The Australian National University
Final Examination – November 2015

Comp2310 & Comp6310
Concurrent and Distributed Systems

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.

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. [16 marks] General Concurrency

(a) [3 marks] Give three examples of hardware components that support concurrency.

(b) [5 marks] What will be the output (or the possible outputs) of the following concurrent program? Give precise reasons for your answer. If you need to make assumptions about the underlying operating system, runtime environment or hardware then state those assumptions as well.

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

procedure x_Value is

   x : Positive := 1;

   task One;
   task body One is

   begin
      x := x + x;
   end One;

   task Two;
   task body Two is

   begin
      x := x + x;
   end Two;

begin
   Put (Positive’Image (x));
end x_Value;
```

(c) [3 marks] How does a block of Dijktra's guarded commands differ from a `switch` statement as you find it in Java or C? Name at least two essential differences.

(d) [3 marks] How does a `forall` statement (as in Chapel) differ from a `for` statement as you find it in Java or C? Be as precise as you can.

(e) [2 marks] If you define a boolean expression which is true when all your concurrent processes are terminated could such an expression be regarded as a safety or a liveness property? Do all concurrent programs have to fulfil such a termination condition? Give precise reasons for both answers.

2. [14 marks] Synchronization and Communication

(a) [5 marks] Can synchronous message passing systems emulate asynchronous message passing? Can asynchronous message passing systems emulate synchronous message passing? For both questions provide either a reason why this is not possible or a diagram which is depicting how it could work. If it is only possible under certain assumptions then also state those assumptions.

(b) [3 marks] Define a general semaphore (as offered by an operating system) precisely. Only the defining characteristics are asked for here – not the additional convenience functions which might be offered as well by an operating system.

(c) [2 marks] Can you program a semaphore if your programming language does not provide one? If you need to make assumptions to answer this question then explain them here as well.

(d) [4 marks] Which problematic issues with semaphores are resolved by monitors? Which issues are not? Explain at least one resolved and one unresolved issue.

3. [9 marks] Selective Concurrency

(a) [9 marks] Read the following Ada code carefully. The tasks and the calling code section are syntactically correct and will compile without warnings.

```
task Selector is
   entry Start;
   entry E1;
   entry E2;
end Selector;
```

with three different versions for its body (all delay values are in seconds):

Version 1:

```
task body Selector is

begin
   accept Start;

   for i in 1 .. 2 loop

      select

         accept E1 do
            delay 2.0;
            Put (‘X’);
         end E1;

      or
         accept E2;
         delay 2.0;
         Put (‘Y’);

      or
         terminate;

      end select;

      delay 1.0;
      Put (‘Z’);

   end loop;

end Selector;
```

Version 2:

```
task body Selector is

begin
   accept Start;

   for i in 1 .. 2 loop

      select

         accept E1;
         delay 2.0;
         Put (‘X’);
      or
         delay 2.0;
         accept E2;
         Put (‘Y’);
         exit;

      end select;

      delay 1.0;
      Put (‘Z’);

   end loop;

end Selector;
```

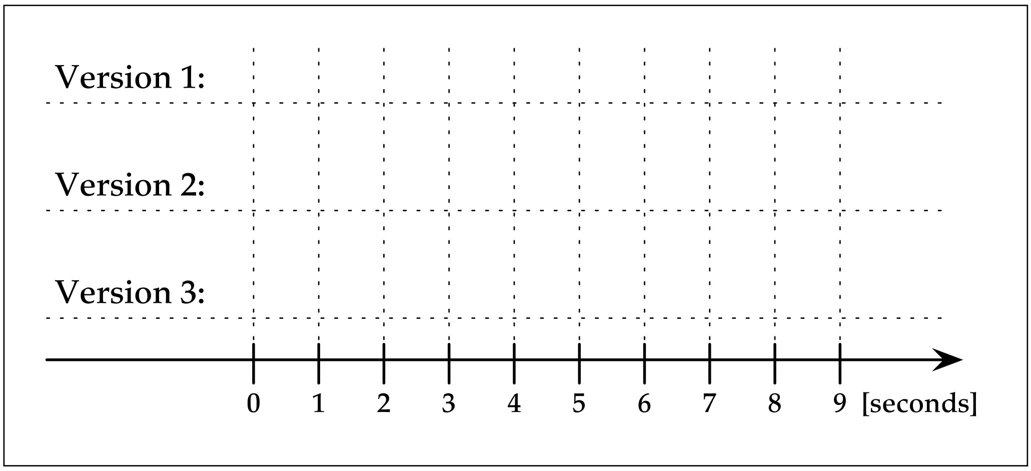
Version 3:

```
task body Selector is

begin
   accept Start;

   for i in 1 .. 2 loop

      select

         accept E1 do
            delay 2.0;
            Put (‘X’);
         end E1;

      else
         accept E2;
         delay 2.0;
         Put (‘Y’);
         exit;

      end select;

      delay 1.0;
      Put (‘Z’);

   end loop;

end Selector;
```

Called by this code section:

```
Selector.Start;

Put (‘A’);
delay 1.0;

select
   Selector.E1;
   Put (‘B’);
else
   Put (‘C’);
end select;

delay 1.0;
Selector.E2;

delay 1.0;
Put (‘D’);
```

(i) [9 marks] Add the outputs for all three versions to the time lines below (assume that `Start` is called at time zero):



4. [17 marks] Safety and Liveness

(a) [17 marks] Read the following Ada package carefully. The package is syntactically correct and will compile without warnings.

See explanation and questions after the program code.

```
generic
   type Element is private;
   Size : Positive;

package Stack_with_Semaphores_Generic is

   type Stack_Type is limited private;

   procedure Push (Item :     Element; Stack : in out Stack_Type);
   procedure Pop  (Item : out Element; Stack : in out Stack_Type);

   function Is_Empty (Stack : in out Stack_Type) return Boolean;
   function Is_Full  (Stack : in out Stack_Type) return Boolean;

private

   protected type Semaphore (Initial : Natural := 0) is

      entry Wait;
      procedure Signal;

   private
      Value : Natural := Initial;

   end Semaphore;

   type List is array (1 .. Size) of Element;

   type Stack_Type is record
      Top        : Natural := 0;
      Elements   : List;
      Write_Lock : Semaphore (1);
      Read_Lock  : Semaphore (1);
      Is_Empty   : Semaphore (0);
      Is_Full    : Semaphore (Size);
      Readers    : Natural := 0;
   end record;

end Stack_with_Semaphores_Generic;


package body Stack_with_Semaphores_Generic is

   protected body Semaphore is

      entry Wait when Value > 0 is

      begin
         Value := Natural’Pred (Value);
      end Wait;

      procedure Signal is

      begin
         Value := Natural’Succ (Value);
      end Signal;
   end Semaphore;
```

(continued on next page)

```
   procedure Push (Item : Element; Stack : in out Stack_Type) is

   begin
      Stack.Write_Lock.Wait;
      Stack.Is_Full.Wait;

      Stack.Top := Positive'Succ (Stack.Top);
      Stack.Elements (Stack.Top) := Item;

      Stack.Is_Empty.Signal;
      Stack.Write_Lock.Signal;
   end Push;

   procedure Pop (Item : out Element; Stack : in out Stack_Type) is

   begin
      Stack.Write_Lock.Wait;
      Stack.Is_Empty.Wait;

      Item      := Stack.Elements (Stack.Top);
      Stack.Top := Positive'Pred (Stack.Top);

      Stack.Is_Full.Signal;
      Stack.Write_Lock.Signal;
   end Pop;

   procedure Start_Read (Stack : in out Stack_Type) is

   begin
      Stack.Read_Lock.Wait;
      if Stack.Readers = 0 then
         Stack.Write_Lock.Wait;
      end if;
      Stack.Readers := Natural'Succ (Stack.Readers);
      Stack.Read_Lock.Signal;
   end Start_Read;

   procedure Stop_Read (Stack : in out Stack_Type) is

   begin
      Stack.Read_Lock.Wait;
      Stack.Readers := Natural'Pred (Stack.Readers);
      if Stack.Readers = 0 then
         Stack.Write_Lock.Signal;
      end if;
      Stack.Read_Lock.Signal;
   end Stop_Read;

   function Is_Empty (Stack : in out Stack_Type) return Boolean is

   begin
      Start_Read (Stack);
      Stop_Read (Stack);
      return Stack.Top < Stack.Elements'First;
   end Is_Empty;

   function Is_Full (Stack : in out Stack_Type) return Boolean is

   begin
      Start_Read (Stack);
      Stop_Read (Stack);
      return Stack.Top = Stack.Elements'Last;
   end Is_Full;

end Stack_with_Semaphores_Generic;
```

The package is intended to provide concurrency-safe access to a stack where the operations `Push` and `Pop` are mutually exclusive to all other operations and the side-effect free (with respect to the stack data) operations `Is_Empty` and `Is_Full` can be executed by multiple tasks concurrently (unless `Push` or `Pop` are currently executing).

(i) [3 marks] Explain the usage and the meaning of the initialization values of the individual semaphores. What minimum and maximum values can each Semaphore reach?

(ii) [3 marks] Can any data be accessed in an unsynchronized way? If so, point out where this can happen and how you would prevent this.

(iii) [3 marks] Are there any possibilities for deadlocks in the operations `Push`, `Pop`, `Is_Empty` or `Is_Full`? If so, point out where this could happen and how you would prevent this.

(iv) [8 marks] Write a package with the same functionality, which is deadlock-free, synchronizes all access to shared data, distinguishes read and write access and is less than half the length of the given package. Safety and liveness properties of your version should be obvious or easy to check.

5. [19 marks] Data Parallelism

(a) [8 marks] Read this syntactically correct Chapel expression and then proceed to the questions below:

```
|| reduce (Vector_1 != Vector_2)
```

where you should assume the declarations:

```
const Index = {1 .. 100000000};
var Vector_1, Vector_2 : [Index] real;
```

(i) [1 mark] What is the type of this expression?

(ii) [5 marks] Enumerate and explain the data parallel operations which are implemented by this Chapel expression.

(iii) [2 marks] How many concurrent entities (tasks, processes, threads or alike) are created with this expression? Give reasons for your answer.

(b) [11 marks] Blocking operations are commonly kept at a minimum in high performance applications.

(i) [6 marks] Explain how a shared queue data structure can be implemented such that some/all interferences between readers and writers of such a queue can be avoided. Briefly outline a possible implementation.

(ii) [5 marks] Can blocking/synchronization operations be completely avoided in some concurrent programs? Give an example if you think this is the case and explain which sorts of applications could be implemented without any blocks/synchronizations? If you think that this is not the case then explain why blocking/synchronization is always necessary.

6. [25 marks] Distributed Systems

(a) [5 marks] What can you conclude about the events a and b (including whether they happened on the same or on different processors) if the relations between the logical times $C(a)$ and $C(b)$ associated with these events are:

(i) [1 mark] $C(a) \neq C(b)$

(ii) [1 mark] $C(a) = C(b)$

(iii) [1 mark] $C(a) > C(b)$

(iv) [2 marks] Is it true that if $C(a) < C(b)$ then there always exists an event c, such that: $C(a) < C(c) < C(b)$? Will your answer change if you measure time in calendar (or "real") time instead of logical time? Give precise reasons for your answers.

(b) [20 marks] Read the following Ada program carefully. The program is syntactically correct and will compile without warnings. See questions on the following pages. This first page contains only definitions for the sequential part of the program and you can ignore it when analyzing the concurrent aspects.

```
with Ada.Containers.Vectors; use Ada.Containers;
with Ada.Text_IO;            use Ada.Text_IO;

procedure Mini is

   No_Of_Stages   : constant Positive := 8;
   No_Of_Elements : constant Positive := 2 ** No_Of_Stages;

   subtype Element is Natural;

   type Element_Array is array (Natural range <>) of Element;

   function Is_Sorted (D : Element_Array) return Boolean is
     (for all i in D'First .. D'Last - 1 => D (i) <= D (i + 1));

   function Is_Permutation (Field_A, Field_B : Element_Array) return Boolean is

      package Element_Vectors is
                          new Vectors (Positive, Element); use Element_Vectors;
      package Sorting         is new Generic_Sorting;     use Sorting;

      Vector_A, Vector_B : Vector := Empty_Vector;

   begin
      for A of Field_A loop
         Append (Vector_A, A);
      end loop;
      for B of Field_B loop
         Append (Vector_B, B);
      end loop;
      Sort (Vector_A);
      Sort (Vector_B);
      return Vector_A = Vector_B;
   end Is_Permutation;

   function Merge (A, B : Element_Array) return Element_Array is

     (if    A'Length = 0 then B
      elsif B'Length = 0 then A
      elsif A (A'First) < B (B'First)
      then A (A'First) & Merge (A (Natural'Succ (A'First) .. A'Last), B)
      else B (B'First) & Merge (A, B (Natural'Succ (B'First) .. B'Last)))

   with Pre  => Is_Sorted (A) and then Is_Sorted (B),
        Post => Is_Sorted (Merge'Result) and then
                                   Is_Permutation (Merge'Result, A & B);

   subtype Stage_Range is Natural range 0 .. No_Of_Stages - 1;
```

(continued on next page)

```
   task type Stage is
      entry Hand_over_Id (Set_Id : Stage_Range);
      entry Feed         (E      : Element);
   end Stage;

   Stages : array (Stage_Range) of Stage;

   task body Stage is

      Id : Stage_Range := Stage_Range'Invalid_Value;

   begin
      accept Hand_over_Id (Set_Id : Stage_Range) do
         Id := Set_Id;
      end Hand_over_Id;

      declare
         type Channels is (Left, Right);

         Feeds : array (Channels) of Element_Array (1 .. 2 ** Natural (Id));

      begin
         loop
            for Ch in Channels loop
               for F of Feeds (Ch) loop
                  select
                     accept Feed (E : Element) do
                        F := E;
                     end Feed;
                  or
                     terminate;
                  end select;
               end loop;
            end loop;

            declare
               Merged_Feed : constant Element_Array :=
                                             Merge (Feeds (Left), Feeds (Right));

            begin
               if Id < Stage_Range'Last then
                  for M of Merged_Feed loop
                     Stages (Id + 1).Feed (M);
                  end loop;
               else
                  Put_Line ("Pipeline output is " &
                  (if Is_Sorted (Merged_Feed) then "sorted" else "not sorted"));
               end if;
            end;
         end loop;
      end;

   end Stage;

begin
   for Id in Stage_Range loop
      Stages (Id).Hand_over_Id (Id);
   end loop;
   for E in reverse 1 .. No_Of_Elements loop
      Stages (Stages'First).Feed (E);
   end loop;
end Mini;
```

(i) [2 marks] How many concurrent entities are implemented with this program? What are they?

(ii) [4 marks] What are the dependencies (shared data, synchronisation, etc.) between the concurrent entities and how do they interact? Could these concurrent entities be physically distributed? Give precise reasons.

(iii) [2 marks] How many stages do you need to process *n* elements with this algorithm (i.e. by feeding *n* elements into the first stage)?

(iv) [2 marks] When will the last stage receive its first data and when does it start to process (merge) data? Express this in global time, where the time unit is a single message.

(v) [4 marks] What is the time complexity for this algorithm (assuming that all stages are running on physically parallel hardware)?

(vi) [4 marks] What would be the total computational complexity (calculate this by adding up the computational complexities for all nodes)?

(vii) [2 marks] Will this program terminate? Give precise reasons.

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* □

continuation of answer to question □ *part* □

continuation of answer to question □ *part* □