

The exam is worth 100 points. There are 7 questions, and 8 pages. Answer all questions.

Please be brief, succinct, and to-the-point in your answers.

You will lose points for incorrect statements--even when your answer is correct!

- (5 pts) **(1)** Consider a set of sequential processes that cannot share any variables except semaphores. Can these processes communicate with each other? Explain your answer.
Hint: It will suffice to consider the transmission of a single bit.

- (20 pts) **(2)** In the class, we have seen two hardware-instruction-based solutions to concurrent programming, namely, test-and-set and swap instructions. They are designed to solve the mutual exclusion problem.

(i) Do they satisfy the progress property? If yes, explain. If not, give a scenario.

(ii) Do they satisfy the bounded waiting property? If yes, explain. If not, give a scenario.

(iii) Do they have livelocks? If yes, explain. If not, give a scenario.

(iv) Do they allow deadlocks? If yes, explain. If not, give a scenario.

(v) Choose one of the two hardware instructions, and explain in detail as to how (a) the instruction is defined, and (b) how it is used by giving the entry and exit code that accompany the instruction.

(5 pts) **(3)** Assume we have processes P1 and P2, each executing over its own independent address space, but with shared variables. Consider the following constraint: The code that P1 executes (in its own address space) has an instruction I1 that must always be executed *after* P2 executes in its own address space another instruction I2. Using *any* concurrent programming construct, how can you make sure that the constraint is satisfied?

(20 pts) (4) Semaphore primitives **wait** and **signal** are atomic operations, i.e., CPU cannot be interrupted when it is executing a semaphore primitive.

(a) Assume that you have a correct solution to a problem in which **Signal()** is executed atomically. Explain what goes wrong if the atomicity requirement is not followed for a non-blocking, binary semaphore **signal()** (where **signal(S)** is defined as $S := S + 1$).

(b) Assume that you have a correct solution to a problem in which **Signal()** is executed atomically. Explain what goes wrong if the atomicity requirement is not followed for a blocking, non-binary semaphore primitive **Signal(S)** (given below).

```
signal(S): S.value := S.value + 1;  
           if S.value ≤ 0  
             then begin  
               Remove a process P from S.L;  
               Wakeup(P);  
             end;
```

(15 pts) (5) Consider the monitor-based solution to the Dining Philosophers problem given below.

```

type dining-philosophers = monitor
  var state: array [0..4] of (thinking, hungry, eating);
  var self: array [0..4] of condition;

procedure entry pickup (i: 0..4)
begin
  state[i]:= hungry;
  test(i);
  if state[i] ≠ eating then self[i].wait;
end

procedure entry putdown(i: 0..4)
begin
  state[i]:= thinking;
  test ((i + 4) mod 5);
  test ((i + 1) mod 5);
end

procedure test (k: 0..4)
begin
  if state[(k+4) mod 5] ≠ eating and state[(k+1) mod 5] ≠ eating
    and state[k] = hungry ----- (*)
  then begin state [k]:= eating; self[k].signal end
end

begin
  for i:=0 to 4 do state[i] := thinking;
end.

```

Monitor use (process i):

```

Var dp: dining-philosophers;
repeat
  dp.pickup(i);
  EAT;
  dp.putdown(i);
  THINK;
until false;

```

(i) Assume philosopher 0 is *eating*, and all others are *thinking*. Assume philosopher 1 wants to eat, and gets to the CPU. What happens? State the values of variables at the beginning and at the end of philosopher 1 CPU execution. State where Philosopher 1 gets blocked.

(ii) Continuing with (i), assume philosopher 0 gets to the CPU, finishes eating, and successfully goes into the *thinking* state. What happens next? State the values of variables at the end of CPU executions of philosophers 0 and 1.

(iii) With the solution above, can you have a philosopher being *hungry* forever? (i.e., because of the scheduling algorithm of the CPU scheduler, the philosopher never switches from the state of *hungry*). Explain your answer.

(iv) With the solution above, can you have a deadlock (i.e., a circular wait of all philosophers)? Why or why not?

(v) Give a scenario explaining what goes wrong if the statement (*) above gets removed.

(20 pts) **(6) Fair-Readers-Writers problem:** When there are readers and writers waiting to read and write (due to a writer or reader in its CS), the service of the system becomes first-come-first-serve (FIFO) with the provision that consecutive FIFO readers are allowed to read concurrently. Consider the monitor-based solution below.

type Fair-Reader-Writer = **monitor**

Variable declarations:

Queue Q: Describes the readers/writers waiting to read/write.

Each record of Q contains R (for reader) or W (for writer).

Q has three procedures: Q.Push() Q.Pop() Q.GetNext(): Checks the top queue entry (but does not remove from the queue), and returns either R(eader), W(riter), or Null (empty queue)

Integer variables: NReading; WWaitCount

Boolean variables: Writing Condition variable: a

procedure entry open-read

begin

if (WWaitCount=0 **and not** writing)

then NReading++

else {

Q.Push(R);

a.wait;

Q.Pop(); ----- (*)

NReading++;

if Q.GetNext()=R **then** a.signal;

}

end.

procedure entry close-read

begin

NReading--;

if (NReading=0)

then a.signal;

end.

begin

Q ← *Empty*;

NReading:=WWaitCount:=0; Writing:=*False*;

end;

Reader use:

var Fair-RW: **Fair-Reader-Writer**;

Fair-RW.open-read();

READ;

Fair-RW.close-read();

Procedure entry open-write

begin

if (NReading >0 **or** Writing)

then {

Q.Push(W);

WWaitCount++;

a.wait;

Q.Pop();

WWaitCount--;

}

Writing:=*True*;

end.

procedure entry close-write

begin

Writing := *False*;

a.signal; ----- (**)

end.

Writer use:

var Fair-RW: **Fair-Reader-Writer**;

Fair-RW.open-write();

WRITE;

Fair-RW.close-write();

(a) Assume one reader, R1, is in its CS, reading. What are the values of variables NReading and Writing?

- (b) Assume, while R1 is executing in its CS, additional readers and writers arrive in the given order:
W1, R2, R3.
i.e., writer W1 arrives, executes and blocks; reader R2 arrives, executes, and blocks, reader R3 arrives, executes and blocks. State where each process blocks, and, at the end of R2 execution, the values of NReading and Writing, and the content of the queue Q.
- (c) Continuing with (b), assume that R1 executes, leaves its CS, successfully executes close-read(), and gets removed from the CPU by the CPU scheduler. What happens next to the blocked processes?
- (d) What goes wrong if you remove the statement (*)?
- (e) Assume that there are no processes in the monitor or in their critical sections. Assume that a single writer comes, executes open-write(), writes, and executes close-write(). What happens when the writer executes the statement (**)?

(15 pts) (7) Consider the Bakery algorithm given below.

```
Pi: repeat
    choosing[i] := true;
    number[i] := max (number[0], number[1], ..., number[n-1])+1; --- (*)
    choosing[i] := false;
    for j:=0 to n-1 do begin
        while choosing[j] do skip;
            while (number[j], j) < (number[i], i)
                and number[j] ≠ 0 ----- (**)
                do skip;
    end
    Critical Section
    number[i] := 0;
    Remainder Section
until false;
```

Initially, all choosing values are false, and all number values are zeroes.

(a) Assume that the line (*) is removed. Describe a scenario in which the algorithm does not work (i.e., it does not solve the critical section problem). Briefly, explain what goes wrong and why.

(b) Assume that the line (**) is removed (and line (*) stays). Describe a scenario in which the algorithm does not work (i.e., it does not solve the critical section problem). Briefly, explain what goes wrong and why.