Chalmers | GÖTEBORGS UNIVERSITET
Josef Svenningsson, Computer Science and Engineering

Concurrent Programming TDA381/DIT390

Wednesday, August 18, 8.30 – 12.30, M.

(including example solutions to programming problems)

Josef Svenningsson, tel. 070 5455542

- Grading scale (Betygsgränser):

  Chalmers: 3 = 20–29 points, 4 = 30–39 points , 5 = 40–50 points
  Chalmers ETCS: E = 20–23, D = 24–29, C = 30–37, B = 38–43, A = 44–50
  GU: Godkänd 20–39 points, Väl godkänd 40–50 points

  Total points on the exam: 50

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):**
  - Dictionary (Ordlista/ordbok)
- **Notes:**
  - Read through the paper first and plan your time.
  - Answer in either Swedish or English.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each of the questions on a new page.
  - The exact syntax of each programming language you are going to use is not so important as long as the graders can understand the intended meaning. If you are unsure just put in an explanation of your notation.

Question 1. What does it mean for a library to be thread-safe? What can cause a library to not be thread-safe? *(2p)*

Question 2. Before an aircraft is allowed to be used in regular traffic the manufacturer has to show that it can be evacuated within 90 seconds in case of an emergency. This can be quite a difficult thing to achieve for large aircrafts such as the Airbus A380 which can fit up to 850 passengers.

Your task is to simulate the evacuation of an Airbus A380 but you don't have to think about the timing aspects. You should have 850 passengers trying to evacuate the aircraft. There are 16 exits and each passenger has a closest exit which he or she will use to get out through. Passengers are assigned to their exits with the formula $exitNo = passengerNo * 16/850$.

A crux is that all passengers cannot fit though the exit at once, at most two passengers can leave through an exit at a time. Your program should keep track of the number of passengers which have evacuated the plane and print a message when all of them are out of the aircraft.

You should write your implementation using either Java or JR and you must only use semaphores to solve the synchronization problems. *(10p)*

```
import java.util.concurrent.*;

class Airbus {

    Semaphore doors[];
    Semaphore mutex;
    int counter;

    public static void main(String[] args) {
        new Airbus();
    }

    public Airbus() {
        doors = new Semaphore[16];
        for(int i=0;i<doors.length;i++) {
          doors[i] = new Semaphore(2,true);
        }

        mutex = new Semaphore(1);

        counter = 0;

        for(int i=0;i<850;i++) {
          (new Passenger(doors[i*16/850], i)).run();
        }
    }

    class Passenger extends Thread {
        Semaphore door;
        int id;
        public Passenger(Semaphore door, int id) {
            this.door = door;
            this.id   = id;
        }

        public void run() {
            door.acquireUninterruptibly();
            door.release();
            mutex.acquireUninterruptibly();
            counter = counter + 1;
            mutex.release();
            if(counter==850)
              System.out.println("Simulation␣done");

        }
    }

}
```

Question 3. In this question your task is to implement a class for multicast channels.

Introduction

A multicast channel provides communication from one sending port to several independent receiving ports. When a sender sends a message to a multicast channel, it is buffered internally, and later received exactly once at each receiving port. A sender atomically appends a message to the end of the channel and continues its execution. Multiple senders are allowed. Each receiver owns its own receiving port to the multicast channel. This way each receiver behaves independently and receives messages at its own pace. A receiver can potentially block if it has read all the messages and it is waiting for a new message to arrive.

The intended interface for multicast channels is as follows:

```
public interface Port<T> {
    public T receive() throws InterruptedException;
}

public interface MChan<T> {
    public void send(T item) throws InterruptedException;

    public Port<T> newPort();
}
```

- send – a sender calls this method to atomically append a message to the end of the channel;
- newPort – a receiver must first call this method to obtain its own receiving port, which initially points to the current end of the channel;
- receive – a receiver calls this method to receive one message from the channel; this method blocks when no message is currently available.

To illustrate the workings of a multicast channel and especially the independent receiving ports consider this simple sequential example, which prints "`P2<-66, P1<-42, P1<-66.`" and then deadlocks.

```
MChan<Integer> mc = new MyMChan<Integer>();
Port<Integer> p1 = mc.newPort();
mc.send(42);
Port<Integer> p2 = mc.newPort();
mc.send(66);
System.out. print ("P2<-"+p2.receive()+", ");
System.out. print ("P1<-"+p1.receive()+", ");
System.out. print ("P1<-"+p1.receive()+".");
System.out. print ("P1<-"+p1.receive()+"?");
```

The Problem

In this question you are asked to implement the multicast channel using Java 5 monitors. To get full points your solution must meet the following requirements:

- The channel has unbounded size. A sender can always send a new message and is never blocked. However, you need to make sure that you only use memory for messages that have a waiting receiver. (No memory leaks if you want full points.)
- Receivers receiving different messages must be allowed to proceed in parallel. For example if process P1 is trying to receive the first message in the channel while a different process P2 is trying to receive the second message in the channel at the same time, both processes must be allowed to receive their respective messages at the same time. (hint: think about the readers/writers problem with a twist)

(10p)

This solution maintains a queue for each port which is perhaps not very nice. But it makes the solution a lot simpler conceptually.

```
public interface Port<T> {
    public T receive () throws InterruptedException ;
}

public interface MChan<T> {
    private Set<MultiPort<T>> ports;
    private Lock setLock;

    public MChan<T>() {
        ports = new HashSet<MultiPort<T>>();
        setLock = new ReentrantLock();
    }

    public void send(T item) throws InterruptedException {
        setLock. lock ();
        try {
            for(MultiPort<T> p : ports) {
```

```
            p.send(item);
        }
    } finally {
        setLock.unlock();
    }
}

public Port<T> newPort() {
    MultiPort<T> port = new MultiPort<T>();
    setLock.lock();
    try {
        ports.add(port);
    } finally {
        setLock.unlock();
    }
    return port;
}

private class MultiPort<T> implements Port<T> {
    private Queue<T> queue;

    private Lock lock;
    private Condition cond;

    public MultiPort<T>() {
        queue = new ArrayDequeue<T>();
        lock  = new ReentrantLock();
        cond  = lock.newCondition();
    }

    public send(T item) throws InterruptedException {
        lock.lockInterruptibly();
        try {
            queue.add(item);
            cond.signal();
        } finally {
            lock.unlock();
        }
    }

    public T receive() throws InterruptedException {
        lock.lockInterruptibly();
        T item;
        try {
            while (queue.size == 0) {
                cond.await();
            }
```

```
                item = queue.remove();
            } finally {
                lock.unlock();
            }
            return item;
        }
    }

}
```

Question 4. In this question you should implement a library for an unbounded buffer in Erlang. It should have the following interface:

```
-module(buffer).
-export([new/0, insert/2, retrieve/1]).
```

The library should work as follows. The function new creates a new unbounded buffer and returns it (or some reference to it). insert takes two arguments, a buffer and the item to put into the buffer. retrieve takes a buffer as an argument and returns the oldest element in that buffer and at the same time removes it from the buffer. As usual, if any of these operations cannot be completed because some condition was not fulfilled then the thread should block until that condition is fulfilled. *(8p)*

```
-module(buffer).
-export([new/0, insert/2, retrieve/1]).

new -> spawn(fun() -> bServer([],[]) end).

 insert (Server,Item) ->
     Server!f insert ,Itemg.

 retrieve (Server) ->
     Server!f retrieve , self ()g,
     receive
         f gotten ,Itemg -> Item
     end.

bServer(Pending,Buffer) ->
     receive
         f insert , Itemg ->
             case Pending of
                 [] -> bServer ([], Buffer ++ [Item]);
                 [Pid|Pend] ->
                     Pid!f gotten ,Itemg,
                     bServer(Pend,Buffer);
             end;
         f retrieve ,Pidg ->
             case Buffer of
```

```
                [] -> bServer (Pending ++ [Pid], []);
                [Item|Buff] ->
                    Pid!f gotten ,Itemg,
                    bServer(Pending,Buff);
            end;
    end.
```

A simpler server

```
bServer(Buffer) ->
    receive
        f insert , Itemg ->
            bServer(Buffer ++ [Item];
        f retrieve , Pidg ->
            case Buffer of
                [] -> receive f insert , Itemg ->
                    Pid!Item,
                    bServer(Buffer);
                [Item|Buff]->
                    Pid!f gotten ,Itemg,
                    bServer(Buff);
            end;
    end.
```

Even simpler server

```
bServer(Buffer) ->
    receive
        f insert , Itemg ->
            bServer(Buffer ++ [Item]);
        f retrieve , Pidg when Buffer /= [] ->
            [Item|Buff] = Buffer,
            Pid!Item,
            bServer(Buff);
    end.
```

Simplest?

```
bServer([]) ->
    receive
        f insert , Itemg -> bServer([Item]);
bServer([Item|Buffer]) ->
    receive
        f insert , NewItemg ->
            bServer([Item|Buffer] ++ [NewItem]);
        f retrieve , Pidg ->
            Pid!Item,
            bServer(Buffer);
    end.
```

Question 5. In this assignment you will use a synchronization primitive called Conditional Critical Regions, or CCR for short. For the purpose of this assignment we are going to assume that Java is equipped with CCR as it's only synchronization primitive. Therefore you may use any Java syntax and its libraries freely except for when it comes to synchronization.

The syntax of CCRs look as follows:

```
atomic (b) {
   ...
}
```

A CCR is a block of code that begins with the keyword **atomic**. After the **atomic** keyword comes a boolean condition which is written b in this example. The boolean condition can be left out if it is always true. After the boolean condition comes the code which should be executed inside curly braces. We will refer to these code blocks as atomic blocks.

Semantically CCRs work as follows: Whenever a thread executes inside an atomic block one can safely assume that no other thread is executing in any other atomic block. In other words, atomic blocks provide mutual exclusion. If there is a thread executing in an atomic block and another thread tries to enter another atomic block this other thread will block until the first one has exited its atomic block.

When a thread is trying to enter an atomic block and no other threads are executing inside an atomic block, the first thing that happens is that the boolean condition is evaluated. If it evaluates to true then the thread gets the mutex and enters the atomic block. If, however, the boolean expression evaluates to false the thread is blocked and will resume some time in the future when the condition becomes true. Therefore, the boolean expression in the atomic block provides a means for conditional synchronization. This completes the description of CCRs.

This assignment is in two parts, and you should use CCRs as your only means of synchronization on both of them.

In this assignment you should implement a solution to the readers/writers problem. Your solution should implement an object with the following interface which encapsulates the entry and exit protocol for the readers/writers problem.

```
class RW {
  public void start_read ()
  public void end_read ()
  public void start_write ()
  public void end_write ()
}
```

The readers/writers problem is described as follows. There are two types of processes which have access to some shared resource, such as a database. The readers only read the data base and therefore several readers are allowed into the database at the same time. Writers modify the database and must establish mutual exclusion before gaining access to the database. *(10p)*

```
class RW {
  int nr = 0, nw = 0;
  public void start_read () {
    atomic (nw == 0) {
```

```
                nr++;
            }
        }
        public void end_read() {
            atomic { nr−−; }
        }
        public void start_write () {
            atomic (nr == 0 && nw == 0) {
                nw++;
            }
        }
        public void end_write() {
            atomic { nw−−; }
        }
    }
```

Question 6. You are given the following library which implements software transactional memory.

```
class TransactionManager {
    public TransactionManager();
    public void beginTransaction ();
    public void endTransaction ();
    public TransactionVariable <A> newTransactionVariable();
    public void retry ();
}

class TransactionVariable <A> {
    public A read ();
    public void write (A value );
}
```

Here is a brief explanation of how the library works: A TransactionManager object handles all the low level details about how transactions are executed. Whenever you want to initiate a transaction call the method beginTransaction and end the transaction with a call to endTransaction. Shared variables which are handled inside a transaction must be of the type TransactionVariable <A> where A is the type of the data contained in the variable. The content of the variable is accessed and manipulated by the methods read and write. The method retry is used for conditional synchronization. When retry is called the transaction is aborted and restarted some time in the future.

Using this library as your only means of synchronization you should implement a class for barrier synchronization with the following interface:

```
class Barrier {
    public Barrier (int n);
    public void synch();
}
```

The argument to the constructor says how many processes are needed at the barrier before they are released. A call to synch blocks until n processes have arrived at the barrier.

Your implementation only needs to handle a single barrier, meaning that a barrier object need not work any more after the first barrier is reached. *(10p)*

Solution

```
class Barrier {
  int n;
  private final TransactionManager tm = new TransactionManager();
  private final TransactionVariable <Integer> arrived =
    tm.newTransactionVariable (0);

  public Barrier (int n) {
    this.n = n;
  }
  public void synch() {
    tm.beginTransaction ();
    arrived.write(arrived.read()+1);
    tm.endTransaction ();
    tm.beginTransaction ();
    if (arrived.read() < n)
      tm.retry ();
    tm.endTransaction ();
  }
}
```