Concurrent and Distribute Systems

Course Project Report

By - Regassa, Yeneakal Girma

Matricola: 539984

Email: yeneakalgirma28@gmail.com

Feb 9, 2016

Content

1. Introduction

This Concurrent and Distributed Systems assignment report consists of four parts. The first part is implementation of a FIFO semaphore by using only the java low level synchronization feature. The second part is simulation of a generic synchronous port SynchPort<T> used in message passing model by using the FIFO semaphore implemented in the first part. The third part is to implement a class PortArray<T> that defines an array of generic synchronous ports. The Fourth part has two sections. Implementing a communication mechanism among 10 producer threads and a single consumer thread using Mailbox server and implementing Mailbox server to receives value from the producer based on priority.

2. Part one- FIFO Semaphore

The first part is the implementation of a FIFO semaphore. In order to Implement this class, I used only the java low-level synchronization features (synchronized blocks, synchronized methods and the Object's methods *wait(), notify()* and *notifyAll().*

The implementation of the class *FairSem* guarantees that threads waiting to acquire a lock are awakened in a FIFO order.

First I implemented a class called *ThreadList* to handle list of threads by providing a linked list<long> to store threads(specifically identified by their thread Id).

ThreadList
List<Long> queue = new LinkedList<>();
synchronized void addToQueue(longTid) synchronized long removeFromQueue() synchronized long getFirst() synchronized boolean empty()

This class have four synchronized methods:

- *void addToQueue(long Tid)* :- which accepts Thread ID as a parameter and stores in the linked list.
- *long removeFromQueue()* :- which removes and returns the id of the thread stored at index 0 of the linked list if the linked list is not empty.
- *long getFirst():-* returns the id of the thread at index 0 of the linked list if the linked list is not empty.
- *boolean empty():-* returns Boolean value by checking whether the linked list is empty or not.

FairSem class contains : -

FairSem
ThreadList waitList; ThreadList awakenedList; int value; boolean enableDemo;
FairSem(int value) FairSem(int value_, boolean demo) synchronized void P( ) synchronized void V( )

- *waitList* is a list of type *ThreadList* which is used to store a list of waiting threads which are blocked on red semaphore waiting for the free lock and *awakenedList* is a list of type *ThreadList* which is used to store a list of awakened threads due to some V operation performed by some other thread.
- An int type variable *value* used to store the values of the semaphore and boolean type *enableDemo* is used to enable and display some notifications during demonstration.
- The P() method checks the value of *FairSem*. If *Fairsem* is green it completes P operation by decrementing the *Fairsem* value. If Fairsem is red semaphore it will be inserted into the *waitList* and waits until it become green semaphore. Then when it awakens from wait state

it tests weather the thread currently running is the thread at index zero in the *awakenedList* or not to assure the FIFO order, if it is true it continues and removed from the awakened list and completes its P operation otherwise it will be blocked again until it will awaken in the order of insertion into the waiting list. I used while loop to prevent spurious wake up.

- The V() method works using passing the condition technique first it checks whether the *waitList* is empty or not and if the *waitList* is not empty it adds the first element in the *waitList* to awakened list and calls notifyAll to wakeup all blocked threads by implicitly passing the condition to the threads awakened. Otherwise if there is no any thread in the *waitlist* it increments the variable value and notifies all blocked threads in other condition using *notifyAll()* method.

Demonstration:

The demonstration shown below is to show how threads acquire the semaphore in a FIFO manner. This is a screen shot of the output in which we can see the lock handler, blocked threads.

FairSem
Thread-0 begins P Operation
Thread-0 ends P Operation
Thread-0: is running
Thread-2 begins P Operation
Thread blocked: Thread-2
Thread-1 begins P Operation
Thread blocked: Thread-1
Thread-0 executed V Operation
Thread-2 ends P Operation
Thread-2: is running
Thread-2 executed V Operation
Thread-1 ends P Operation
Thread-1: is running
Thread-1 executed V Operation
Thread Completed Operation on Semaphore sem

3. Part Two- Generic Synchronous Port

The generic synchronous port is simulated by the class SynchPort<T>. I used the FairSem class to synchronize the sending and the receive operation.

- *buffer* :- is a field that represents a single slot storage for the message sent by the sender to the port.
- *semSend* :- is a mutual exclusion semaphore for the senders.
- *semRecv* :- is an event semaphore for the receiver.
- *semEvent* :- is an event semaphore that is used for rendezvous to occur between the sender and receiver.

SynchPort<T>
Message<T> buffer; FairSem semSend = new FairSem(1); FairSem semRecv = new FairSem(0); FairSem semEvent = new FairSem(0);; boolean enableDemo;
SynchPort() SynchPort(boolean demo) void send(Message<T> mes) Message<T> receive()

Message<T>
T data; int index; String ThreadName; int priority; SynchPort<T> reply;
Message() Message(T _content) Message(T _content, SynchPort<T> SenderID) T getContent() void setContent(T _content) int getIndex() void setIndex(int index) String getThreadName() void setThreadName(String ThreadName) int getPriority() void setPriority(int priority) SynchPort<T> getReplyport() void setReplyport(SynchPort<T> SenderID)

I implemented also a class called Message<T> that contains five fields:

- *data*:- is a value of type T, which represents the information sent by the sending process.
- *index*:- is a field to store the index of the specific port used in third part of the project.

- *ThreadName*:- is a field used to store the names of the executing thread used in part 4A and 4B of the project.
- *Priority*:- is a field used to store the priority of a thread used in part 4B of the project.
- *reply*:-which allows the receiving process, when it is needed, to return a response message to the sending process.
- In addition each field have getter and setter to access and put their value respectively.

The SynchPort class contains to method:

- Send(Message<T> m) is a method which is used to send message m to the port.
- Message<T> m Receive() is a method to receive a message from a port.

The method *send(message<T> m)* operates as follows first it locks the semaphore *semSend* then puts the message m to the *buffer* type message<T> then signals *semRecv* that was previously red semaphore. Then it will be blocked on an event semaphore *semEvent* waiting for rendezvous to occur with the receiver and finally when both the sender and the receiver are at communication point it unlocks *semSend*.

The method *Message<T> receive()* operates as follows when receiver process calls receive operation it was blocked on semaphore *semRecv* if the semaphore is red otherwise it copies the content of the *buffer* to local variable m of type message<T> then signals *semEvent* to inform the receiver is at communication point and finally returns m.

Demonstration:

To demonstrate this part, there are senders which sends a certain number of messages to the receiver. *Thread.sleep()* is used to create a condition in which the sender doesn't need to wait because the receiver reads the message instantly. The classes used to demonstrate are:

- SynchPort
- SynchPortDemo
- PortReceiverDemo
- PortSenderDemo
- Message
- FairSem

4. Part three - Array of synchronous port(PortArray)

The *PortArray* class defines an array of generic synchronous port. To implement this class I used the class *SynchPort<T>* and *Message<T>*. I used also the *FairSem* class to synchronize the sending and the receive operation.

PortArray<T>
int port_num; boolean[] occupied; List<SynchPort<T>> ports = new ArrayList<SynchPort<T>>(); FairSem mutex = new FairSem(1); FairSem semAvailable = new FairSem(0); boolean enableDemo;

```
PortArray(int n, boolean demo)
 void send(Message<T> m, int n)
Message<T> receive(int[] v, int n)
```

The *PortArray* class contains:-

- *port_num* :- which holds the dimension of Array List of type SynchPort.
- *occupied* :- is a boolean type array with size equal to the dimension of the PortArray which used to holds the state of each SynchPort in the port array whether the port contains a message or not.
- *ports* :- is a field which represents an Array list of type SynchPort.
- *mutex* :- is a semaphore which is used to guarante mutual exclusion between client and server.
- *semAvailable*:- is a an event semaphore used by client to notify the arrival of message on some port and used by the server to wait until message is arrived on one of the ports.
- *enable demo*:- is used to enable and display some notifications during demonstration.

In addition I implemented a class *PortNotAvailable* class that extends Exception to be thrown when the port number specified by the client or the server is out of the Port Array bound.

The method *void send(Message<T> m, int n)* operates as follows first it acquire the mutex and update the state of port n by making (*occupied[n] = true)* then signals the receiver that tells some message is arrived in one of the ports in the PortArray using *semAvailable* then release the mutex and sends message m to port n.

The method *Message<T> receive(int[] v, int n)* implemented as follows I declared three local variables :-

- *int randIndex* :- is used to hold some randomly chosen port from the list of SynchPort arrays.
- *int j* :- is used to hold port index of the chosen port each time throughout the process of searching a port that have pending message on it.
- *boolean available* :- is used as a flag to make sure some message is found in one of the selected list of port by receive to receive. It was initialized as false.

First it checks whether the dimension of the selected port by receiver is in the range of dimension of array of ports if it is not throws *PortNotAvailable* exception. Then the receiver acquires the mutex and checks the flag *available* if it is false it selects random index, the choice of the port can be done in an arbitrary way using *ThreadLocalRandom.current().nextInt(0, port_num)* but avoiding any possible starvation then compares the randomly selected port index with each port index of selected port list by receiver and the state of the port that shows some message is pending in that port. If one of the condition is false by increasing j by one until it reaches the first randomly selected port again it continues iterating through all selected port indexes of the receiver. And if there is no any pending message in all port of the selected ports the receiver releases the mutex and blocked on event semaphore *semAvailable* waiting for message arrival in one of its port and when it gets signalled it repeats the same process again until it gets pending message in one of its port. Finally if the condition matches it updates the state of that port, receives the message on that specific port

furthermore it appends the port index that a message was received in the receiving message index field and returns the message.

Demonstration:

To demonstrate this part, there are senders which sends a certain messages to one of the Port arrays. Thread.sleep() is used to simulate message production. I used also a server process that have two sets of selected ports each contains five indexes. The first set of index is to receive from ports that have odd number of indexes and the second set contains ports that have even number of index.The classes used to demonstrate are:

- SynchPort
- PortArray
- PortArrayDemo
- PortNotAvailable
- Message
- FairSem

5. Part Four - Producer Consumer Application

In this part of the project it is aimed to implement an application that provides communication mechanism among 10 producer and a single consumer threads by using the classes implemented in the previous part of the project.

First I implemented two classes the so called MessageQueue and TnameQueue that provides a circular array of four slots that are used to store the message inserted by the producer and the name of the producer that inserted the message respectively.

MessageQueue	TnameQueue
int[] buffer; int front; int rear; int count;	String[] buffer; int front; int rear; int count;
MessageQueue() void insert(int val) int remove() boolean empty() boolean full()	TnameQueue() void insert(String name) String remove() boolean empty() boolean full()

Each classes have Four methods:

- *insert*:- is used to insert a value into the buffers corresponding to the type that a buffer holds.
- *remove*:-is used to remove and return a value corresponding to the type that a buffer holds from the buffer.
- *empty* :- returns Boolean type by checking whether count equal to 0 or not.
- *full*:- returns Boolean type by checking whether count equal to 4 or not.

Section A

For this section I implemented the class MailboxA as a server that have two ports.

MailboxA
SynchPort<Integer> request; SynchPort<Integer> insert_prod; int waiting_prod; boolean waiting_cons; MessageQueue buffer; TnameQueue tname_queue; Message<Integer> msg_in, msg_out; int value; String prod_tname;
MailboxA() void run()

Port *request* :- is used to receive the type of service requested from the clients.

- if value received from port *request* is 0 it means a producer is wanted to insert a message into the mailbox buffer.
- if value received from port *request* is 1 it means a consumer is wanted to remove a message from the mailbox buffer.

Port *insert_prod*:- is used to receive a message sent from the producer if there is an empty slot in the buffer.

The run() method of the MailboxA is implemented as a server which execute an endless loop. The operation performed in the while loop is as follows:

The server accepts a message sent to its *request* port then using switch statement it checks the content of the message with the switch case.

- **case 0** :- if the requested service is 0 it means a producer wants to insert a value into the buffer so it checks whether the buffer is full or not. If the buffer is full it increments the waiting producer(*waiting_prod++)* else it accepts the message pending at its *insert_prod* port and put to *msg_in*. Then it insert the data of the message into queue *buffer* and the name of the sending producer into queue *tname_queue* respectively. Then if there is waiting consumer it removes a value and the name of the producer that inserts that data from the front of both queues and put to *msg_out.* Then changes the status of the consumer to *(waiting_cons = false)* and sends *msg_out* to consumers *in* port.
- **case 1** :- if the requested service is 1 it means a consumer wants to remove a value from the buffer so it checks whether the buffer is empty or not. if the buffer is empty it changes the status of the consumer *(waiting_cons = true)* else it removes a value and the name of the producer that inserts that data from the front of both queues and put to *msg_out.* Then changes the status of the consumer to *(waiting_cons = false)* and sends *msg_out* to consumers *in* port. Then if there is waiting producer since there is an empty slot now it decerments the waiting producer(*waiting_prod--) and* accepts the message pending at its *insert_prod* port and put to *msg_in*. Then it insert the data of the message into queue *buffer* and the name of the sending producer into queue *tname_queue* respectively.

Demonstration:

To test the MailboxA I implemented a class ProducerA and ConsumerA.

ProducerA thread executes a cycle of 5 iterations. During each iteration the producer sends a request to the MailboxA port *request* by putting 0 to the message then it sends an the real message value to the MailboxA port *insert_prod*.

ConsumerA thread have a SynchPort *in* which is used to receive a message sent from the MailboxA. The consumer performs 50 iterations to receive all the values sent by any producer. During each iteration, the consumer sends a request to the MailboxA port request by putting 1 to the message. Then the consumer receives a message and prints the received value on the screen together with the identifier of the sending thread.

Section B

For this section I implemented the class MailboxB as a server that have two ports the same as section A.

MailboxB
SynchPort<Integer> request; SynchPort<Integer> insert_prod; List<SynchPort<Integer>> prod_reply = new ArrayList<SynchPort<Integer>>(); boolean[] blocked = new boolean[10]; boolean enableDemo; String[] prod_name_list; int waiting_prod; boolean waiting_cons; MessageQueue buffer; TnameQueue tname_queue; Message<Integer> msg_in, msg_out; int value; String prod_name;
MailboxB() MailboxB(boolean t, String[] tname) void run()

Most of the operation performed by MailboxB is the same as MailboxA described in section A.

The additional things implemented in MailboxB that gives the ability to awaken the blocked producers based on their priority are described as follows:

- *prod_reply* :- is an array list of type SynchPort which is used to store the reply port of each blocked producer waiting to insert a message into the mailbox.
- *blocked[]* :- is a boolean type array of dimension 10 which is used to store the status of each producer whether it is blocked or not.
- *prod_name_list[]* : is a string type array used to store set of producer that have blocked status true.

The run() method of the MailboxB is implemented as a server which execute an endless loop. The operation performed in the while loop is as follows:

The server accepts a message sent to its *request* port then using switch statement it checks the content of the message with the switch case.

- **case 0** :- if the requested service is 0 it means a producer wants to insert a value into the buffer so it checks whether the buffer is full or not. If the buffer is full it increments the waiting producer(*waiting_prod++)* ,inserts the reply port of the blocked producer at index of (priority-1) of the array list *prod_reply* and modifies its blocking status true *(blocked[priority-1] = true)* else it accepts the message pending at its *insert_prod* port and put to *msg_in*. Then it insert the data of the message into queue *buffer* and the name of the sending producer into queue *tname_queue* respectively. Then if there is waiting consumer it removes a value and the name of the producer that inserts that data from the front of both queues and put to *msg_out.* Then changes the status of the consumer to *(waiting_cons = false)* and sends *msg_out* to consumers *in* port.
- **case 1** :- if the requested service is 1 it means a consumer wants to remove a value from the buffer so it checks whether the buffer is empty or not. if the buffer is empty it changes the status of the consumer *(waiting_cons = true)* else it removes a value and the name of the producer that inserts that data from the front of both queues and put to *msg_out.* Then changes the status of the consumer to *(waiting_cons = false)* and sends *msg_out* to consumers *in* port. Then if there is waiting producer since there is an empty slot we have to awaken one of the blocked producer. But now the producer to be awakened is selected based on their priority. In this program producer with smallest integer number have highest priority (p0 > p1 >p2 >.....p8>p9) so starting from index 0 it checks the blocking status of each producer then when it gets (blocked[i] = true) it decrements the waiting producer (*waiting_prod--),* updates the status *(blocked[i]=false)* and gets its reply port from the port_reply array list and sends an empty signalling message then it accepts the message pending at its *insert_prod* port and put to *msg_in*. Then it insert the data of the message into queue *buffer* and the name of the sending producer into queue *tname_queue* respectively.

Demonstration:

To test the MailboxB I implemented a class ProducerB and ConsumerB.

ProducerB thread have a unique priority value specified as an integer value p ($1 \leq p \leq 10$) and SynchPort *in* which is used to receive a signalling message sent from the MailboxB. A producer executes a cycle of 5 iterations. During each iteration the producer sends a request to the MailboxB port *request*. The message contains requested service type(0), producer thread name, priority and reply port of the service requesting producer thread. Then waits for signalling message from mailbox server and when it gets the signalling message it sends the real message to the MailboxB port *insert_prod*.

ConsumerB thread have a SynchPort *in* which is used to receive a message sent from the MailboxB. The consumer performs 50 iterations to receive all the values sent by any producer. During each iteration, the consumer sends a service request to the MailboxB port *request* by putting 1 to the message. Then the consumer receives a message and prints the received value on the screen together with the identifier of the sending thread.