UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF2140 — Parallel Programming
Day of examination:	15. June 2012
Examination hours:	9.00 – 13.00
This problem set consists of 11 pages.	
Appendices:	None
Permitted aids:	None, i.e., no special exam resources are allowed.

Please make sure that your copy of the problem set is complete before you attempt to answer anything.

Some general advises and remarks:

- This problem set consists of several independent parts. It is wise to make good use of your time.
- You can score a total of 100 points on this exam. The number of points stated on each part indicates the weight of that part.
- You can make your own clarifications if you find the examination text ambiguous or imprecise. Such clarifications must be written clearly in the delivered answer.
- Make short and clear explanations!

Good luck!

(Continued on page 2.)

Problem 1 Actions (weight 20)

1a Actions in FSP (weight 8)

Consider the following FSP processes:

```
A = (a -> b -> A  |   d -> A).
B = (b -> a -> B  |   d -> B).
C = (c -> C  |   d -> C).
||AB  = (A || B).
||ABC = (A || B || C).
```

1. Draw the state machine defined by process A.
2. Explain the behavior of ||ABC.
3. Is there a possibility of deadlock for ||AB? Solution: No deadlock.
4. Are there any progress problems for ||AB for some of the actions? Explain briefly. Solution: progress violation for a and b
5. Redefine A by hiding the action a. What is the resulting alphabet of A ? Is there now a possibility of deadlock for ||AB? Are there now any progress problems for ||AB with the redefined A?

   Solution: no deadlock, no longer progress violation for a and b
6. Redefine C by extending the alphabet of C by $\{a, b\}$. Now consider A || B || C. Are there any deadlock or progress violations?

   Solution: no deadlock, progress violation for a and b

1b Shared actions in Java (weight 7)

How would you implement the action d of the model `||ABC` in a Java solution? Solution:

```
class Monitor(){
int i;
publ sync void d(){
i=1+1; if( i=3 ) {notifyAll(); i=0 } else wait; }
}
```

(Continued on page 3.)

1c Buffers in FSP (weight 5)

Consider a system in FSP with connects a producer PROD to a consumer CONS by means of a buffer.

```
PROD = (in[i:0..3]-> PROD).
CONS = (out[i:0..3] -> CONS).
```

1. Define a *single-cell* buffer BUF in FSP; i.e., the buffer can at most contain one element and construct a system SYS which connects the buffer to the producer and the consumer.

   Solution:

   ```
   BUF = (in[i:0..3] -> out[i] -> BUF).
   || SYS = (PROD || BUF || CONS) .
   ```

2. Define a *two-cell* buffer BUF2 by composing two copies of BUF and construct a system SYS2 which connects the buffer to the producer and the consumer.

   Solution:

   ```
   || BUF2 = (BUF/{mid/out} || BUF/{mid/in})\{mid} .
   || SYS2 = (PROD || BUF2 || CONS) .
   ```

3. Define BUF3 by modifying BUF2 to allow *two* different producers and construct a system SYS3 which connects the buffer to the producers and the consumer.

   Solution:

   ```
   || BUF3 = BUF2/{ {a.in,b.in}/in } .
   || SYS3 = ({a,b}:PROD || BUF3 || CONS) .
   ```

Problem 2 Semaphores (weight 25)

2a Semaphores in FSP (weight 3)

Make an FSP process which models a semaphore.

Solution:

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up -> SEMA[v+1]
              | when (v>0) down -> SEMA[v-1]
              ),
SEMA[Max+1] = ERROR.
```

(Continued on page 4.)

2b Semaphores in Java (weight 3)

Semaphores can be implemented in Java as passive objects that react to up and down actions. Define a class `Semaphore` in Java which implements general semaphores.

Solution:

```
class Semaphore {
  private int value;

public Semaphore (int initial){
  value = initial; }

  synchronized public void up(){
    value++;
    notify();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value == 0) wait();
    value--;
  }
}
```

2c Job processing using semaphores (weight 7)

Consider a `Server` object which processes jobs from `Client` objects. Jobs have a given size. When a client wants to process a job, the client calls the method `request(size)` on the server (the parameter `size` is needed for Problem 2d). When this method returns, the client can process the job. When the jobs has finished, the client calls `release()` on the Server to allow another job to be processed. A `Server` class which processes jobs may be implemented as follows:

```
class Server {
  private bool free;

  public Server(){
    free = true; }

  public void request(Int size){
   free = false;
  }

  public void release(){
    free = true;
  }
}
```

Modify the `Server` class to make sure that jobs are processed *one at the time* and in the *order of arrival*. You should only use instances of the `Semaphore` class

(Continued on page 5.)

to synchronize the processes. In your solution, `request` and `release` should not be synchronized methods.

Solution: Although many implementations of semaphores use a FIFO queue, one cannot in general rely on FIFO queing (book, p. 87). Therefore, we implement a ticket system. We use the baton style of signalling; i.e., we do not leave CR if there are other waiting processes, but pass on CR to the next waiting process.

```
public class Server {
  private boolean free;
  int wp; // waiting processes
  int np; // next process
  private Semaphore mutex = new Semaphore(1);
  private Semaphore queue = new Semaphore(0);

  public Server(){
    free=true; wp=0; np=0;
  }

  public void request(int size) throws InterruptedException {
    mutex.down(); // enter CR
    if (!free){ // server is in use
      int myticket = wp; // take a ticket
      wp++; // increase counter
      while (np!=myticket){ // not my turn
        queue.up(); // signal another process
        queue.down(); // wait
      }
    }
    free = false; // server is in use
    mutex.up();   // leave CR
  }

  public void release() throws InterruptedException {
    mutex.down();
    if (wp>np) { // waiting processes
      queue.up(); // signal process
    } else { // no waiting process
      free = true; // server is free
      mutex.up(); // leave CR
    }
  }
}
```

2d Shortest job first using semaphores (weight 12)

Assume that jobs may have the sizes 1, 2, 3,...,`maxsize` (where `maxsize` is a parameter of `Server`). Modify the server such that jobs are scheduled using a *shortest job first* strategy; i.e., if there are waiting jobs of different sizes, a job with a smaller size should be selected before a job with a larger size.

Solution: We use one semaphore for each job size i, and use the counters `wp[i]` to keep track of the number of waiting processes of each size.

(Continued on page 6.)

```
public class Server {
  private boolean free;
  private int wp[];
  private Semaphore mutex;
  private Semaphore queue[];

  public Server(int maxsize){
    free = true;
    mutex = new Semaphore(1);
    int i=1;
    while (i<= maxsize){
       wp[i]=0;
    queue[i]= new Semaphore(0);
    }
  }

  public void request(int size) throws InterruptedException {
    mutex.down(); // enter CR
    if (!free){ // server is in use
      wp[size]++; // increase counter for my queue
      mutex.up(); // leave CR
      queue[size].down(); // wait in my queue
      wp[size]--; // decrease counter for my queue
    }
    free = false; // server is in use
    mutex.up(); // leave CR
  }

  public void release() throws InterruptedException {
    mutex.down(); // enter CR
    int i = 1;
    boolean flag = true;
    while (i<= maxsize && flag){ // check queues
      if (wp[i]>0) { // waiting processes of size i
        queue[i].up(); // signal process of size i
        flag=false; // stop checking
      }
      i++;
    }
    if (flag){ // no waiting processes
      free = true;
      mutex.up();
    }
  }
}
```

Problem 3 A Shared Shower Room (weight 55)

Imagine that there is one shower room to be used by both girls and boys, but not at the same time. Thus at any time, there can either be boys or girls in the shower, but not both. The shower room has a limited capacity of M, thus the

(Continued on page 7.)

number of persons in the shower room at a given time should be at most M.

Each boy or girl is supposed to repeat the following cycle of actions:

enter, *shower*, and *leave*

(the actions may possibly be labeled or indexed). No other actions should be needed in the first subproblem below.

3a Process Modeling (weight 15)

Program the shower system in FSP by means of a monitor SHOWER and with a number N of BOY processes and a number N of GIRL processes ($N > M$). First define processes BOY, GIRL, SHOWER, and afterwards the whole shower system SYS.

Solution:

```
const M = 3          -- places in the shower
const N = 4           -- number of boys and of girls
PERSON = (enter -> shower -> leave -> PERSON).
||GIRL = girl:PERSON.
||BOY  =  boy:PERSON.

SHOWER = SH[0][0],
SH[girls:0..M][boys:0..M] =
( when (boys==0 && girls < M) girl.enter -> SH[girls+1][boys]
| girl.leave -> SH[girls-1][boys]
| when (girls==0 && boys < M)  boy.enter -> SH[girls][boys+1]
| boy.leave -> SH[girls][boys-1]

|  when (girls  > 0) girl.shower  -> SH[girls][boys]  -- redundant
|  when (boys   > 0)  boy.shower  -> SH[girls][boys]  -- redundant
).
||SYS = ([1..N]:GIRL || [1..N]:BOY || [1..N]::SHOWER) .
```

3b Deadlock (weight 2)

Explain briefly if there is any deadlock in the system SYS or not.

Solution: No deadlock since there is only one shared resource and at least one girl or boy may progress.

(Continued on page 8.)

3c Safety (weight 8)

1. Define a safety property SAFE in FSP ensuring that a boy and a girl will never be in the shower at the same time.
2. Show how this can be checked in FSP for the shower system SYS.
3. Will the safety property be satisfied for SYS?
4. Will the safety property be satisfied for the monitor process SHOWER alone?

Explain briefly.

Solution 1:

```
property SAFE =  SA[0][0],
SA[girls:0..M][boys:0..M] =
( girl.enter -> SA[girls+1][boys]
| girl.leave -> SA[girls-1][boys]
| boy.enter  -> SA[girls][boys+1]
| boy.leave  -> SA[girls][boys-1]

| when (girls > 0 && boys == 0) girl.shower -> SA[girls][boys]
| when (boys  > 0 && girls== 0) boy.shower  -> SA[girls][boys]
).

||SAFESYS    = (SAFE || SYS).    -- the property holds here
||SAFESHOWER = (SAFE || SHOWER). -- the pretty does NOT hold here,
```

This property holds for SYS, but not for SHOWER (i.e. if boys and girls processes are omitted).

Solution 2: may have more redundancy (or use if-then-else), like

```
property SAFE =  SA[0][0],
SA[girls:0..N][boys:0..N] =
( when (boys == 0 && girls < M) girl.enter -> SA[girls+1][boys]
| when (girls > 0 && boys == 0) girl.leave -> SA[girls-1][boys]
| when (girls ==0 && boys < M)  boy.enter -> SA[girls][boys+1]
| when (boys  > 0 && girls== 0) boy.leave -> SA[girls][boys-1]

| when (girls > 0 && boys == 0) girl.shower  -> SA[girls][boys]
| when (boys  > 0 && girls== 0) boy.shower  -> SA[girls][boys]
).

||SAFESYS    = (SAFE || SYS).    -- the property does hold here
```

(Continued on page 9.)

```
||SAFESHOWER = (SAFE || SHOWER). -- the property does NOT hold here,
```

3d Progress (weight 3)

Define a progress property in FSP expressing that some girl will eventually be able to enter the shower.

Solution: Progress P = { {1..N}.girl.enter}

3e Adverse Conditions (weight 3)

Solution: use priorites to give *girl enter* less priority and then reuse the progress property from above

3f Fair System (weight 8)

1. Improve the shower system such that it is fair with respect to both girls and boys wishing to enter the shower. You may introduce new actions.
2. How can you show with FSP that the revised system is fair?

Solution:

```
SHOWER = SH[0][0][False],
SH[girls:0..M][boys:0..M][q:0..1] =
( when (boys==0 && girls==0) boy.wait->boy.enter ->SH[girls][boys+1][q]
| when (boys==0 && girls==0)girl.wait->girl.enter->SH[girls+1][boys][q]
| when (boys==0 && girls<M && q==0)   girl.enter ->SH[girls+1][boys][q]
| when (girls > 1)                    girl.leave ->SH[girls-1][boys][q]
| when (girls ==1&&q) girl.leave->    boy.enter  ->SH[girls-1][boys+1][False]
| when (girls > 0 && q==0)            girl.wait  ->SH[girls][boys][False]
| when (girls > 0)                    boy.wait   ->SH[girls][boys][True]

| when (girls==0 && boys<M && q==0)   boy.enter  ->SH[girls][boys+1][q]
| when (boys > 1)                     boy.leave ->SH[girls][boys-1][q]
| when (boys==1 && q)    boy.leave -> girl.enter ->SH[girls+1][boys-1][False]
| when (boys > 0 && q==0)             boy.wait  ->SH[girls][boys][False]
| when (boys > 0)                    girl.wait  ->SH[girls][boys][True]

|  when (girls > 0)                  girl.shower ->SH[girls][boys][q]
|  when (boys  > 0)                   boy.shower ->SH[girls][boys][q]

|  when (boys==0 && girls==0) boy.wait->boy.enter ->SH[girls][boys+1][q]
```

(Continued on page 10.)

```
|  when (boys==0 && girls==0)girl.wait->girl.enter->SH[girls+1][boys][q]
).
||SYST = ( [1..N]:GIRL|| [1..N]:BOY || [1..N]::SHOWER)
     >>  [1..N].girl.enter, [1..N].boy.enter.
progress GIRLprogress = [1..N].girl.enter
progress BOYprogress = [1..N].boy.enter
```

LTSA: No deadlocks/errors. No progress violations detected.

3g Java implementation (weight 10)

Make a Java implementation of the first version of the (unfair) monitor SHOWER. You do not need to consider fairness to boys and girls.

Solution:

```
class SHOWER {
  private int girls =0;
  private int boys  =0;

public synchronized void girlEnter()
                                throws InterruptedException {
    while (boys>0 || girls==M) wait();
    ++girls; }
public synchronized void girlLeave() {
    --girls; notifyAll(); }
public void shower() { //for boy or girl, not synchronized!
   ...take shower...}
public synchronized void boyEnter()
                                throws InterruptedException {
    while (girls>0 || boys==M) wait();
    ++boys; }
synchronized public void boyLeave() {
    --boys; notifyAll(); }
}
```

3h Java implementation: Notification (weight 3)

Explain whether you should use *notify* or *notifyAll* in your Java solution.

Solution: there are several waiting conditions therefore a simple notify is not sufficient.

3i Java implementation: Synchronization (weight 3)

Explain if all methods in the Java implementation need to by synchronized.

(Continued on page 11.)

Solution: Showering is supposed to be done in parallel, so method shower need not be synchronized. Shower variables for the showering (when added) are supposed to be disjoint. There is nothing (so far) that makes it necessary to make shower synchronized.