended. But in order to guarantee the send operation to be blocking is needed a rendez-vous. Thus, the sender will wait the receiver before ending the operation.

2.2.1 Message<T>

It is the class that represents the message exchanged. Its public fields are:

- *data:* information of the message;
- *response:* SynchPort where eventually the sender could wait for the response;
- *tid:* Thread-identifier of the sender process;
- *index:* Used to save the index of the Port after a receive in Portarray;
- *priority:* Priority of the thread;

2.2.2 SynchPort(int Nsenders)

It's the constructor of the class.

Nsenders is the number of the senders that can use the port. The semaphore full is initialized with 0 and empty with 1. Then at the end, sync is initialized with 0.

2.2.3 send(Message<T> m)

The caller of this operation must wait until SynchPort is empty, waiting on the FairSem empty. After this, the buffer data can be filled with the message and this will be notified with the signal on the full FairSem. At the end the sender waits on waitR FairSem in order to guarantee the port to be synchronous.

2.2.4 receive()

The caller of this operation must wait until some data is available in the buffer, waiting eventually on the full FairSem. Then, when it's available the message could be retrieved and this is notified with a signal to the empty FairSem. In the end, it's done a signal on the waitR semaphore in order to conclude the rendez-vous.

2.2.5 state()

It returns true if the value of the semaphore is 1, otherwise 0.

All the methods shown are defined synchronized in order to provide the mutual exclusion.

2.3 Test

The purpose of the test is to verify if:

1. data is correctly exchanged;
2. send operation is blocking;
3. the field response works;

The sender forwards a data to the receiver and waits the response. The receiver sleeps for an amount of time in order that the sender will do first the operation (i.e. it will test if the operation is blocking). Then it prints the data received in order to verify if data is correctly exchanged. It will be done an operation of increasing and then the data will be sent on the response synchronous port of the sender. If the result printed by the sender is correct and no deadlock occurs, the field response works.

Running

```
> make Part2
> java TestSynchPort
```

3 PortArray

This is Part3 of the assignment. It will be shown all the techniques and test adopted.

3.1 Assignment Details

PortArray<T> defines an array of generic synchronous ports that provides two methods:

- **pa.send(Message<T> m, int p)**: called by a client process to send the message m to the port whose index is p within the array of ports pa;
- **pa.receive(int v[], int n):** called by a process to receive a message from n of the ports of the array pa. The first parameter v is an array of integers whose dimension is the integer n passed as second parameter. The values contained in the array v represent indexes of ports of the array pa;

When a process calls the function pa.receive(int v[], int n) it is blocked if all the ports, whose indexes are contained in the array v, are empty. In this case, the process will be awakened when a message arrives in any one of these ports. Viceversa, if one or more of these ports contain messages, one of these ports is chosen and a message is received from that port.

The choice of the port can be done in an arbitrary way but avoiding any possible starvation.

The function receive returns both the received message and the index of the port of the array pa, used to receive the message.

3.2 Implementation

PortArray<T> provides a contructor and the methods: checkPorts(int v[], int n), send(Message<T> m, int p), receive(int v[], int n).
Its private data are:

- *PortArray:* that is an array of generic SynchPort;
- *ports_available*: it is a queue for recording all ports that contains data;
- *available:* it's a FairSem useful for blocking a thread if no data is available in any port;
- *dim:* it's the dimension of the array of port;

3.2.1 PortArray(int n, int Nsenders)

It's the constructor of the class.

It initializes the class with an array of n synchronous port, where each port can handle Nsenders.
Besides it initializes available with 0.

3.2.2 send(Message<T> m, int p)

It signals on the available semaphore to notify that there's something available on the PortArray then sends the message to the SynchPort of the PortArray with index p.

3.2.3 receive(int v[], int n)

It returns a Message object.

First, it's done a P operation on the available semaphore. So the caller thread can go on only if there's something on the Ports. Then in a loop it's checked using the indexes contained in v (n is the dimension of v) if there's a SynchPort with data, otherwise it's done a P for blocking.

If there's any SynchPort, then it's done a receive operation to get the data

3.2.4 checkPorts(int v[], int n)

It return the index of an available port with index contained in v.

In order to guarantee that he choice of port avoids any starvation, it's implemented a queue (PortQueue) that provides these methods:

1. *remove*: remove an element from the front (FIFO fashion);
2. *insert*: insert an element in the tail of the queue;
3. *remove_x:* remove a particular element in the queue;
4. *mode_extraction(int v[], int n)*: if the first element is contained in the vector v return true, otherwise false;
5. *find(int v[], int n)*: checks if an element in the queue is equal to any element of v and returns the index of the element in the queue, otherwise -1;

When it's called a receive and the PortQueue ports_available is empty, are checked every ports in PortArray (accessed by indexes in v) and inserted (except the first that will be used) in the PortQueue.

Otherwise if ports_available is not empty, it's removed the first element. But, it could be possible that the receiver could change the set of ports used in two consecutive receive. Thus, if the first element of ports_available is equal to any element of v, then it's removed the first element. Otherwise it's checked if any element of v is equal to any element in ports_available and then removed, if it occurs. Otherwise it's checked in every ports in PortArray (accessed by indexes in v).

3.3 Test

The test is done with a communication between a receiver that uses a PortArray of 4 elements and two sender that sends their data on two different sets of SynchPorts in PortArray.

The operations are interleaved by sleeping time in order to guarantee that:

- Sender2 forwards first data on the second set of SynchPort;
- Receiver receives in the first set after the send operation of Sender2 in order to test if it is blocked until arrives data on the first set;
- After that Sender1 forwards two data on the first set;
- Receiver after receiving the first data, receives on the second set in order to test that even if

there's a port with data, it's served the port in the actual set. The it receives on the first set. The correctness of the test is by the order of the output.

4 Mailbox

This is Part4 of the assignment. It will be shown all the techniques and test adopted.

4.1 Assignment Details

Implement the class mailbox that provides a buffer, to be organized as a circular array of four slots of type integer. An object of this class can be conceived as a server process that provides two services. The first service is offered to the 10 producer threads. This first service is: to receive a value, sent by a producer, and insert this value into the buffer, when at least an empty slot is available in the buffer. The second service is offered to the consumer thread. This second service is: to remove a value from the buffer and send this value to the consumer thread, when at least a full slot is available in the buffer.

A) To test the implementation of the mailbox, implement a producer thread that executes a cycle of 5 iterations. During each iteration the producer sends an integer value to the mailbox. So, the consumer must execute 50 iterations to receive all the values sent by any producer. During each iteration, the consumer receives a value and prints this value on the screen together with the identifier of the sending thread.

B) Modify the previous solution in order to allow the server process mailbox to offer the service to the producers according to their priorities. For this purpose, assign to each producer thread a unique priority value specified as an integer value p ($1 \leq p \leq 10$). When the buffer is full, some producer threads can be blocked waiting for an empty slot of the buffer. In this case, when a value contained in the buffer is removed, the process mailbox must receive a new value from the blocked producer with the higher priority.

4.2 MailboxA

An object of this class implements a server process that handles a buffer organized as a circular array of Message_T (which is a class with two fields: an integer and the tid of the sender). It provides two service: one for inserting and another one for removing a value into the buffer. Before an operation, consumer and producers must send a request to the mailbox specifying the type of the operation requested. This guarantees to avoid that a ready process (consumer or producer) should wait because the mailbox is blocked in an operation because the other front of the communication is not ready.

The data of the producers and the consumer is on different ports because a SynchPort must be declared in the receiver process.

4.2.1 Implementation

Its private data are:

- *countExtractions:* it' the number of the removing from the mailbox (useful to know when it has to end the run);
- *pendingProducers:* it's the number of producers that are ready and waiting for an insert

(waiting because the buffer is full);

- *mq:* it's the buffer;
- *ready:* it's the SynchPort where to send a request for a specific operation (insert or remove);
- *dataProducers:* SynchPort where the data is exchanged with producers;
- *dataConsumer*: SynchPort where the data is exchanged with the consumer. It's private because the port is declared as a public field in the receiver (consumer). It's useful in order to separate the code from the test application;

This class extends the class Thread and overrides the method run. In the method are privileged pending requests of the producers. They are served with the respect to the buffer constraints (until the buffer is full).

If there are no pending requests, the mailbox waits for a request of service on ready. Then, if it is a request of inserting, receives on dataProducers and inserts into the buffer. Otherwise removes a data from the mailbox and sends to the consumer.
If the buffer is empty and the request of operation is of removing, the mailbox waits for a receive operation and after that sends it to the consumer. If the buffer is full and the request of operation is to insert, receives on ready until the request is of removing (increasing pendingProducers that they will be served when it's possible).

Test

In the test are implemented 10 producers thread that executes a cycle of 5 iterations. During each iteration the producer sends an integer value to the mailbox. So, the consumer must execute 50 iterations to receive all the values sent by any producer. During each iteration, the consumer receives a value and prints this value on the screen together with the identifier of the sending thread. By the output is possible to see that all the data is received by the consumer.

Running

```
> make Part4

> java TestPortWithMailboxA
```

4.3 MailboxB

This class modifies MailboxA in order to guarantee that waiting producers are served with the respect to their priorities. In order to wake-up a specific producer, after the request the producer must receive an ACK from the mailbox and then send data. The port where to receive the ACK is inserted in the message of request (precisely in the response field).

4.2.1 Implementation

This class is similar to MailboxA except for providing a list of blocked threads. When a producer must be blocked, it's inserted an element Blocked (that contains the tid, the priority and the port for the ACK) in this list.

When it's served a pending request, it's removed the Blocked element with highest priority from the list. Then it's sended an ACK and after that, received the data that is inserted in the buffer.

Test

The test is the same of partA, but now are printed blocked threads and the awakened one. So it's possible to see by the output that always the thread blocked with highest priority is served first among the other blocked threads.

Running

```
> make Part4
```

```
> java TestPortWithMailboxB
```