Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Concurrent Programming TDA382

Thursday, March 14, 14:00 PM.

(including example solutions to programming problems)

Alejandro Russo, tel. 0705 110896

- The maximum amount of points you can score on the exam: 68 points. To pass the course, you need to pass each lab, and get at least 24 points on the exam. Further requirements for grades (Betygsgränser) are as follows, for total points on exam + labs:

  Chalmers: grade 3: 40 - 59 points, grade 4: 60 - 79 points, grade 5: 80 - 100 points.
  Chalmers ETCS: E = 40–47, D = 48–59, C = 60–75, B = 76–87, A = 88-100
  GU: Godkänd 45-79 points, Väl godkänd 80-100 points

- 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. Preferably in English, some assistants might not read Swedish yet.
  - 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.
  - Points will be deducted for solutions which are unnecessarily complicated.
  - As a recommendation, consider spending around 45 minutes per exercise. However, this is only a recommendation.
  - To see your exam:
    * April 9th 2013, 12:15-13:00, room 5128 (5th floor), EDIT building.
    * April 12th 2013, 12:15-13:00, room 5128 (5th floor), EDIT building.

Question 1. The Problem We consider a barbershop in Sicily, Italy. The shop has three barbers which work independently from each other. Every barber has a fixed barber chair where he cuts customers' hair. Since it is a hot country, a barber sits away from his barber chair in order to sleep (little siesta) when there is no customer sitting there. As soon as a customer sits on the chair, he/she presses a button to wake only the barber working on that chair. Once the barber finishes with the haircut, he returns to his spot to rest (possible more siesta) and the client then leaves. In order to avoid injuries, it is important that the client does not leave the chair until the barber finishes doing the haircut. To simplify the scenario, we assume that there is a fixed number $N > 3$ of clients waiting to cut their hair and each client decides to get a service from a specific barber before getting into the shop. This problem is a variation of the Sleeping Barber Problem, which was first proposed by Edsger W. Dijkstra in 1968.

Your assignment Your task is to implement a simulation of the barbershop. The simulation should capture the synchronization between clients and barbers. For instance, a client must wait until a chair is free, a client must wait until his/her haircut is completed before leaving, a barber sleeps if no client sits on his chair, a client needs to press a button to get a haircut from the barber.

(12p)

To get full points your solution must fulfill the following criteria:

- You can use either Java or JR. If you choose to use JR, remember that the primitive **sem** is for declaring semaphores (e.g. **sem s**), the primitive **P(s)** for acquiring semaphore *s*, and **V(s)** for signaling semaphore **s**.
- You must use semaphores for synchronization or mutual exclusion. No other synchronization constructs are allowed.
- Every barber in the simulation must execute the same code. Similarly, every client must also execute the same code.
- It is OK if your solution leaves the barbers waiting for new customers when all the customers have been served, i.e. deadlocks at the end.

1.
```
import edu.ucdavis.jr.JR;
public class Barber {

 private final int N = 10 ;

 public cap void() chair [] = new cap void()[3];
 public cap void() done [] = new cap void()[3];
 public cap void() button [] = new cap void()[3];

 public Barber() {
    for(int i=0; i<3; i++)
     {
       chair [i] = new sem(1);
       button[i] = new sem(0);
       done [i] = new sem(0);
     }
 }
```

```
process Barber((int i=0 ; i < 3; i++)) {
  while (true)
  {

    // Siesta time if no one is on the chair
    System.out.println("Siesta␣time␣for␣Barber␣"+i) ;
    P(button[i]);
    System.out.println("Cutting␣hair␣Barber␣"+i) ;
    JR.nap(200) ;
    V(done[i]);
    System.out.println("Done␣Barber␣"+i) ;
  }
}

process Client((int i=0;i<N;i++)) {
    System.out.println("Client␣"+i+"␣choose␣the␣chair␣"+(i%3)) ;
    P(chair[i%3]) ;
    System.out.println("Client␣"+i+"␣pressed␣button␣␣"+(i%3)) ;
    V(button[i%3]);

    P(done[i%3]) ;
    System.out.println("Client␣"+i+"␣leaving") ;
    V(chair[i%3]) ;
}

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

}
```

Question 2. Consider a savings account shared by several people. Each person associated with the account may deposit or withdraw money from it. The current balance in the account is the sum of all deposits to date less the sum of all withdrawals to date. Clearly, the balance must never become negative. A person making a deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds.

Your assignment You are giving the task to write up a Java class to implement shared accounts. More specifically, you need to provide the following interface.

```
class SharedSavingsAccount {
   public SharedSavingsAccount(long initialBalance) ;
   public void deposit(int amount) ;
   public withdraw(int amount) ;
}
```

Please, do not care about fairness in your solution. You should use Java 5 monitors as synchronization primitive. Here is a short reference of what you will need from `java.util.concurrent.locks`.

```
class ReentrantLock {
 public ReentrantLock();
 public Condition newCondition();
 public void lock();
```

```
  public void unlock();
}
class Condition {
  public void await();
  public void signal();
  public void signalAll();
}
```

(10p)

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SharedSavingsAccount {

    private final Lock lock = new ReentrantLock();
    private final Condition sufficientBalance = lock.newCondition();

    protected long balance;

    public void SharedSavingsAccount(long initialBalance) {
               balance = initialBalance;
    }

    public void deposit(int amount) {
          lock.lock() ;

          balance += amount;

          sufficientBalance.signalAll() ;

          lock.unlock() ;
    }

   public void withdraw(int amount) throws InterruptedException {
          lock.lock() ;

          while (amount > balance)
                sufficientBalance.await();
          balance -= amount;

          lock.unlock() ;
    }
}
```

Question 3. The Problem Haskell utilizes special variables, called MVars, as means for synchronization among threads. For simplicity, we assume that we have only one MVar that stores integer numbers. We are going to learn how this MVar works by showing its primitives:

- `putMVar(int x)`: This primitive stores the value of x into the MVar only if the MVar is empty. Once a value is stored into the MVar, we say that the MVar is full.
- **`int`** `takeMVar()`: This primitive reads the value stored into the MVar and empties it.

Your assignment You should implement a server that handle an MVar in JR and serves the primitives mentioned above. As usual, if any of these operations cannot be completed because some condition was not fulfilled (e.g., the MVar is empty), then the thread should block until that condition is satisfied. Below, you can see an skeleton of a server implementation serving the mentioned operations.

```
...

private process server_MVar {
while (true)
{
    inni void putMVar(int x) ... {
        ...
    }

    [] int takeMVar() ... {
       ...
    }
}
```

Your task is to fill in the dots in the skeleton provided above in order to complete the implementation of the server.

To get full points your solution must fulfill the following criteria:

- You must use message passing for synchronization. No other synchronization constructs are allowed.

(8p)

```
import edu.ucdavis.jr.JR;
public class MVar {


public op void putMVar(int) ;
public op int takeMVar() ;

private op int pending () ;
int value = 0 ;
boolean full = false ;

private process server_MVar {
while (true)
{
    inni void putMVar(int x) st !full {
             full = true ;
             value = x ;
    }

    []
        int takeMVar() st full {
           full = false ;
           return value ;
    }
```

```
 }
}


 private process client_MVar {
    int x ;
    putMVar(42) ;
    x = takeMVar() ;
    System.out.println("The␣variable␣is␣"+x);
    x = takeMVar() ;
    System.out.println("The␣variable␣is␣"+x);
 }


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

}
```

Question 4. **Background** There are several approaches to parallelize sequential code. In this exercise, we explore one of them.

Consider the definition of function `any` in Erlang.

```
any(P, []) -> false ;
any(P, [H|T]) -> case P(H) of
                    true -> true ;
                    false -> any(P,T)
                  end.
```

Function `P` is a predicate, i.e., when `P` is applied to an element it returns true or false. Function `any` determines if there is an element in the list which fulfills predicate `P`. For example, calling

```
any(fun (X) -> X > 0 end, [-3,0,-2])
```

returns **`false`**. On the other hand,

```
any(fun (X) -> X > 0 end, [-3,1,-2])
```

returns **`true`**.

Problem Computing if an element fulfills the predicate `P` (i.e., `P(X)`) could be expensive, and therefore take a considerable amount of time. Moreover, there is no reason to apply `P` in the same order as the elements appearing in the list. It is enough to just find one that fulfills `P`. With this in mind, we will to speed up the code that uses `any` with a parallel version of it, called `pany`.

Function `pany` works like `any`, but when we call `pany(P,L)`, it creates one parallel process for each application of `P` to an element in `L`. We assume that applying `P` to an argument always succeed, i.e., it will not crash the program. Moreover, it is OK if your code leave running processes that are still computing `P(x)` after `pany` has returned the answer. Do not worry to process all the messages of a process' mailbox as long as `pany` gives the right answer.

Your assignment Your task is to provide an implementation of `pany` in Erlang. To achieve that, you need to implement a series of auxiliary functions. Even though it sounds difficult, do not be afraid, the solution of this exercise is just a few lines in Erlang!

a) Implement a function called `gather`. This function takes a positive number `N` and an unique identifier `Ref` as arguments, i.e., it is called as `gather(N,Ref)`. The process calling `gather(N,Ref)` waits to get *at most* `N` messages of the form {`Ref, Fulfill`} in order to do something with them (it is up to you to figure out what!). For instance, if a process calls `gather(2,123456)` and it has the messages {`123456, True`} and {`555555, False`} in its mailbox, then the function process the message {`123456, True`}. *(5p)*

b) Implement a function called `distribute`. This function takes a list `List`, a predicate `P`, and a unique reference `Ref` as arguments. When calling `distribute(List,P,Ref)`, the function creates, for each element `X` of the list, a process to compute `P(X)`. Each of these processes will send a message to the process executing `distribute`. The messages will have two components. The first component is the unique identifier `Ref`, and the second component being `P(X)`. *(3p)*

c) Implement function `pany` using `gather` and `distribute`. *(2p)*

```
-module(pany).
-export([gather/2, distribute/3, pany/2]).


gather(0,_) -> false ;
gather(N,Ref) ->
    receive
      {Ref, true } -> true ;
      {Ref, false} -> gather (N-1, Ref)
    end.


distribute([], _, _) -> true ;
distribute([Value|Rest], P, Ref) ->
    S = self(),
    spawn( fun() -> S ! {Ref, P(Value)}

            end ),
    distribute(Rest, P, Ref).

pany(P,L) ->
    Ref = erlang: make_ref(),
    distribute(L,P,Ref),
    gather(length(L),Ref).
```

Question 5. The command `test-and-set` is used to change the value of a boolean variable in an atomic manner. More precisely, `test-and-set(C, L)` is defined as atomically executing the instructions {`L := C; C:= false`} only when `C` is true. Otherwise, the command does nothing.

This command is useful to solve the critical section problem as follows.

boolean C := true	
Process p	Process q
boolean Lp := false	boolean Lq := false
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p-: repeat	q-: repeat
p2: test-and-set(C, Lp)	q2: test-and-set(C, Lq)
p3: until Lp	q3: until Lq
p4: critical section	q4: critical section
p5: C:= true	q5: C:= true

Here, we have two processes trying to get into their critical section (Process p on the left, and Process q on the right). The main idea of this solution is to consider the boolean value *true* (initially stored in C) as an access token. In other words, when a process has the token, it can access its critical section. The command test-and-set is responsible to pass the token among the processes. For instance, observe that line p2 will pass the token from variable C into variable Lp. This only occurs when C is true, i.e., when the token is available in C.

As usual, we assume that lines p4 and q4 terminate, but do not assume that p1 or q1 do so. Each line p1, p4, q1, and q4 take some time to execute, but assume nothing about their relative runtimes. Assume further that p and q run on separate (but identical) processors. The code segments from line p- to p3 and q- to q3 are busy-waiting loops.

Your assignment

a) Construct the state diagram for an abbreviated version of this program. Try to abbreviate it as much as possible. *(6p)*

b) Use the diagram above to show that mutual exclusion holds for the program and that whenever process p is at p2, it will progress eventually to p5. *(4p)*

c) Interpreting the boolean value true as 1 and false as 0, we assume the following invariant: C+Lp+Lq ≤ 1. Using this invariant, prove that it cannot happen that both processes are simultaneously into their critical sections. Note that you do not need to prove that the given invariant is indeed an invariant, but just use it. *(6p)*

a) To construct the diagram, you can abbreviate the program by removing the lines corresponding to the non-critical sections as well as p_4 and q_4.

b) Here, you need to check that there is no state in the diagram which includes p_5 and q_5 at the same time.

c) The invariant does not hold by the program. This was a typo mistake which I clarified during the exam. More specifically, it should have been test-and-set(Lp,C) instead of p_5 and test-and-set(Lq,C) instead of q_5. However, assuming the invariant as the exercise dictates, you can still prove what it is requested above. I will proceed to do that.

We prove mutual exclusion by contradiction. If both processes are in the critical sections, then we are in p_5 and q_5 simultaneously. This implies that Lp and Lq are true (see p_3 and q_3). Therefore, this contradicts the invariant since Lp + Lq = 2.

Question 6. Temporal logic is a logic that helps us to reason when certain events occur in certain concurrent computations. This type of logic often consists of standard logical operators (like implication

$p \rightarrow q$, negation $\neg p$, etc) as well as special ones expressing facts about time. For this exercise, we consider a logic with two special operators related to time.

For the purpose of this exercise, a computation is a (possibly infinite) sequence of states s_0, s_1, s_2, ... A formula F applied to an state i is written as $F(s_i)$. We formally define the special operators *always* and *eventually* as follows.

$$\Box_k(F) = \forall n \geq k \cdot F(s_n) \qquad\qquad \Diamond_k(F) = \exists n \geq k \cdot F(s_n)$$

The definition of $\Box_k(F)$ indicates that formula F holds for any state s_n after state s_{k-1}. In other words, F *always* holds after state s_{k-1}. On the other side, $\Diamond_k(F)$ indicates that formula F holds for a particular state s_n after state s_{k-1}. Thus, we can say that F *eventually* holds after state s_{k-1}.

Your assignment Your task is to formally prove the following equivalences.

1 $\neg(\ \Box_k(F)\ ) = \Diamond_k(\neg F)$. This equation indicates that if "it is not the case that F is always true after s_{k-1}", then "it must be the case that F gets false at some point after s_{k-1}." *(6p)*

2 $\neg(\ \Diamond_k(F)\ ) = \Box_k(\neg F)$. This equation, on the other hand, indicates that if "it is not the case that F is true at some point after s_{k-1}", then "it must be the case that F is false all the time after s_{k-1}." *(6p)*

To prove these items, you need to just apply the definition given above and know the rule for negating $\forall$ and $\exists$.

1 $\neg(\ \Box_k(F)\ ) = \neg(\ \forall n \geq k \cdot F(s_n)\ )$ by applying the definition of $\Box_k$. Now, we apply the definition of $\neg\forall$ (from propositional logic). So, we have that $\neg(\ \forall n \geq k \cdot F(s_n)\ ) = \exists n \geq k \cdot \neg F(s_n)$. Then, we know that $\exists n \geq k \cdot \neg F(s_n) = \Diamond_k(\neg F)$ by definition of $\Diamond_k$.

2 $\neg(\ \Diamond_k(F)\ ) = \neg(\ \exists n \geq k \cdot F(s_n)\ )$ by applying the definition of $\Diamond_k$. Now, we apply the definition of $\neg\exists$ (from propositional logic). So, we have that $\neg(\ \exists n \geq k \cdot F(s_n)\ ) = \forall n \geq k \cdot \neg F(s_n)$. Then, we know that $\forall n \geq k \cdot \neg F(s_n) = \Box_k(\neg F)$ by definition of $\Box_k$.