Written examination, December 9, 2024

Course: Concurrent Programming Course no. 02158

Aids allowed: All written works of reference

Exam duration: 4 hours

Weighting:				
	PROBLEM 1:	approx. 15 %	PROBLEM 4:	approx. 30 %
	PROBLEM 2:	approx. 15 %	PROBLEM 5:	approx. 20 %
	PROBLEM 3:	approx. 20 %		

PROBLEM 1 (approx. 15 %)

Three processes P_A, P_B, and P_C execute three operations A, B, and C respectively. The operations are to be synchronized, which is accomplished by means of semaphores:

var SA, SB, SC : *semaphore*;

$SA := 2$; $SB := 0$; $SC := 0$;

process P_A;	**process** P_B;	**process** P_C;
repeat	**repeat**	**repeat**
P(SA);	P(SB);	C;
A;	B;	P(SC);
V(SB)	P(SB);	V(SA);
forever	V(SC)	V(SA)
	forever	**forever**

Question 1.1:

Draw a Petri Net in which the three operations A, B, and C are synchronized in the same way as in the above program. In the net, each operation should be represented by one or more transitions.

Question 1.2:

Let the number of times the operations B and C have been executed be denoted by b and c respectively. Define a predicate I which characterizes the reachable combinations of b and c in the above program.

Question 1.3:

The operations are now to be executed by three sequential CSP-processes P_1, P_2, and P_3 respectively:

process P_1;	**process** P_2;	**process** P_3;
repeat	**repeat**	**repeat**
A	B	C
forever	**forever**	**forever**

Show how the processes may exchange void messages using CSP's synchronous communication so that A, B, and C are synchronized in the same way as in the above, semaphore-based program. [The operation to be executed by a process may occur more than once.]

PROBLEM 2 (approx. 15 %)

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

In a system, computations are carried out by submitting tasks to a thread pool with a fixed number k of worker threads which repeatedly execute tasks from the pool's task queue. The system is executed on a machine with four uniform processors. The ordering of the tasks in the queue is not known. It is assumed that there are no other activities in the system and that overhead from thread pool management and scheduling can be ignored.

Question 2.1:

A given computation may be split into any positive number n of independent tasks each demanding $1/n$ of the full computation time. The overhead of such splitting can be ignored.

(a) Assume that the computation is split into $n = 5$ tasks and that $k = 3$ worker threads are allocated for the thread pool. Determine the resulting speedup.

(b) Suggest how many tasks n this computation should be split into and how many worker threads k should be allocated for the thread pool in order to achieve the best possible speedup on this system. Determine the speedup expected.

Question 2.2:

In this question, $k = 3$ worker threads are assumed to be allocated for the thread pool.

A computation with a sequential execution time of 24 seconds can be divided into five independent tasks with corresponding execution times (in seconds).

A	B	C	D	E
1	3	5	7	8

A *master thread* performs the division into tasks, submits them to the thread pool and awaits their execution. All of these operations are assumed to take negligible processing time.

(a) Draw a worst-case task scheduling scenario and determine the resulting speedup.

(b) Explain why a speedup of 3 would be optimal for this thread pool and show that such a speedup is feasible by drawing a task scheduling scenario.

(c) Although the execution order of tasks submitted right after each other is not known, the master thread may influence the task scheduling by delaying the submission of some of the tasks.

Assume that a task T is submitted by the operation $submit(T)$ and that $sleep(d)$ makes the master thread sleep for d seconds. State a sequence of submission and sleep operations for which the best speedup, cf. (b), will be obtained.

PROBLEM 3 (approx. 20 %)

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

Question 3.1: A concurrent program is given by:

$$\textbf{var } x, y : \textit{integer} := 0;$$
$$\textbf{co } \; x := x+4 \; \| \; x := y+1 \; \| \; y := y+2 \; \textbf{oc}$$

(a) For each of the three processes, draw a transition diagram showing its atomic actions.

(b) Determine all possible final values of x for the program.

Question 3.2:

Consider the concurrent program:

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

$$\begin{array}{l}\textbf{co}\\ \quad \textbf{repeat } a_1\colon \langle\, x < 2 \wedge y = 0 \rightarrow x := x+1 \,\rangle \textbf{ forever}\\ \|\\ \quad \textbf{repeat } a_2\colon \langle\, y := x; \;\; x := 0 \,\rangle \textbf{ forever}\\ \|\\ \quad \textbf{repeat } a_3\colon \langle\, x \geq 1 \rightarrow y := x+1 \,\rangle \textbf{ forever}\\ \textbf{oc}\end{array}$$

(a) State for each of the following predicates P, Q, and R whether it is (in general) preserved by each of the actions a_1, a_2, and a_3:

$$\begin{array}{rcl} P & \triangleq & x \neq y \\ Q & \triangleq & 0 \leq x \leq y \\ R & \triangleq & x + y \geq 0 \end{array}$$

(b) Prove inductively that $I \triangleq 0 \leq x \leq 2$ is an invariant of the program.

(c) Draw the (reachable part of) the transition graph for the program. Only the (x, y) part of the state has to be shown.

(d) Consider the following temporal logic properties:

$$\begin{array}{rcl} F & \triangleq & \Box\Diamond\,(x+y>4) \\ G & \triangleq & (\Box\,(x+y\neq 3)) \rightsquigarrow y = 3 \end{array} \qquad \begin{array}{rcl} H & \triangleq & \Box\Diamond\,(y=2) \\ J & \triangleq & x = 2 \rightsquigarrow y = 2 \end{array}$$

Determine for each of F, G, H, and J whether it holds for the program under the assumption of weak fairness. Do the same under the assumption of strong fairness.

PROBLEM 4 (approx. 30 %)

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

In a system, there is a synchronization component called a *kitty* which may be seen as a shared object *Kit* holding a sum, s, which may be manipulated using two operations *put* and *take* specified by:

object *Kit*;

var s : *integer* := 0;

op *put*(k : *integer*) : $\langle\, s := s + k \,\rangle$;

op *take*() **returns** *integer* : {**var** r : *integer*; $\langle\, s > 0 \rightarrow r := s;\ s := 0 \,\rangle$; **return** r}

end

[Note that k as well as s may be negative.]

Question 4.1:

(a) Implement *Kit* as a monitor.

(b) Define a predicate I expressing that calls of *take* do not wait unnecessarily and argue that I is a monitor invariant.

Question 4.2:

The given component *Kit* is now to be implemented by a module specified by:

module *Kit*
op *put*(k : *integer*);
op *take*() **returns** *integer*;
end

Write a server process for the module which services the operations by rendezvous in such a way that it functions like the given component *Kit*.

Question 4.3:

Implement the operations of the given component *Kit* using semaphores for synchronization. The technique of *passing-the-baton* should be applied for that.

Question 4.4:

Show how n processes, $P_1, P_2, \ldots, P_n$ ($n \geq 1$), may use the given component *Kit* to establish a *one-time barrier* (i.e. a synchronization point, which is to be used only once). The component may be brought into a desired state by calling *put* before the processes are started.

Question 4.5:

(a) Show how a system of n ($n \geq 1$) *reader processes* and a single *writer process* may use the given component *Kit* for reader/writer synchronization. You may assume an initial call of *put* to be made in order to bring the component into a desired state before the processes are started.

(b) Discuss whether the solution is fair towards the readers and the writer respectively.

PROBLEM 5 (approx. 20 %)

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

In a system, it is possible to send *messages* of a given type *Msg* by *synchronous broadcast* to a number of receivers through a communication module *Broadcast* given below. Receiver processes call *listen*() to receive a message. Sender processes call *send*(m) to send a message m to the receivers currently listening but limited to a maximum of *Lim* receivers, where *Lim* is a given positive constant. A call of *send*(m) does not return until at least one receiver can receive the message.

```
module Broadcast
  op send(m : Msg);
  op listen() returns Msg;
body

  process Control;
    var rem : integer;
    repeat
      in send(m) and ?listen > 0 →
        rem := min(?listen, Lim);                    // min is the minium function
        while rem > 0 do
          in listen() → rem := rem − 1;  return m ni
      ni
    forever

end Broadcast;
```

Question 5.1:

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

[The calls of *send* and *listen*, respectively, may be handled in any feasible order. You may use the function *length*(c) which returns the actual number of processes waiting on a condition queue c. It may be utilized that *spurious wakeups* are assumed **not** to occur in the standard monitor definition.]

Question 5.2:

The behaviour of the module *Broadcast* should now be changed so that each message is sent simultaneously to *exactly* K receivers (without any limitation). Initially the value of K should equal the constant *Lim*. It must be possible to change the value of K dynamically using a new operation *set*:

```
op set(v : posinteger);          // posinteger is the type of positive integers.
```

Write a new module which implements this behaviour.