Written examination, December 6, 2023

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 %)

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 six uniform processors which are scheduled evenly between all active threads when seen over a period of time. 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 achieved for a computation if five worker threads are allocated for the thread pool.

(b) In this sub-question we consider a time-consuming computation which may be split into $t = 2^n$ (for some integer $n \geq 1$) independent tasks each demanding $1/t$ of the full computation time. Due to the nature of the computation problem, only such splitting into a power of two number of tasks is feasible.

Suggest how many tasks t 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. State any assumptions you might have to add.

Question 1.2:

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

A computation with a sequential execution time of 22 seconds can be divided into six independent tasks with corresponding execution times (in seconds).

A	B	C	D	E	F
1	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.

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

(b) 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 guarantee a speedup of at least 3.

PROBLEM 2 (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*;

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

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

Question 2.1:

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, the operations should be represented by transitions.

Question 2.2:

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

Question 2.3:

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.

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 $x := y + 2;\ \ y := 1 \quad \| \quad y := 3;\ \langle\, x := x + 1 \,\rangle$ **oc**

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

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

Question 3.2:

Consider the concurrent program:

var x, y : *integer* := 0;
co
  repeat a_1: $\langle\, x \leq 1 \rightarrow y := y + 2(1 - x);\ \ x := x + 1 \,\rangle$ **forever**
‖
  repeat a_2: $\langle\, y = 2 \rightarrow x := x + 1;\ \ y := 3 \,\rangle$ **forever**
‖
  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:

$$\begin{aligned} P &\triangleq 0 \leq y \leq x \\ Q &\triangleq x = y \\ R &\triangleq 0 < x + y \end{aligned}$$

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

(c) Determine whether the predicate I defined by

$$I \triangleq y \geq 2 \Rightarrow (1 \leq x \leq y)$$

is an invariant of the program.

(d) Consider the following temporal logic properties:

$$\begin{aligned} F &\triangleq \Box\Diamond\,(x = 2) & \qquad H &\triangleq x + y = 4 \leadsto x + y = 6 \\ G &\triangleq x = 2 \leadsto x = 3 & \qquad J &\triangleq \Box\Diamond\,(y = 3) \end{aligned}$$

Determine for each of F, G, H, and J whether the property 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.

Below, a monitor implementation of a *generalized semaphore* synchronization mechanism is shown. The mechanism has two operations which generalize the usual semaphore operations: $pass(k : posinteger)$ and $release(k : posinteger)$ where *posinteger* is the type of positive integers.

```
monitor GenSem

  var s : integer := 0;
      queue : condition;

  procedure pass(k : posinteger) {
    while s < k do wait(queue);
    s := s − k
  }

  procedure release(k : posinteger) {
    s := s + k;
    signal_all(queue)
  }

end
```

Question 4.1:

Specify the effect of the operations $pass(k)$ and $release(k)$ in the form of conditional atomic actions acting upon the state variable s.

Question 4.2:

(a) Prove that $I \triangleq s \geq 0$ is a monitor invariant of *GenSem*.

(b) Let $params_{op}(c)$ be a notation for the vector of parameters $\overline{p}$ of those calls of operation $op(p)$ which are currently waiting on condition queue c (in queuing order). Furthermore, for a non-empty vector of values $\overline{v}$, let $\min(\overline{v})$ and $\max(\overline{v})$ denote the minimum and maximum value in $\overline{v}$ respectively.

Define a monitor invariant J expressing that calls of *pass* do not wait unnecessarily. The expression $params_{pass}(queue)$ is expected to occur in J.

(c) Explain why *signal_all* in *release* cannot be replaced by *signal* in order to avoid unnecessary wakeups.

(d) Assume that the condition queue operations for prioritized waiting, $wait(c, rank)$ and $minrank(c)$, are available. Using these, show how the monitor may be modified in order to reduce unnecessary wakeups.

The problem is continued on the next page

Question 4.3:

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

module *GenSem*;
  op *pass*(k : *posinteger*);
  op *release*(k : *posinteger*);
end

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

Question 4.4:

In a system n ($n \geq 1$) *reader processes* and m ($m \geq 1$) *writer processes* are to be synchronized using the given module *GenSem*.

(a) Show how this may be accomplished by stating any operations which should be applied initially as well as the pre and post protocols for reading and writing.

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

Question 4.5:

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

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

Question 4.6:

The given monitor *GenRes* does not guarantee that all calls of *pass* are served. In particular a call of *pass*(k) may be overtaken by later, less demanding calls if the semaphore value s remains below the required k.

We now want a monitor *FCFS_GenSem* with the same operations as *GenSem*, but for which calls of *pass* are served in strictly first-come-first-served order.

Implement such a monitor.

[You may assume that condition queues are FIFO and spurious wakeups do not occur.]

PROBLEM 5 (approx. 15 %)

In a system certain critical decisions need to be carefully considered. This is done by letting a *decision problem* described by a data type D be forwarded to a panel of advisors for approval before action is taken.

The advisory panel consists of n ($n \geq 1$) concurrent *advisor processes* each of which is able to make a judgment of a problem using a specific approach (utilizing AI, applying rules, consulting a human expert, etc.).

To coordinate the decision making, a module *Resolve* is used. The module provides the following interface:

```
module Resolve
  op decide(D) returns boolean;
  op get(integer) returns (D, integer);
  op result(boolean, integer);
end
```

The operation *decide*(d) is called when a decision on the problem described by d has to be made. The operations *get* and *result* are called by the advisor processes in order to obtain a problem for judgment and return the result of that.

The calls of *decide* should be handled one by one. After having given the problem a number (see below), the server should pass it on to any advisors calling *get* that have not yet made a judgment of this problem. As soon as the constant number m ($1 \leq m \leq n$) of the advisors have returned an approval result for this problem, the call of *decide* should return. The *decide*-call should return true if and only if at least 2/3 of the m advisors have approved the problem.

In order to keep problems separated, these should be consecutively numbered by the coordinator starting with 1. An advisor calls *get*(l) with the number l of the last problem considered in order not to get that problem again (if it is still being decided). The result of *get*(l) is a new problem and its number. This number is used when returning the result. Thus, advisors repeatedly call the operations as follows:

```
process Advisor[i : 1..n];
  var d : D;
      no, last : integer := 0;
      approval : boolean;
  repeat
    (d, no) := Resolve.get(last);
    make judgment of d using approach i and obtain decision in approval
    Resolve.result(approval, no);  last := no;
    possibly do other work and prepare for next judgment
  forever;
```

It must always be possible for an advisor to return a result. If the problem has already been decided, the result should be discarded.

Question 5.1:

Write a server process for the module *Resolve* which services the operations by rendezvous in such a way that it functions as specified above.