Written examination, December 7, 2022

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. 35 %
	PROBLEM 2:	approx. 15 %	PROBLEM 5:	approx. 15 %
	PROBLEM 3:	approx. 20 %		

PROBLEM 1 (approx. 15 %)

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

Question 1.1:

For a given computation, 10 % must be executed sequentially while the remaining 90 % may by be split arbitrarily into independent tasks to be executed in parallel on a machine with multiple processors (cores).

(a) Determine an upper limit for the speedup which may be obtained by executing the computation on a machine with 6 uniform processors.

(b) Determine an overall upper limit for the speedup which may be obtained by executing the computation on any multi-processor machine.

Question 1.2:

In a system, computations are carried out by submitting tasks to a thread pool with a fixed number of worker threads. The ordering of the tasks in the pool's task queue is not generally 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.

In this question, it is assumed that the system is executed on a machine with 4 uniform processors and that 3 worker threads are allocated for the thread pool.

(a) State the maximal speedup which might be obtained by using this thread pool.

A computation with a serial execution time of 21 seconds can be divided into five independent tasks with corresponding execution times (in seconds):

A	B	C	D	E
2	3	4	5	7

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.

(b) Draw a task scheduling scenario in which the maximal speedup is obtained for the computation.

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

PROBLEM 2 (approx. 15 %)

In a system, three operations A, B, and C are to be synchronized. Given that the number of times the operations have been executed is denoted by a, b, and c respectively, the following predicate I must be a *characteristic invariant* of the system:

$$I \triangleq a - 1 \leq b \leq a + 1 \quad \wedge \quad b \leq c \leq b + 1$$

That an invariant is characteristic means that it precisely describes the reachable states of the system.

Question 2.1:

Draw a Petri Net in which the three operations A, B, and C are synchronized so that I is a characteristic invariant for the net. In the net, the operations must appear as transitions.

Question 2.2:

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

process P_A;	**process** P_B;	**process** P_C;
repeat	**repeat**	**repeat**
A	B	C
forever	**forever**	**forever**

Show how to use semaphores for synchronizing the three processes so that I becomes a characteristic invariant of the program.

Question 2.3:

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

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 I becomes a characteristic invariant of the program.

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:

var x, y : *integer* := 0;
co $\langle x := y + 1 \rangle \parallel y := x + y + 2 \parallel x := 4$ **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:

var x, y : *integer* := 0;

co
  repeat a_1: $\langle x = y \wedge x < 2 \rightarrow y := y + 1 \rangle$ **forever**
$\parallel$
  repeat a_2: $\langle x \neq y \wedge x < 2 \rightarrow x := x + 1 \rangle$ **forever**
$\parallel$
  repeat a_3: $\langle x := y; \; y := 0 \rangle$ **forever**
oc

(a) Determine 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:

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

(b) Prove inductively that the predicate $I \triangleq 0 \leq y \leq 2$ 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:

$$\begin{aligned} F &\triangleq \Box\Diamond (x + y \geq 2) & \qquad H &\triangleq \Box\Diamond (x = 1) \\ G &\triangleq y = 2 \rightsquigarrow x = 2 & \qquad J &\triangleq \Box\Diamond (x = 2 \vee y = 2) \end{aligned}$$

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. 35 %)

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

In some systems, there may be many *atomic readings* of a shared data structure D and a few *atomic updates* of the form $\langle\, D := f(D) \,\rangle$ where the computation f takes very long time (say an image transformation). Using normal read/write locking, the structure D cannot be accessed from other processes while f is being computed.

For such systems the synchronization component L specified below may be used. An atomic reading may as usual be implemented by obtaining a read lock using the operation sequence:

$$L.lockR();\quad \text{read } D;\quad L.unlockR()$$

whereas an atomic update may be implemented by the operation sequence:

$$L.lockU();\quad \text{read } D;\quad \text{compute } D' = f(D);\quad L.upgradeU();\quad \text{write } D';\quad L.unlockU()$$

Informally, $lockU()$ first obtains an *update lock* which is like a read lock but with an intent to write later. The $upgradeU()$ operation converts the lock to a write lock which is finally released by $unlockU()$. This allows for concurrent reading during the computation phase.

In the questions below, the operations are assumed to be used according to the protocol schemes shown above.

object L;
  var r, u, w : *integer* := 0; // locks held
  op $lockR()$: $\langle\, w = 0 \rightarrow r := r + 1 \,\rangle$;
  op $unlockR()$: $\langle\, r := r - 1 \,\rangle$;
  op $lockU()$: $\langle\, u + w = 0 \rightarrow u := u + 1 \,\rangle$;
  op $upgradeU()$: $\langle\, r = 0 \rightarrow u := u - 1;\ \ w := w + 1 \,\rangle$;
  op $unlockU()$: $\langle\, w := w - 1 \,\rangle$;
end

Question 4.1:

(a) Assuming that the protocol schemes are followed, give a brief argument that the predicate:

$$H \ \triangleq\ r \geq 0 \ \wedge\ u \geq 0 \ \wedge\ w \geq 0$$

is an invariant of the component.

(b) Define a predicate I which is an invariant of the component and which characterizes the desired synchronization behaviour of the component.

(c) Explain why update locks (u) must be mutually exclusive.

The problem is continued on the next page

Question 4.2:

(a) Implement L as a monitor. You should avoid unnecessary signalling, but you need not care for starvation.

(b) State a predicate J which expresses that calls of $upgradeU()$ do not wait unnecessarily and argue that J is a monitor invariant.

(c) Show how to add a new operation $dropU()$ to the monitor. This operation is to be used by updates instead of $upgradeU() \ldots unlockU()$ if it turns out that writing is not necessary. Thus, the new operation should merely release the update lock.

Question 4.3:

(a) Explain why the operation sequence $L.lockU(); L.upgradeU()$ may be seen as a normal write lock operation.

(b) Show how n concurrent processes, $P_1, P_2, \ldots, P_n$ ($n \geq 2$), can use the given component L 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 executing some initialization code before the concurrent processes are started.

(c) Discuss whether your solution proposed for (b) can be used as a normal barrier (i.e. be used for repeated synchronization among the n processes).

Question 4.4:

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

```
module Lock
  op lockR();
  op unlockR();
  op lockU();
  op upgradeU();
  op unlockU();
end
```

(a) Write a server process for the module *Lock* which services the operations by rendezvous in such a way that it functions like L. The operations may be served in any feasible order and the solution does not have to be fair.

(b) Write an alternative server process for the module *Lock* so that the solution is fair towards both readings and updates.

[If your solution to (a) already fulfils this requirement, you may just refer to that.]

Question 4.5:

In a system with a maximum of m ($m \geq 1$) concurrent processes, the synchronization properties of the given component L may instead be achieved using two semaphores:

var SR : *semaphore* := m; SU : *semaphore* := 1;

Show how to implement the synchronization by providing code for each of the five operations of L using these semaphores.

PROBLEM 5 (approx. 15 %)

Question 5.1:

In a system there are two instances of a resource type A denoted by *resource identifiers* A^1 and A^2. Similarly, there is one instance B^1 of type B and two instances C^1 and C^2 of type C. The type of resource identifiers is generally denoted *rid*. The resources are used by three processes P_1, P_2, and P_3.

The resources are controlled via *asynchronous channels* acting as buffers for the resource identifiers considering them as messages. Three channels, C_A, C_B, and C_C, are initially loaded with the respective resource identifiers by executing:

chan C_A, C_B, C_C : *rid*;

send $C_A(\mathsf{A}^1)$; **send** $C_A(\mathsf{A}^2)$;
send $C_B(\mathsf{B}^1)$;
send $C_C(\mathsf{C}^1)$; **send** $C_C(\mathsf{C}^2)$;

The three processes may then acquire and release resource instances by respectively receiving and sending on the various channels. The processes have the following form:

```
process P1;                 process P2;                 process P3;
  var a, b, c : rid;          var a, b, c : rid;          var a, c : rid;

  receive CB(b);              receive CC(c)               receive CC(c);
→ use b                       receive CA(a);            → use c
  receive CA(a);            → receive CB(b);            receive CA(a);
  receive CC(c);              use a, b and c              use a and c
  use a, b and c            send CA(a);                 send CA(a);
  send CA(a);               send CB(b);                 send CC(c);
  send CB(b);               send CC(c);
  send CC(c);
```

At a given moment, the processes have reached the locations indicated with arrows (→).

(a) State the message contents (if any) of the three channels at this moment.

(b) 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.

(c) Explain why this resource allocation situation would normally be considered *safe*.

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

(e) Show with a brief argument how the system can be made generally *deadlock free* by exchanging neighbour acquisitions (possibly several times within the same process).