T–79.4301 **Autumn 2008**

Parallel and Distributed Systems

Home Exercise 3 - Deadline: 4th of December 2008 at 12:15 (strict deadline!)

Return your answer via email to `t794301-autumn08@tcs.hut.fi` with "Homework 3, [student number]" as the subject. Attach to the email a zip or tar file with a separate file for each part of the homework, named "`part1.pml`", "`part2-a.pml`", "`part2-b.txt`", etc. A template tar file for the homework is at: `https://noppa.tkk.fi/noppa/kurssi/t-79.4301/harjoitustyot` under Home exercise 3. Download it, unpack the files, and modify them to contain your answers. When you are done, pack the files to a tar or zip file. On unix workstations this can be done with the command "`tar cvf homework3.tar homework3`", when you have a directory "`homework3`" with your answer files in it. Then send the package to the course email address with the subject above. Submissions that arrive late are not graded! Be sure to send your answer in time, and remember that it may take some time for email messages to arrive. You will receive a confirmation when your submission arrives.

The home exercises are personal, no group work allowed! There are three rounds of home exercises of 10 points each. To pass the home exercises ≥ 15 points are needed and ≥ 24 points gives a +1 to the exam grade (no effect to exam grades 0 or 5).

1. We are designing a microprocessor containing $N \geq 1$ independent execution units (cores) numbered $0, 1, \ldots, N-1$ and denoted `core(0)`, `core(1)`, ..., `core(N-1)`. The design also contains M ($1 \leq M \leq N$) floating point co-processors (FPUs) numbered $0, 1, \ldots, M-1$ and denoted `fpu(0)`, `fpu(1)`, ..., `fpu(M-1)`. The design also includes a floating point unit controller `fpu_controller` for handling the allocation of the floating point co-processors to the different execution units. The task of the controller is to ensure that no FPU is accessed by more than one core at a time. The communication between the cores and FPUs as well as the behaviour of the FPUs themselves have been abstracted away from the model. The communication between the cores and the FPU controller is modelled and is as follows.

   For each execution unit `core(i)` ($0 \leq i \leq N-1$), there exists a channel `from_core[i]` sending messages from `core(i)` to the FPU controller. This channel has a capacity one message and carries two kinds of messages. When `core(i)` wishes to gain access to an FPU co-processor, it first sends the message `(req_fpu,any)` to the FPU controller. When the core has finished using FPU j ($0 \leq j \leq M-1$),

it sends the message `(free_fpu,j)` to the controller. In a similar way, the controller sends messages back to `core(i)` ($0 \leq i \leq N-1$) via another channel `to_core[i]` with capacity 1. The messages sent via this channel are of the form `(grant_fpu,j)`, telling `core(i)` that it can start using FPU j for some $0 \leq j \leq M-1$.

The hardware is able to implement all channel manipulation operations available in Promela, including tests for the emptiness or fullness of a message channel. No other means of communication are available between the cores and the FPU controller besides the message channels mentioned above. No new message types for these channels may be added to the design, either.

Your task is to model the FPU controller in Promela. As a starting point, use the partial Promela model given at the end of this exercise sheet. (Also available on the course Noppa page homework 3 template.) Return your Promela model as your answer. (5 p.)

2. a) Modify your Promela model to contain assertions which triggers if more than one core is using the same FPU at the same time. Return a modified Promela model.

   b) Return the log of a verification run showing that the model with four cores and two floating point units satisfies this safety property.

   (1 p.)

3. a) Does your FPU controller guarantee progress of each core in the sense that if `core(i)` has sent the `(req_fpu,any)` message to the FPU controller, then the FPU controller will also eventually grant some FPU to `core(i)` by sending it a message of the form `(grant_fpu,j)` for some $0 \leq j \leq M-1$?

   b) If not, modify your Promela model to guarantee this property and return it as your answer together with a verification log showing that the assertions you added in question 2 remain valid (for the same number of cores and FPUs). To simplify the task, you may assume that every core that is granted access to an FPU j will eventually send the corresponding `(free_fpu,j)` message back to the FPU controller. In either case, return a Promela model satisfying the property.

   c) In either case, explain also (in English, Swedish, or Finnish) the ideas you used in your solution to guarantee that your model has the above described progress property.

   (4 p.)

```
#define N 4 /* number of cores */
#define M 2 /* number of FPUs */
#define any M

/* Message types used by the protocol. Do not change. */
mtype = {req_fpu, grant_fpu, free_fpu};

/* Message channels used by the protocol. Do not change. */
chan from_core[N] = [1] of { mtype, byte };
chan to_core[N] = [1] of { mtype, byte };

/* A process modelling the FPU use of a core. Do not change. */
proctype core(byte id) {
    byte my_fpu = M;
    do
        :: from_core[id] ! req_fpu, any;
           to_core[id] ? grant_fpu, my_fpu;
           atomic {
               printf("MSC: Core %d using FPU %d\n", id, my_fpu);
               from_core[id] ! free_fpu, my_fpu;
               my_fpu = M
           }
    od
}

active proctype fpu_controller() {
    /* Add your Promela model of the FPU controller here. */

    /* Following line added to make a syntactically valid Promela model */
    /* Remove it first. */
    skip
}

/* The init process for starting all other processes. */

init {
    int id;
    atomic {
        id = 0;
        do
            :: (id < N) -> run core(id); id++
            :: (id == N) -> break
        od;
        id = 0
    }
}
```