The Australian National University
Mid Semester Examination – August 2018

Comp2310 & Comp6310
Systems, Networks and Concurrency

Study period:	15 minutes
Time allowed:	1.5 hours (after study period)
Total marks:	50
Permitted materials:	None

Questions are **not** equally weighted – sizes of answer boxes do **not** necessarily relate to the number of marks given for this question.

All your answers must be written in the boxes provided in this booklet. You will be provided with scrap paper for working, but only those answers written in this booklet will be marked. Do not remove this booklet from the examination room. There is additional space at the end of the booklet in case the boxes provided are insufficient. Label any answer you write at the end of the booklet with the number of the question it refers to.

Greater marks will be awarded for answers that are simple, short and concrete than for answers of a sketchy and rambling nature. Marks will be lost for giving information that is irrelevant to a question.

Student number:

The following are for use by the examiners

Q1 mark	*Q2 mark*	*Q3 mark*

Total mark

1. [13 marks] General concurrency

(a) [3 marks] Assume that all your CPUs are currently executing one process each. What possible next states could any of those tasks transition to? Also explain how the next state is decided and what hardware is used to make this decision.

(b) [2 marks] If a task is currently blocked, name at least two things which need to happen before this task will be running in a CPU again.

(c) **[5 marks] Is the following statement correct?: "If all processes are executed on a single CPU (as a single, sequential stream of instructions), then all data access is automatically mutually exclusive". Provide reasons for your answer.**

(d) **[3 marks] Which if the following statements are correct? Tick all correct statements – marks will be subtracted for wrongly ticked statements, so do not just tick all of them.**

- ☐ One can use one or multiple atomic test-and-set operation(s) to implement a semaphore.
- ☐ One can use one or multiple semaphore(s) to implement a protected object.
- ☐ One can use a single protected object to implement a semaphore.
- ☐ One can use a single semaphore to implement an atomic test-and-set operation in software.
- ☐ An atomic test-and-set operation is syntactically bound to shared data or critical regions.
- ☐ A semaphore is syntactically bound to shared data or critical regions.
- ☐ A protected object is syntactically bound to shared data or critical regions.
- ☐ The total number of occurring `wait` operations and the total number of occurring `signal` operations on a semaphore can be different.

2. [31 marks] Synchronization

(a) [8 marks] You want to limit the number of tasks which can access a specific resource simultaneously to 10. Provide code which can be used to gain access to this resource. You can provide your answer in any programming language of your choice, including machine code or pseudo code. Safer solutions will gain higher marks.

(b) Read the following Ada program carefully. The program is syntactically correct and will compile without warnings. See comments and questions on the following page.

```
with Ada.Text_IO; use Ada.Text_IO;
with Id_Dispenser;

procedure Round_Robin is

   type Nodes is mod 5;

   protected Scheduler is

      entry Start (Nodes);
      procedure Stop;

   private
      Turn : Nodes := Nodes'First;

   end Scheduler;

   protected body Scheduler is

      entry Start (for n in Nodes) when Turn = n is

      begin
         null;
      end Start;

      procedure Stop is

      begin
         Turn := Nodes'Succ (Turn);
      end Stop;

   end Scheduler;

   package Node_Id_Dispenser is new Id_Dispenser (Element => Nodes);
   use Node_Id_Dispenser;

   Sum_A, Sum_B : Natural := 0;

   task type Node;
   task body Node is

      Id : constant Nodes := Draw_Id;

   begin
      Sum_A := Sum_A + Natural (Id);

      delay 1.0; -- second
      Scheduler.Start (Id);

      Sum_B := Sum_B + Natural (Id);
      Put_Line ("Task" & Nodes'Image (Id) & " reports:" &
                Natural'Image (Sum_A) & Natural'Image (Sum_B));
      Scheduler.Stop;
   end Node;

   Pool : array (Nodes) of Node; pragma Unreferenced (Pool);

begin
   null;
end Round_Robin;
```

The `pragma Unreferenced` prevents a compiler warning, which would point out that `Pool` is not referenced in this program. Note that `entry Start (Nodes)` declares a family of entries. The package `Id_Dispenser` and the function `Draw_Id` works as you would expect them to (for example from your lab exercises). `Draw_Id` will provide unique ids of the type `Nodes`.

(i) [2 marks] How many concurrent entities are implemented by this program? Name them.

(ii) [3 marks] Is this program deterministic? Give precise reasons for your answer.

(iii) [4 marks] Will this program terminate always, sometimes, or never? Give precise reasons for your answer. If you think that some tasks will terminate while others won't, then also enumerate those non-terminating tasks.

(iv) [6 marks] What output (or multiple possible outputs) would you expect from running this program? If you found the output to be non-deterministic, then do not write out all possible outputs, but provide rules which describe the possible outputs (for instance: "the printed value for `Sum_A` will always be the negative value of the local task id" or "output line *a* will always appear before line *b*").

(c) Read the following Ada program carefully. The program is syntactically correct and will compile without warnings. See comments below and questions on the following page.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Task_Scopes is

   task type Dynamic_Outer_Task;
   type Outer_Task_Ptr is access Dynamic_Outer_Task;
   task body Dynamic_Outer_Task is

   begin
      Put_Line ("Dynamic_Outer_Task");
   end Dynamic_Outer_Task;

   task Static_Outer_Task;
   task body Static_Outer_Task is

   begin
      Put_Line ("Start of Static_Outer_Task");
      declare

         task Static_Inner_Task;
         task body Static_Inner_Task is

         begin
            Put_Line ("Start of Static_Inner_Task");
            declare
               Outer_Instance : Outer_Task_Ptr := new Dynamic_Outer_Task;
               pragma Unreferenced (Outer_Instance);
            begin
               null;
            end;
           Put_Line ("End of Static_Inner_Task");
         end Static_Inner_Task;

      begin
         null;
      end;
      Put_Line ("End of Static_Outer_Task");
   end Static_Outer_Task;

begin
   Put_Line ("Task_Scopes");
end Task_Scopes;
```

The `pragma Unreferenced` prevents a compiler warning which would point out that `Outer_Instance` is not referenced in this program.

(i) [3 marks] Explain at least one purpose of dynamic task allocations.

(ii) [5 marks] Which if the following statements are correct? Tick all correct statements – marks will be subtracted for wrongly ticked statements, so do not just tick all of them.

☐ "`Start of Static_Outer_Task`" will always appear before "`Task_Scopes`".

☐ "`Start of Static_Outer_Task`" will always appear before "`Start of Static_Inner_Task`".

☐ "`Start of Static_Inner_Task`" will always appear before "`Dynamic_Outer_Task`".

☐ "`Dynamic_Outer_Task`" will always appear before "`End of Static_Inner_Task`".

☐ "`Dynamic_Outer_Task`" will always appear before "`End of Static_Outer_Task`".

☐ "`Dynamic_Outer_Task`" will always appear before "`Task_Scopes`".

☐ "`Task_Scopes`" can appear anywhere in the output of this program.

☐ "`Dynamic_Outer_Task`" can appear anywhere in the output of this program.

☐ "`Start of Static_Inner_Task`" will always appear before "`End of Static_Inner_Task`".

☐ "`End of Static_Inner_Task`" will always appear before "`End of Static_Outer_Task`".

☐ "`Outer_Instance`" will live at least as long as the task which it refers to.

3. [6 marks] Message Passing

(a) Read the following Ada program carefully. The program is syntactically correct and will compile without warnings. See comments and questions on the following page.

```
with Id_Dispenser_Task;

procedure Factorial_Pool is

   No_of_Servers : constant Positive := 10;
   No_of_Clients : constant Positive := 100;

   type Nodes is mod No_of_Servers;

   package Node_Id_Dispenser is new Id_Dispenser_Task (Element => Nodes);
   use Node_Id_Dispenser;

   task type Server is
      entry Calc_Factorial (n : Natural; Result : out Positive);
   end Server;

   task type Client;

   Servers : array (Nodes) of Server;
   Clients : array (1..No_of_Clients) of Client; pragma Unreferenced (Clients);

   task body Server is

      Id : Nodes := Nodes'Invalid_Value;

      function Factorial (n : Natural) return Positive is
        (if n <= 1 then 1 else n * Factorial (n - 1));

   begin
      Dispenser.Draw_Id (Id);
      loop
         accept Calc_Factorial (n : Natural; Result : out Positive) do
            if n mod No_of_Servers = Natural (Id) then
               Result := Factorial (n);
            else
               requeue Servers (Nodes (n mod No_of_Servers)).Calc_Factorial;
            end if;
         end Calc_Factorial;
      end loop;
   end Server;

   task body Client is

      Result : Positive := Positive'Invalid_Value;

   begin
      for i in 0 .. 10 loop
         Servers (Servers'First).Calc_Factorial (i, Result);
      end loop;
   end Client;

begin
   null;
end Factorial_Pool;
```

The `pragma Unreferenced` prevents a compiler warning which would point out that `Clients` is not referenced in this program. A `requeue` operation in the context of message passing works exactly the same as in the context of protected operations.

(i) [2 marks] Will this program terminate? Provide reasons in either case. If it does not terminate, name the tasks which do not terminate.

(ii) [4 marks] Explain how concurrency is used in this program. Does it provide any performance benefits? If you think it does, provide an estimate for the expected speed-up (and state your assumption). Provide reasons for you answer in either case.

continuation of answer to question [ ] *part* [ ]

continuation of answer to question [ ] *part* [ ]

continuation of answer to question [ ] *part* [ ]

continuation of answer to question [ ] *part* [ ]