**DTU Compute**
Technical University of Denmark
Building 324
DK–2800 Lyngby
Denmark

*HHL 2024–08–29*

02158 CONCURRENT PROGRAMMING

FALL 2024

# Auxiliary Exercises

# Petri Nets

## Exercise Petri.1

Five boats shuttle between two jetties, A and C, via a jetty B. Jetty A and C each has a capacity of two boats, whereas jetty B has a capacity of three boats. Make a Petri Net model of the boat traffic.

## Exercise Petri.2

Four actions $A$, $B$, $C$, and $D$ are to be synchronized as follows:

> After execution of $A$, either $B$ or $C$ is executed. Concurrently with this, $D$ is executed. All of this is repeated forever.

(a) Draw a Petri-net in which the actions $A$ to $D$ are represented by transitions and synchronized as described above.

(b) Which pairs of actions can be executed in parallel?

(c) Which interleavings (sequences of single transition firings) are possible for the first cycle of the execution?

## Exercise Petri.3

Make a Petri Net for following process:

*Root Galettes*

| | | |
|---|---|---|
| *1 leek* | *2 large potatoes* | *salt, pepper* |
| *2 carrots* | *1 egg yolk* | *olive oil* |
| *1 parsnip* | *(1 tsp Maizena)* | |

The leek is rinsed and finely chopped. The root vegetables are peeled and cut *en julienne*. Mix with egg yolk and optionally *Maizena* (cornflour). Season with salt and pepper and fry in hot oil like small pancakes, a couple of minutes on each side. Put on absorbing kitchen paper and serve.

Note: The paste is divided into 10 portions and there are two frying pans available.

*Problem originally due to the Danish cook, Claus Meyer.*

## Exercise Petri.4

Draw the Petri Net $N = (P, T, F)$ where

$$
\begin{array}{rcl}
P & = & \{p_1, p_2, p_3\} \\
T & = & \{t_1, t_2, t_3\} \\
F & = & \{(p_1, t_1), (p_1, t_2), (p_3, t_2), (p_2, t_3), (t_1, p_2), (t_2, p_2), (t_2, p_3), (t_3, p_1)\}
\end{array}
$$

with the marking $M_0 = (2, 0, 1)$ (corresponding to $(p_1, p_2, p_3)$).

Write all *simultaneous firings* possible from $M_0$ using the notation $M_0 \xrightarrow{U} M'$.

## Exercise Petri.5

Write down the mathematical model for Figure 1.1 in [Basic].

# Transition Systems

## Exercise Trans.1

Find all interleavings of the two processes:

$$\begin{array}{ll} P_a\colon & a_1, a_2, a_3 \\ P_b\colon & b_1, b_2 \end{array}$$

## Exercise Trans.2

Assume that two processes $P_1$ and $P_2$ consist of sequences of $n_1$ and $n_2$ actions respectively. Find an expression for the number of possible interleavings of $P_1$ and $P_2$.

## Exercise Trans.3

Find all interleavings of the three processes:

$$\begin{array}{ll} P_a\colon & a_1 \\ P_b\colon & b_1, b_2 \\ P_c\colon & c_1 \end{array}$$

## Exercise Trans.4

Assume that $P_i$ is a process consisting of $n_i$ actions. Find an expression for the number of interleavings of

$$P_1, P_2, \ldots, P_k$$

## Exercise Trans.5

Draw *transition diagrams* for the two processes in the following program:

$$\begin{array}{l} \textbf{var}\ x, y : \mathit{integer}; \\ x := 1;\quad y := 2; \\ \textbf{co}\ x := y + 1 \parallel y := x - 1\ \textbf{oc} \end{array}$$

Now, draw (the reachable part of) the *transition graph* for the full program. The nodes should be states of the form $(x, y, t_1, t_2, \pi_1, \pi_2)$, where $t_i$ are local variables and $\pi_i$ are the control variables of the two processes.

From the graph, determine the possible final values of $x$ and $y$.

## Exercise Trans.6

Make a transition diagram for each of the two processes in Figure 2.2 in [Andrews].

# Shared Variables

## Exercise Share.1 (Lock-step problem)

Write, using shared variables only, to pieces of program, $SYNC_A$ and $SYNC_B$, that synchronize to processes $P_A$ and $P_B$ such that they proceed in *lock-steps*. More precisely, if $op_A$ and $op_B$ are operations in $P_A$ and $P_B$ respectively, then the number of times these two operations have been executed must differ by at most one.

| | |
|---|---|
| **process** $P_A$; | **process** $P_B$; |
| **repeat** | **repeat** |
| $SYNC_A$; | $SYNC_B$; |
| ... | ... |
| $op_A$; | $op_B$; |
| ... | ... |
| **forever** | **forever** |

## Exercise Share.2

In the following program, it is attempted to establish a critical region for two concurrent processes by using two shared boolean variables $C_1$ and $C_2$:

**var** $C_1, C_2$ : *boolean*;

$C_1 := \mathit{false}$; $C_2 := \mathit{false}$;

| | |
|---|---|
| **process** $P_1$; | **process** $P_2$; |
| **repeat** | **repeat** |
| $nc_1$: non-critical section$_1$; | $nc_2$: non-critical section$_2$; |
| $r_1$: **repeat** | $r_2$: **repeat** |
| $a_1$: $C_1 := \neg C_2$; | $a_2$: $C_2 := \neg C_1$; |
| **until** $\neg C_2$; | **until** $\neg C_1$; |
| $cs_1$: critical section$_1$; | $cs_2$: critical section$_2$; |
| $e_1$: $C_1 := \mathit{false}$ | $e_2$: $C_2 := \mathit{false}$ |
| **forever**; | **forever**; |

(a) Draw the transition diagrams for $P_1$ and $P_2$.

(b) Show that the program does **not** ensure mutual exclusion.

(c) Assume that $a_1$ and $a_2$ are executed as *atomic statements* instead, i.e. $a_1$: $\langle\, C_1 := \neg C_2 \,\rangle$ and $a_2$: $\langle\, C_2 := \neg C_1 \,\rangle$.

Determine whether the algorithm now ensures mutual exclusion.

## Exercise Share.3

Consider the problem of establishing a critical region using a coordinator process addressed in Andrews Ex. 3.12. A proposal for the form of the processes is:

**process** $P[i : 1..n] =$
  **repeat**
    non critical section$_i$;
    $enter[i] := true$;
    $\langle$ **await** $in[i] \rangle$;
    critical section$_i$;
    $in[i] := false$
  **forever**

(a) Write a proposal for the coordinator process.

(b) Express mutual exclusion among $n$ process as an invariant.

(c) State and prove some auxiliary invariants of your program that may be combined to show mutual exclusion.

Hint: What can be said about $in[i]$ in the critical section? What is is known about the state of the coordinator process when $in[i]$ is true?

(d) Is your algorithm fair?

# Theory (Safety and Liveness)

## Exercise Theory.1

Consider the concurrent program:

$$
\begin{array}{l}
\textbf{var } x, y : \mathit{integer} := 0; \\
\\
\textbf{co} \\
\quad \textbf{repeat } a_1\text{: } \langle\, y < 2 \to y := y+1;\ \ x := y \,\rangle \textbf{ forever} \\
\| \\
\quad \textbf{repeat } a_2\text{: } \langle\, x = 0 \to y := 0 \,\rangle \textbf{ forever} \\
\| \\
\quad \textbf{repeat } a_3\text{: } x := 0 \textbf{ forever} \\
\textbf{oc}
\end{array}
$$

**Question 1.1:**

(a) Prove inductively that $I \triangleq 0 \le x \le y \le 2$ is an invariant of the program.

(b) Draw the (reachable part of) the transition graph for the program. Since control remains at the $a$-actions, only the $(x, y)$ part of the state needs be shown.

(c) Determine whether $\neg(x = 1 \land y = 2)$ is an invariant of the program.

**Question 1.2:**

(a) Argue that $\Box\Diamond x = 1$ holds for the program under the assumption of weak fairness.

(b) Show that $\Box\Diamond x = 2$ does not hold, even under the assumption of strong fairness.

**Question 1.3:**

(a) Assume that the action $a_2$ cannot be considered atomic as a whole.

Draw the transition diagram representing $a_2$ then and show that $I$ is no longer an invariant of the program.

(b) In the original program, assume that the action $a_1$ is replaced by the refinement:

$$b_1\text{: } \textbf{await } y < 2;\quad c_1\text{: } t := y;\quad d_1\text{: } y := t + 1;\quad e_1\text{: } \langle\, x := y \,\rangle$$

where $t$ is a local integer variable.

State a predicate $H$ that implies $I$, holds initially, and is inductive for the program (i.e. strong enough to be preserved by all atomic actions).

# Semaphores

## Exercise Sema.1

Three processes $P_1$, $P_2$, and $P_3$ execute three operations $A$, $B$, and $C$ respectively.

The operations are to be synchronized using semaphores as follows:

**var** $SA, SB, SC$ : *semaphore*;

$SA := 0$; $SB := 0$; $SC := 0$;

```
process PA;        process PB;        process PC;
  repeat             repeat             repeat
    A;                 B;                 P(SC);
    V(SC);             V(SC);             P(SC);
    P(SA)              P(SB)              C;
  forever            forever              V(SA);
                                          V(SB);
                                        forever
```

Draw a Petri net in which the operations $A$, $B$, and $C$ are synchronized the same way as in the above program. In the net, the operations must occur as transitions.

## Exercise Sema.2

Recall the problem and solution to Exercise Petri.2.

Now, the four operations/actions are to be executed by four sequential processes $P_A$, $P_B$, $P_C$, and $P_D$ respectively. Write a program using semaphores to synchronize the four processes such that the operations $A$ to $D$ are synchronized as in the Petri-net. (The choice of which operation to execute, $B$ or $C$, need not be fair, and can be left to the semaphore mechanism.)

## Exercise Sema.3

The meeting problem (barrier problem, lock-step problem) for two processes has been solved in Section 3.6 in [Basic] using general semaphores.

(a) Show that this solutions does **not** work with *binary semaphores*.

(b) Solve the meeting problem for two processes using binary semaphores only. Use the semaphore invariant to show that binary semaphores are sufficient.

## Exercise Sema.4

Solve the meeting problem for three processes using semaphores.

## Exercise Sema.5

Write a piece of code $SYNC_i$ that solves the meeting/barrier problem for an arbitrary number of processes $N$ using semaphores only.

## Monitors

In all the below exercises you should assume Signal-and-Continue (SC) semantics of condition queues unless otherwise stated.

### Exercise Mon.1

Write a monitor with two procedures $SYNC_A$ and $SYNC_B$ to be used by two processes $P_A$ and $P_B$ respectively. The monitor should synchronize the two processes, i.e. make them meet/wait for each other.

### Exercise Mon.2

Now, the above problem is generalized to making $N$ processes meet before any of them can proceed (also known as the *barrier problem*). Write a monitor with a single procecedure $SYNC$ to be used by all the processes for the synchronization.

### Exercise Mon.3

The general meeting problem from the preceeding exercise is now to be modified such that the $N$ processes not only meet but also “share the loot”. I.e. each process comes with a number (given as a parameter to $SYNC$) and get the mean value of all numbers back (as a return value). Write a monitor that solves this problem. Beware that due to the SC semantics, processes may call the monitor again before all have got their share of the loot.

Hint: Use an extra ”pre-queue” where processes may be delayed while the sharing takes place.

### Exercise Mon.4

(a) Write a monitor with two operations *sleep* and *wakeup* that implements the synchronization mechanism of Andrews Ex. 4.6. You should assume that *spurious wakeups* do not occur.

(b) Show how the monitor could be implemented if *spurious wakeups* might occur.

Hint: Use e.g. a modification of the *ticket algorithm* using a counter to hold the current round.

## Exercise Mon.5

Let $M$ be a positive constant. Consider the following specification of a *chunk semaphore*:

**monitor** *ChunkSem*;

  **var** $s$ : *integer* := 0;

  **procedure** $V()$ : $\langle\, s = 0 \rightarrow s := s + M \,\rangle$;

  **procedure** $P()$ : $\langle\, s > 0 \rightarrow s := s - 1 \,\rangle$;

**end**;

(a) Implement the monitor.

(b) State and argue for a monitor invariant expressing the range of the variable $s$.

(c) State and argue for a monitor invariant expressing that calls of $P()$ do not wait unneccesarily.

(d) Suppose that $M$ is small compared to the number of processes that may call $P()$. Does your solution avoid unneccesary wakeups? If not, try to minimize the wakeups. Is the property from (c) still a monitor invariant? If not, try to remedy this.

(e) Determine if your monitor (and invariants) would be robust towards *spurious wakeups*.

(f) Describe how you would have to implement the monitor in Java.

## Exercise Mon.6

In Andrews Figure 5.7, a `Timer` monitor is implemented by a *covering condition* which is *true*, ie. whenever the *tod* (time-of-day) is changed, all waiting processes are woken up. This may be quite expensive.

Optimize the monitor such that the processes are only woken up when the next relevant point of time is reached.

# CSP

## From Concurrent Programming Exam, June 1994 (4-hours)

### PROBLEM 3 (approx. 15 %)

Three CSP processes $P_1, P_2$, and $P_3$ perform three operations $A$, $B$, and $C$ respectively. The operations are to be synchronized which is accomplished by communication among the processes:

```
process P1 =          process P2 =                process P3 =
  repeat                repeat                      repeat
    P2 ! ();              if  P1 ? () → skip          P2 ! ();
    A                     []  P3 ? () → skip          C
  forever                 fi;                       forever
                          B
                        forever
```

**Question 3.1:**

Draw a Petri-net in which the three operations $A$, $B$, and $C$ are synchronized the same way as in the CSP program. In the net, the operations should be represented by transitions.

The operations are now to be executed by three sequential processes $P_A$, $P_B$, and $P_C$:

```
process PA =       process PB =       process PC =
  repeat             repeat             repeat
    A                  B                  C
  forever            forever            forever
```

**Question 3.2:**

Show how semaphores can be used to synchronize the three processes such that $A$, $B$, and $C$ are synchronized in the same way as in the CSP program.

# Rendezvous

## Exercise Rendez.1

A monitor-implementation of a semaphore-like mechanism is given below. In the monitor, *posinteger* is the type of all positive ($> 0$) integers.

**monitor** *Event*

  **var** $S$ : *integer* := 0;
      $Q$ : *condition*;

  **procedure** *Pass*;
    **if** $S = 0$ **then** *wait*($Q$)

  **procedure** *Clear*(**var** $r$ : *integer*);
    $r := S$;
    $S := 0$

  **procedure** *Release*($v$ : *posinteger*);
    $S := S + v$;
    *signal_all*($Q$)

**end**

(a) Define a module with the same interface as *Event* and implement it using rendezvous. Make sure that you get an effect similar to *signal_all* in *Release*.

(b) The *Event* monitor corresponds to the semaphore mechanism found in the (now bygone) operating system OS/2. Show how to use (an instance $e$ of) *Event* to implement a classical semaphore $s$. Hint: `V`($s$) can be implemented simply as $e.Release(1)$.

# Deadlocks

## From Concurrent Programming Exam, December 1998 (4-hours)

**PROBLEM 4** (approx. 10 %)

In a system there is one instance of a resource type $A$, two instances of a type $B$, and three instances of a type $C$. The resources are used by four processes $P_1$, $P_2$, $P_3$, and $P_4$. The processes have declared their maximal resource demands as shown below. Furthermore, it is shown which resources have been allocated and which are requested at a certain moment.

| | *Max* | | |
|---|---|---|---|
| | $A$ | $B$ | $C$ |
| $P_1$ | 1 | 2 | 0 |
| $P_2$ | 0 | 1 | 1 |
| $P_3$ | 1 | 0 | 3 |
| $P_4$ | 0 | 2 | 2 |

| | *Allocation* | | | *Request* | | |
|---|---|---|---|---|---|---|
| | $A$ | $B$ | $C$ | $A$ | $B$ | $C$ |
| $P_1$ | 1 | 0 | 0 | 0 | 0 | 0 |
| $P_2$ | 0 | 1 | 0 | 0 | 0 | 0 |
| $P_3$ | 0 | 0 | 1 | 0 | 0 | 1 |
| $P_4$ | 0 | 1 | 0 | 0 | 0 | 1 |

**Question 4.1:**

Draw a resource allocation graph corresponding to this situation. In the graph, the expected, not yet requested, resource needs should be indicated by dashed arrows.

**Question 4.2:**

(a) Show that the situation is safe.

(b) Determine whether $P_4$ can be granted the requested $C$-instance according to the banker's algorithm.

# Parallel Computation

## Exercise ParComp.1

In a system computations may be carried out by submitting tasks to a thread pool with a fixed number of worker threads. The system is executed on a machine with 8 uniform processors. It is assumed that there are no other activities in the system and that overhead from the thread pool management and scheduling can be ignored.

A computation consists of five independent computation tasks with the following execution times (in seconds):

| | |
|---|---|
| $A$ | 1 |
| $B$ | 2 |
| $C$ | 2 |
| $D$ | 5 |
| $E$ | 6 |

Assume that two worker threads are allocated for the thread pool.

(a) Draw a task scheduling scenario in which the shortest possible execution time is achieved for the computation. State the execution time and determine the speedup obtained.

(b) Draw a task scheduling scenario in which the longest possible execution time is achieved for the computation. State the execution time and determine the speedup obtained.

# FURTHER SELECTED EXAM PROBLEMS

## From Concurrent Systems Exam, December 2002 (4-hours)

### PROBLEM 3 (approx. 15 %)

Below, a monitor implementation of a synchronization mechanism *Gate* is shown. The gate may be opened or closed by an operation *Set*. Processes call *Pass*() to pass the gate and have to wait if the gate is closed. A special operation $Go(k)$ lets up to $k$ of the currently waiting processes pass through the gate.

```
monitor Gate

  var open : boolean := false;
      Queue : condition;

  procedure Pass() {
    if ¬open then wait(Queue);
  }

  procedure Set(b : boolean) {
    open := b;
    if open then signal_all(Queue);
  }

  procedure Go(k : integer) {
    for j in 1..k do signal(Queue);
  }

end
```

**Question 3.1:**

(a) Define a predicate $I$ expressing that calls of *Pass*() do not wait unnecessarily.

(b) Argue that $I$ is an invariant of the monitor.

(c) Describe the effect of $Go(k)$ if there are less than $k$ calls of *Pass*() currently waiting.

**Question 3.2:**

The functioning of the given monitor *Gate* is now to be implemented by a module with the following specification:

```
module Gate
  op Pass();
  op Set(boolean);
  op Go(integer);
end
```

Write a server process for the module *Gate* that services the operations by rendezvous in such a way that it functions like the given monitor *Gate* as seen from the calling processes.

# From Concurrent Systems Exam, December 2003 (2-hours)

## PROBLEM 2 (approx. 30 %)

*The questions in this problem can be solved independently of each other.*

**Question 2.1:**

A process $P$ uses three shared integer variables $x$, $y$, and $z$. The variable $x$ is both read and written by other processes, whereas $y$ and $z$ are only read by other processes. Determine which of the following statements in $P$ can be considered to be atomic.

$$\begin{array}{llcll} a: & x := x + 1 & \qquad & d: & y := y + 1 \\ b: & x := y + 1 & & e: & x := y + z \\ c: & y := x + 1 & & f: & z := y + z \end{array}$$

**Question 2.2:**

A concurrent program is given by:

$\textbf{var}\ x, y : \mathit{integer} := 0;$

$\textbf{co}\ x := y + 1 \parallel \langle\, y := x + 2 \,\rangle;\ x := 2\ \textbf{oc}$

(a) Draw a transition diagram for each process.

(b) Determine all possible final states $(x, y)$ of the program.

**Question 2.3:**

Let $x$ and $y$ be integer variables. Determine which of the predicates $P$, $Q$, and $R$ are preserved by which of the actions $a_1$, $a_2$, and $a_3$, respectively:

$$\begin{array}{rcllll} P & \triangleq & x + y \geq 0 & \qquad & a_1: & y := 0 \\ Q & \triangleq & 0 \leq y \leq x & & a_2: & \langle\, y < 0 \rightarrow y := x + 1 \,\rangle \\ R & \triangleq & x \neq y & & a_3: & \langle\, y = 0 \rightarrow x := 0 \,\rangle \end{array}$$

**Question 2.4:**

Let $x$ and $y$ be integer variables and let the temporal logic formula $F$ be defined by:

$$F \triangleq (\Box\, y > x \geq 0) \wedge (\Box\Diamond\, x = 0) \wedge (x = 0 \leadsto x \neq 0)$$

(a) Let states be given by pairs $(x, y)$. Give an example of an execution for which $F$ holds. The execution should be given as a short sequence of states which is repeated forever.

Now, consider each of the following actions within a program:

$$\begin{array}{llcll} a_1: & \langle\, \textbf{await}\ x = 0 \,\rangle & \qquad & a_3: & \langle\, \textbf{await}\ x = 0 \vee y > 1 \,\rangle \\ a_2: & \langle\, \textbf{await}\ y > 1 \,\rangle & & a_4: & \langle\, \textbf{await}\ x = 0 \wedge y > 1 \,\rangle \end{array}$$

Assume that control has reached the particular action and that $F$ is valid for the program.

(b) Determine which of the actions will be eventually executed assuming weak fairness.

(c) Determine which of the actions will be eventually executed assuming strong fairness.

# From Concurrent Systems Exam, December 2003 (2-hours)

## PROBLEM 3 (approx. 20 %)

Let $N$ be a positive integer. The server-based module *Batch* given below implements a synchronization mechanism that "collects" a batch of $N$ items provided by calls of $put()$ which may then be "removed" by a call of $unload()$.

```
module Batch
  op put();
  op unload();
body

  process Control;
    var count : integer := 0;
    repeat
      while count < N do
            in put() → count := count + 1 ni;
      in unload() → count := 0 ni;
    forever;

end Batch;
```

**Question 3.1:**

Assume $N = 3$. Suppose that, concurrently, $unload()$ is called by two processes and $put()$ is called by five processes. Assuming no further calls, describe the overall effect of these seven calls.

**Question 3.2:**

Now, the module *Batch* is to be replaced with a monitor which provides the same operations and behaves in the same way. Write such a monitor.

# From Concurrent Systems Exam, December 2004 (2-hours)

## PROBLEM 3 (approx. 25 %)

The server-based module *Latch* given below implements a simple synchronization mechanism. The latch maintains a non-negative count which may be set to some value by $set(k)$ and decremented by $down()$. The operation $await()$ returns only when the count has reached zero.

```
module Latch
  op set(integer);
  op down();
  op await();

body

  process Control;
    var count : integer := 0;
    repeat
      in set(k : integer)          → if k ≥ 0 then count := k
      [] down()                    → if count > 0 then count := count − 1
      [] await() and count = 0     → skip
      ni
    forever;

end Latch;
```

**Question 3.1:**

The module *Latch* is to be replaced with a monitor which provides the same operations and behaves in the same way. Write such a monitor.

**Question 3.2:**

In this question we consider *Latch* to be a type of which distinct instances can be declared.

A number of worker processes $P_1, P_2, \ldots, P_n$ need to establish a *barrier* to synchronize their rounds of work. The barrier is to be implemented by four *Latch*-instances and a *coordinator process* $Q$.

The code for the processes is shown below:

```
var latch1, latch2, latch3, latch4 : Latch;

latch1.set(n);   latch2.set(1);

process P[i : 1..n];                 process Q;
  repeat                               repeat
    do work_i;                           ⋮
    SYNCHRONIZE:                       forever;
      latch1.down();
      latch2.await();
      latch3.down();
      latch4.await()
  forever;
```

Write the body of the coordinator process $Q$ such that the given SYNCHRONIZE code implements barrier synchronization for $P_1, P_2, \ldots, P_n$.

# From Concurrent Systems Exam, December 2006 (4-hours)

## PROBLEM 2 (approx. 25 %)

In a system, a number of operations $A_1, A_2, \ldots, A_n$ with corresponding successor operations $B_1, B_2, \ldots, B_n$ ($n \geq 1$) plus an operation $C$ are to be executed the following way:

($*$) $A_1, A_2, \ldots, A_n$ are executed concurrently. When $A_i$ is has finished, the corrresponding $B_i$ is executed. As soon as all of $A_1, A_2, \ldots, A_n$ have finished, $C$ can be executed concurrently with the $B$-operations. When all the operations $B_1, B_2, \ldots, B_n$, and $C$ have finished, the execution starts all over again.

**Question 2.1:**

For a system with $n = 2$, draw a Petri Net in which the five operations $A_1$, $A_2$, $B_1$, $B_2$, and $C$ are synchronized as described by ($*$). In the net, the operations should be represented by transitions.

**Question 2.2:**

The operations are to be executed by $n$ sequential processes $P_1, P_2, \ldots, P_n$ plus a sequential process $Q$. These processes have the form:

| | |
|---|---|
| **process** $P[i : 1..n]$;<br>  **repeat**<br>    $A_i$ ;<br>    $B_i$ ;<br>  **forever** | **process** $Q$;<br>  **repeat**<br>    $C$;<br>  **forever** |

Show how to synchronize these processes using semaphores so that the operations $A_1, \ldots, A_n$, $B_1, \ldots, B_n$ and $C$ become synchronized as described by ($*$).

**Question 2.3:**

The processes $P_1, P_2, \ldots, P_n$ and $Q$ are now to be synchronized using a monitor *Sync*.

**monitor** *Sync*<br>  ⋮<br>**end**

| | |
|---|---|
| **process** $P[i : 1..n]$;<br>  **repeat**<br>    $A_i$ ;<br>    $B_i$ ;<br>  **forever** | **process** $Q$;<br>  **repeat**<br>    $C$;<br>  **forever** |

Write a monitor *Sync* providing appropriate synchronization procedures and show how these procedures are to be used by the processes $P_i$ ($i = 1..n$) and $Q$ such that the operations $A_1, \ldots, A_n$, $B_1, \ldots, B_n$ and $C$ are executed as described by ($*$).

# From Concurrent Systems Exam, December 2007 (4-hours)

## PROBLEM 1 (approx. 25 %)

The implementation shown below of a critical region for two processes, $P_1$ and $P_2$, utilizes a machine instruction which indivisibly reads and increments an integer variable and another instruction which indivisibly decrements an integer variable. In the program these instructions are denoted by statements of the form $\langle\, Y := X;\; X := X + 1\,\rangle$ and $\langle\, X := X - 1\,\rangle$. The variables $C_1$, $C_2$ and $L$ are not changed in any other places than those shown.

**var** $C_1, C_2, L$ : *integer*;

$C_1 := 0;\quad C_2 := 0;\quad L := 0;$

| | |
|---|---|
| **process** $P_1$; | **process** $P_2$; |
| **repeat** | **repeat** |
| $nc_1$: non-critical section$_1$; | $nc_2$: non-critical section$_2$; |
| $a_1$: $\langle\, C_1 := L;\; L := L + 1\,\rangle$; | $a_2$: $\langle\, C_2 := L;\; L := L + 1\,\rangle$; |
| $w_1$: **await** $C_1 = 0$; | $w_2$: **await** $C_2 = 0$; |
| $cs_1$: critical section$_1$; | $cs_2$: critical section$_2$; |
| $b_1$: $\langle\, L := L - 1\,\rangle$; | $b_2$: $\langle\, L := L - 1\,\rangle$; |
| $d_1$: $C_2 := 0$ | $d_2$: $C_1 := 0$ |
| **forever**; | **forever**; |

**Question 1.1:**

Draw the part of the transition diagram for $P_1$ which would represent $a_1$ if the statements $a_i$ were not atomic, but instead given as ordinary statement sequences $\{C_i := L;\; L := L + 1\}$ (for $i = 1, 2$).

For $i = 1, 2$ the predicate $R_i$ is defined by: $R_i \triangleq at\ w_i \lor in\ cs_i \lor at\ b_i$

Furthermore, the predicate $H$ is defined by:

$$H \triangleq |R_1| + |R_2| = L$$

where $|\cdot|$ denotes the numerical value of a boolean expression, ie. $|true| = 1$, $|false| = 0$.

**Question 1.2:**

Argue that $H$ is an invariant for the given program.

You are informed that the following predicates $F$, $G_1$, and $G_2$ are invariants of the program:

$$
\begin{aligned}
F &\triangleq 0 \le C_1 \le 1 \land 0 \le C_2 \le 1 \\
G_1 &\triangleq C_1 = 1 \Rightarrow at\ w_1 \land (R_2 \lor at\ d_2) \\
G_2 &\triangleq C_2 = 1 \Rightarrow at\ w_2 \land (R_1 \lor at\ d_1)
\end{aligned}
$$

**Question 1.3:**

(a) State which values the variables $L$, $C_1$, and $C_2$ may take in a state where $at\ a_1 \land R_2$ holds.

(b) Define a predicate $I$ of the form

$$I \triangleq R_1 \land R_2 \Rightarrow \text{"boolean expression using } C_1 \text{ and } C_2\text{"}$$

which expresses what can be said about $C_1$ and $C_2$ when both processes are in their $R$-sections. Argue that $I$ is an invariant of the program.

(c) Show, using the invariants, that mutual exclusion is ensured in the given program.

**Question 1.4:**

(a) Express as a temporal logic formula the property of resolution for the critical region.

(b) Argue that resolution holds for the critical region.

(c) Determine if starvation may occur in the given program and justify your answer.

## From Concurrent Systems Exam, December 2008 (4-hours)

**PROBLEM 3** (approx. 40 %)

A *region with variable capacity* is a resource which may be acquired by a number of *user processes* up to a certain capacity which may be changed dynamically. Access to the region is obtained by calling the operation *acquire*() and release of the region is done by calling the operation *release*(). The capacity of the region is initially given by the constant $N$ ($N \geq 0$). The capacity may be changed dynamically by calling the operation $set(k : natural)$ (where *natural* is the type of non-negative integers) which will set the capacity to $k$. If the number of current users is greater than $k$, a call of $set(k)$ will block until the number of users equals $k$.

The server-based module *VarReg* given below implements such a variable capacity region:

```
module VarReg
  op acquire();
  op release();
  op set(k : natural);
body

  process Control;
    var users : integer := 0;
        max : natural := N;
    repeat
      in acquire() and users < max → users := users + 1
      [] release()                  → users := users − 1
      [] set(k : natural) →
            max := k;
            while users > max do
                in release() → users := users − 1 ni
      ni
    forever;

end VarReg;
```

*The questions in this problem are all related to the VarReg module, but can be solved independently of each other.*

**Question 3.1:**

It is desired to add an operation *get_users*() **returns** *integer* which should return the current number of region users. The operation may be called at any time and should return immediately.

Describe the changes which should be made to the *VarReg* module in order to implement such an operation.

**Question 3.2:**

A system with exactly one *writer process* and $m$ ($m > 0$) *reader processes* are to be synchronized using the given module *VarReg*.

Show how this may be accomplished by stating the required initial capacity $N$ as well as the pre and post protocols for reading and writing.

**Question 3.3:**

In a system there are three instances of a resource type $A$, one instance of type $B$ and two instances of type $C$. The resources are used by three processes $P_1$, $P_2$, and $P_3$.

The three resource types are controlled by a copy of the *VarReg* module each, called $Reg_A$, $Reg_B$, and $Reg_C$. The modules are initialized with the constants $N_A = 3$, $N_B = 1$, and $N_C = 2$ respectively, but are otherwise identical to *VarReg*. The resource instances are acquired and released by using the *acquire*() and *release*() operations on the respective modules. The *set*() operation is not used in this system.

The processes have the following form where ... indicates use of the acquired resources:

| | | |
|---|---|---|
| **process** $P_1$; | **process** $P_2$; | **process** $P_3$; |
| $Reg_C.acquire()$; | $Reg_A.acquire()$; | $Reg_A.acquire()$; |
| $Reg_A.acquire()$; | ... | $\rightarrow$ ... |
| $\rightarrow$ ... | $Reg_B.acquire()$; | $Reg_C.acquire()$; |
| $Reg_B.acquire()$; | $\rightarrow$ $Reg_C.acquire()$; | $Reg_A.acquire()$; |
| ... | ... | ... |
| $Reg_A.release()$; | $Reg_A.release()$; | $Reg_A.release()$; |
| $Reg_B.release()$; | $Reg_B.release()$; | $Reg_A.release()$; |
| $Reg_C.release()$; | $Reg_C.release()$; | $Reg_C.release()$; |

At a given moment, the processes have reached the locations indicated with arrows ($\rightarrow$). In particular, $P_2$ has ended the call of $Reg_B.acquire()$, but has **not yet** called $Reg_C.acquire()$.

(a) Draw a resource allocation graph corresponding to the situation at the given moment. In the graph, the expected future resource claims should be indicated by dashed arrows.

(b) Explain why this situation would normally be called *safe*.

(c) Demonstrate that deadlock may occur from the given situation.

(d) Show with a brief argument how the system can made generally *deadlock free* by exchanging neighbour acquisitions.

**Question 3.4:**

Specify the effect of the operations *acquire*(), *release*(), and *set*($k$) in terms of conditional atomic actions acting upon the state variables *users*, *max* and possibly auxiliary variables.

Hint: The effect of *set*($k$) must be stated as two consecutive atomic actions.

**Question 3.5:**

The given module *VarReg* is to be replaced by a monitor which provides the same operations and behaves in the same way.

(a) Write such a monitor.

(b) State a monitor invariant expressing that calls of *acquire*() do not wait unnecessarily.