Written examination, December 7, 2021

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

PROBLEM 1 (approx. 15 %)

In a system, computations are carried out by submitting tasks to a thread pool with a fixed number 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 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.

Question 1.1:

(a) Determine an upper limit for the speedup which may be obtained for a computation if two worker threads are allocated for the thread pool

(b) Determine an upper limit for the speedup which may be obtained for a computation if six worker threads are allocated for the thread pool

Question 1.2:

In this question, three worker threads are assumed to be allocated for the thread pool.

A computation with a serial execution time of 15 seconds can be divided into five independent tasks with corresponding execution times (in seconds).

A	1
B	2
C	3
D	4
E	5

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 task scheduling scenario in which the shortest possible execution time is achieved for the computation. State the execution time and calculate the speedup.

(b) Draw a task scheduling scenario for which the resulting speedup is less than 2.

(c) Assume that the ordering of the task queue is known to be first-in-first-out (FIFO). State a sequence in which the master thread could submit the tasks to the thread pool in order to achieve the optimal execution time determined in (a).

PROBLEM 2 (approx. 20 %)

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

Question 2.1:

A concurrent program is given by:

var x, y : *integer* := 0;
co $x := 2;\ \ y := x + 3 \quad \| \quad x := x + y + 1$ **oc**

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

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

Question 2.2:

Let x and y be integer variables. Determine for each of the predicates P, Q, and R whether it is preserved by each of the actions a, b, and c:

$$P \triangleq x + y > 0 \qquad a: \ \langle\, x = y \to x := 0 \,\rangle$$
$$Q \triangleq x = 0 \vee y = 0 \qquad b: \ \langle\, y := 2y \,\rangle$$
$$R \triangleq 0 < x \leq y \qquad c: \ \langle\, x < 0 \to x := x + 1 \,\rangle$$

Question 2.3:

Consider the concurrent program:

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

co
  repeat a_1: $\langle\, x < 3 \to y := x;\ \ x := x + 1 \,\rangle$ **forever**
‖
  repeat a_2: $\langle\, y > 0 \to x := 0 \,\rangle$ **forever**
‖
  repeat a_3: $\langle\, x > 1 \to y := 2 \,\rangle$ **forever**
oc

(a) 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 has to be shown.

(b) Consider the following temporal logic properties:

$$F \triangleq \Box\Diamond\,(x + y = 0) \qquad H \triangleq \Box\Diamond\,(y = 2)$$
$$G \triangleq y = 2 \rightsquigarrow x = 1 \qquad I \triangleq \Box\Diamond\,(x + y = 4)$$

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

PROBLEM 3 (approx. 20 %)

In a system, a number of operations $A_1, A_2, \ldots, A_n$ with corresponding successor operations $B_1, B_2, \ldots, B_n$ ($n \geq 2$) are to be executed the following way:

($*$) $A_1, A_2, \ldots, A_n$ are executed concurrently. When A_i has finished, the corresponding B_i is executed as soon as also A_{i+1} has finished (for $i = 1, \ldots, n-1$). When A_n has finished, B_n is immediately executed. When all the operations $B_1, B_2, \ldots, B_n$ have finished, the execution starts all over again.

Question 3.1:

For a system with $n = 3$, draw a Petri Net in which the six operations A_1, A_2, A_3, B_1, B_2, and B_3 are synchronized as described by ($*$). In the net, the operations should be represented by transitions.

Question 3.2:

The operations are to be executed by n sequential processes $P_1, P_2, \ldots, P_n$ of the form:

```
process P[i : 1..n];
  repeat
    A_i ;
    B_i
  forever
```

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

Question 3.3:

The processes $P_1, P_2, \ldots, P_n$ are now to be synchronized using a monitor *Synch* with two procedures *doneA* and *doneB* which are called by the processes as shown:

```
monitor Synch
  procedure doneA(i : integer) :
  procedure doneB();
end

process P[i : 1..n];
  repeat
    A_i ;
    Synch.doneA(i);
    B_i ;
    Synch.doneB()
  forever
```

Implement the monitor in such a way that the operations $A_1, \ldots, A_n$, $B_1, \ldots, B_n$ are executed as described by ($*$).

[Spurious wakeups are assumed not to occur. Efficiency of the monitor is not emphasized.]

PROBLEM 4 (approx. 30 %)

In a candy factory there are a number of *candy processes*, P_i ($i = 1..n$), which control the production of different kinds of candies. Each candy process repeatedly makes a (small) batch of candies and pours them through a funnel into a bag (together with other kinds of candies). The packing is controlled by a single *packing process*, Q, that repeatedly prepares a bag, waits for different candy processes to fill the bag with a desired number of grammes of candies (at the minimum), and finally removes the bag.

The packing is to be coordinated by a control component *Pack* which is used as follows:

```
process P[i : 1..n];
  repeat
    make a batch of kind i;
    w := weight of batch;
    Pack.start(w);
    pour batch through funnel;
    Pack.end()
  forever
```

```
process Q;
  repeat
    prepare a bag for g grammes under funnel;
    Pack.fill(g);
    remove the filled bag from funnel
  forever
```

The following is a monitor implementation of *Pack*:

```
monitor Pack;

  var sum, limit, count : integer := 0;
      BagReady, Filled : condition;

  procedure start(w : posinteger) {
    while sum ≥ limit do wait(BagReady);
    sum := sum + w;
    count := count + 1;
    if sum < limit then signal(BagReady)
  }

  procedure end() {
    count := count − 1;
    if sum ≥ limit and count = 0 then signal(Filled)
  }

  procedure fill(g : posinteger) {
    limit := g;
    sum := 0;
    signal(BagReady);
    wait(Filled);
    limit := 0
  }
```

where *posinteger* is the type of positive (> 0) integers.

The problem is continued on the next page

Question 4.1:

(a) At some moment (while the monitor is free), the following holds: $limit = 500$, $sum = 215$, $count = 3$. What can you tell about the system's processes from these facts?

(b) Give an informal argument that $I \triangleq count \geq 0$ is an invariant of the monitor in the given system.

(c) Define a desired monitor invariant, I_Q, expressing that the packing process does not wait unnecessarily. Argue that I_Q is a monitor invariant.

(d) The monitor would not work if spurious wakeups could occur. Describe how to modify the monitor in order to make it resilient against spurious wakeups.

Question 4.2:

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

```
module Pack
  op start(w : posinteger);
  op end();
  op fill(g : posinteger);
end
```

Write a server process for the module *Pack* which serves the operations by rendezvous in such a way that it controls the packing and candy processes like the given monitor *Pack*.

Question 4.3:

To achieve better mixing of the candies, the synchronization should be modified so that all the candy processes needed for filling a bag start to pour *simultaneously*.

(a) Describe under which conditions such simultaneous pouring should be started. It may be assumed that eventually there will always be enough candy processes to fill a bag.

(b) Write a new *MixPack* monitor (with the same interface as the given monitor) which synchronizes the given processes in this way.

(c) Discuss whether the same synchronization of the given processes could be achieved using a server-based module serving the operations by rendezvous.

PROBLEM 5 (approx. 15 %)

In a system there are eight instances of a single resource type. The instances are used by two processes P_1 and P_2 each of which may require up to a maximum number of instances before it releases them all again. For P_1 and P_2, the maximums are four and six instances respectively.

Question 5.1:

(a) Assume that P_1 and P_2 at a given moment have been granted three instances each. Determine with a brief argument whether this situation is *safe*.

(b) Let the number of instances granted to P_1 and P_2 at a given moment be denoted by r_1 and r_2 respectively. State a predicate *Safe* which determines whether the situation given by r_1 and r_2 is safe.

It can be assumed that the edge conditions ($0 \leq r_1 \leq 4 \ \wedge \ 0 \leq r_2 \leq 6 \ \wedge \ r_1 + r_2 \leq 8$) are always satisfied.

The processes P_1 and P_2 are now assumed to be implemented in CSP. Their use of the resource instances is to be controlled through communication with a resource administrator process *Adm*. The processes acquire and release instances one at a time by synchronous communications of the form $Adm\,?\,acquire()$ and $Adm\,!\,release()$ respectively. The resource administrator should only control how many instances may be used by each process, not which particular instances are to be used.

Question 5.2:

Write a CSP process *Adm* serving P_1 and P_2 in such a way that resource instances are granted according to the principle of the Banker's Algorithm.