The Australian National University
Final Examination – November 2018

Comp2310 & Comp6310 Systems, Networks and Concurrency

Study period:	15 minutes
Writing time:	3 hours (after study period)
Total marks:	100
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 (and also note inside the original answer box that your answer is continued at the end of the booklet).

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*	*Q4 mark*	*Q5 mark*	*Q6 mark*	*Total mark*

1. [13 marks] General Concurrency

Which of 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. *If* you find a statement to be *incorrect*, *then* provide a corrected version of that statement in the answer box underneath.

☐ All **concurrent programming languages** are capable of providing errors or warnings with respect to concurrent operations.

☐ **Rigorous testing** guarantees the correctness of concurrent programs.

☐ **Non-deterministic programs** cannot be correct.

☐ A **fail-safe system** is free of failures.

☐ A **full fault tolerant system** will run forever.

☐ **Deadlock prevention** prevents all forms of deadlocks.

☐ **Race conditions** will always result in **non-deterministic program behavior**.

☐ **Non-deterministic program behavior** will always result in **race conditions.**

☐ If A and B are events **in the same task** and the logical times $C(A)$ and $C(B)$ are in order: $C(A) < C(B)$, then A must have happened earlier than B (in real time).

☐ If A and B are events **in different tasks** and the logical times $C(A)$ and $C(B)$ are in order: $C(A) < C(B)$, then A must have happened earlier than B (in real time).

☐ If A and B happened **concurrently,** then the logical times $C(A)$ and $C(B)$ must be equal: $C(A) = C(B)$.

☐ **Interrupt handlers** have to run on special hardware.

☐ CPU states will be stored by special hardware **when an interrupt occurs**.

2. [20 marks] Synchronization and Communication

(a) [6 marks] Implement a semaphore in a programming language of your choice. Identify the specific language features which you rely on in your implementation (if any).

(b) [6 marks] In the context of concurrent programming explain what is meant by a race condition? Also provide an example in 20 lines or less of pseudo code that shows a race condition.

(c) [8 marks] Can synchronous and asynchronous message passing systems simulate each other? Provide a solution or a reason (if you think that this would not be possible) in each case. If you provide a solution, then also mention potential limitations of your solution.

3. [18 marks] Data Parallelism

(a) [12 marks] Read this syntactically correct Chapel expression and then proceed to the questions below:

```
sqrt (+ reduce ((Vector_1 - Vector_2)**2))
```

where you should assume the following declarations for `Vector_1` and `Vector_2`:

```
const Index = {1 .. 1000};
var Vector_1, Vector_2 : [Index] real;
```

(i) [6 marks] What would be a hardware architecture which can execute this Chapel expression the fastest? For your suggested hardware architecture: provide a diagram of all parallel hardware entities and their connection.

(ii) [6 marks] Given the ideal hardware, what would be the computational time complexity (meaning wall clock time complexity in relation to vector length) which would be required to execute this Chapel expression? What is the computational time complexity for a sequential execution of this Chapel expression?

(b) [6 marks] Write a program to implement the discrete cross-correlation function (as a discrete array) between two cyclic, discrete functions (which are themselves represented by discrete arrays) which optimizes for performance on an 8-core CPU with vector processing units (processing 8 16-bit integer numbers per vector operation):

$$Cross\_Correlation(A,B)_k = \sum_i (A_i \cdot B_{i+k})$$

Sequentially such a function could be implemented like this:

```
subtype Input_Range  is Integer range -(2**15) .. +(2**15 - 1);
subtype Output_Range is Integer range -(2**31) .. +(2**31 - 1);

type Samples is mod 2**16;

type Input_Function  is array (Samples) of Input_Range;
type Output_Function is array (Samples) of Output_Range;

function Cross_Correlation (A, B : Input_Function) return Output_Function is

   CC : Output_Function := (others => 0);

begin
   for k in Samples loop
      for i in Samples loop
         CC (k) := CC (k) + A (i) * B (i + k);
      end loop;
   end loop;
   return CC;
end Cross_Correlation;
```

Use any programming language of your choice (including pseudocode). State what you assume about your compiler.

4. [10 marks] Scheduling

(a) [6 marks] Name three different criteria by which you can evaluate the performance of a scheduling algorithm and name a scheduling algorithm in each case which optimizes for this criterion.

Criterion	Scheduling algorithm

(b) [4 marks] Explain how Feedback scheduling with 2^i pre-emption intervals works. Name one major advantage and one major drawback of this scheduling algorithm.

5. [16 marks] Safety & Liveness

Read the following first part on an Ada program carefully. The whole program is syntactically correct and will compile without warnings. See questions on the following pages.

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

procedure Synced_Processes is

   No_Of_Clients : constant Positive := 10;

   type Resource_Range  is range 1 .. 5;
   type Instance_Range  is range 0 .. 5;
   type Instances_Available is array (Resource_Range'Range) of Instance_Range;

   protected Resources is

      entry     Aquire (Resource_Range);
      procedure Release (Ix : Resource_Range);
      procedure Client_Terminates;
      entry     Wait_For_Deadlock_Or_Termination (Deadlocked : out Boolean);

   private
      function No_Of_Waiting_Clients return Natural;
      Instances            : Instances_Available := (others => Instance_Range'Last);
      No_Of_Active_Clients : Natural             := No_Of_Clients;
   end Resources;

   protected body Resources is

      entry Aquire (for Ix in Resource_Range) when Instances (Ix) > 0 is

      begin
         Instances (Ix) := Instances (Ix) - 1;
      end Aquire;

      procedure Release (Ix : Resource_Range) is

      begin
         Instances (Ix) := Instances (Ix) + 1;
      end Release;

      procedure Client_Terminates is

      begin
         No_Of_Active_Clients := No_Of_Active_Clients - 1;
      end Client_Terminates;

      entry Wait_For_Deadlock_Or_Termination (Deadlocked : out Boolean)
        when No_Of_Waiting_Clients = No_Of_Active_Clients is

      begin
         Deadlocked := No_Of_Active_Clients > 0;
      end Wait_For_Deadlock_Or_Termination;

      function No_Of_Waiting_Clients return Natural is

         function Sum_of_Counts (Ix : Resource_Range) return Natural is
           (Aquire (Ix)'Count + (if Ix = Resource_Range'Last then 0
                                 else Sum_of_Counts (Ix + 1)));

      begin
         return Sum_of_Counts (Resource_Range'First);
      end No_Of_Waiting_Clients;

   end Resources;
```

(i) [4 marks] The protected object `Resources` offers (besides `Aquire` and `Release` of resource instances) a simple deadlock detection feature. It is assumed that the initial `No_Of_Clients` which claim (will try to allocate) a resource is known and that every client which has released all its resources and no longer claims any resources will call `Client_Terminates`. Describe how the implemented deadlock detection mechanism works.

(ii) [4 marks] Will all possible deadlocks be detected by this simple mechanism? Give precise reasons in either case.

Now study the second part of this program carefully and read the questions on the next page. Note that the resources are acquired in each client in reverse (descending) order, while they are released in ascending order.

```
   task type Client;
   task body Client is

      No_Of_Claimed_Instances : constant Positive := 1; -- use 2 for part (iv)

   begin
      for Ix in reverse Resource_Range loop
         for Instance in 1 .. No_Of_Claimed_Instances loop
            Resources.Aquire (Ix);
         end loop;
      end loop;

      for Ix in Resource_Range loop
         for Instance in 1 .. No_Of_Claimed_Instances loop
            Resources.Release (Ix);
         end loop;
      end loop;

      Resources.Client_Terminates;
   end Client;

   Clients     : array (1 .. No_Of_Clients) of Client;
   Deadlocked  : Boolean;

begin
   Resources.Wait_For_Deadlock_Or_Termination (Deadlocked);
   if Deadlocked then
      Put_Line ("---  Deadlock detected, aborting clients ---");
      for c of Clients loop
         abort c;
      end loop;
   else
      Put_Line ("--- All clients terminated normally ---");
   end if;
end Synced_Processes;
```

(iii) [4 marks] Will the program terminate, deadlock, or livelock? Give reasons. What are the possible displays on the terminal by this program. If the program is found to be non-deterministic, discuss all possible outcomes.

(iv) [4 marks] If you replace the value of the constant `No_Of_Claimed_Instances` with `2`, will the program still terminate, deadlock, or livelock in the same way? Give reasons. What are the possible displays on the terminal by this program. If the program is found to be non-deterministic, discuss all possible outcomes.

6. [23 marks] Distributed Systems

(a) [10 marks] Serializable transactions

(i) [4 marks] Why is it desirable to have transactions serializable? Could you execute them sequentially to achieve serializability? Give reasons.

(ii) [6 marks] Describe how you can detect at runtime that two transactions are not serializable.

(b) [13 marks] Read the following Ada program carefully. The whole program is syntactically correct and will compile without warnings. See questions on the following pages.

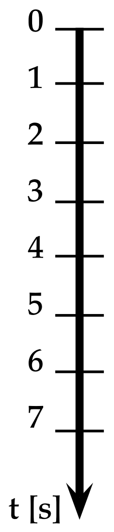
```
with Ada.Text_IO; use Ada.Text_IO;

procedure Distributed_System is

   type Workers_Range is range 1 .. 3;
   type Clients_Range is range 1 .. 7;

   task Server is
      entry Report (w : Workers_Range);
      entry Service;
   private
      entry Hold;
      entry Forward;
   end Server;

   task type Worker is
      entry Identify (Id : Workers_Range);
      entry Service;
   end Worker;

   Workers : array (Workers_Range) of Worker;

   task body Server is

      type State  is (Available, Busy);
      type States is array (Workers_Range) of State;

      All_Workers_Busy : constant States := (others => Busy);
      Workers_State : States := All_Workers_Busy;

   begin
      loop
         select
            accept Report (w : Workers_Range) do
               Workers_State (w) := Available;
            end Report;
         or when Forward'Count = 0 =>
               accept Service do
                  if (for some w of Workers_State => w = Available) then
                     requeue Forward;
                  else
                     requeue Hold;
                  end if;
               end Service;
         or when Forward'Count = 0 and then Workers_State /= All_Workers_Busy =>
               accept Hold do
                  requeue Forward;
               end Hold;
         or accept Forward do
               for i in Workers_Range loop
                  if Workers_State (i) = Available then
                     Workers_State (i) := Busy;
                     requeue Workers (i).Service;
                  end if;
               end loop;
            end Forward;
         or
            terminate;
         end select;
      end loop;
   end Server;                                          --  (continued on next page ..)
   task body Worker is

      Worker_Id : Workers_Range;

   begin
      accept Identify (Id : Workers_Range) do
         Worker_Id := Id;
      end Identify;

      Server.Report (Worker_Id);

      loop
         select
            accept Service;
            Put (" W" & Workers_Range'Image (Worker_Id) & " ");
            delay 1.0; -- interesting, hard work being done here.
         or
            terminate;
         end select;

         Server.Report (Worker_Id);
      end loop;
   end Worker;

   task type Client;
   task body Client is

   begin
      Server.Service;
   end Client;

   Clients : array (Clients_Range) of Client; pragma Unreferenced (Clients);

begin
   for w in Workers_Range loop
      Workers (w).Identify (Id => w);
   end loop;
end Distributed_System;
```

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

(i) [2 marks] Where potentially could a task in this program become blocked? Name all blocking options.

(ii) [3 marks] How is concurrency used in this program? What benefit does this provide?

(iii) [4 marks] On the following time-line provide the output which you expect from this program? If the output is non-deterministic, describe all options. Consider zero seconds to be the start-time of the program.



(iv) [4 marks] Provide reasons why you think the program is deterministic or non-deterministic. If you find the program to be non-deterministic: what is the exact impact of this non-determinism on the overall program behavior? Will it always terminate, livelock or deadlock?

continuation of answer to question | | *part* | |

continuation of answer to question | | *part* | |

continuation of answer to question | | *part* | |

continuation of answer to question | | *part* | |

continuation of answer to question | | *part* | |

continuation of answer to question | | *part* | |