The Australian National University
Second Semester Examination – November 2005

COMP2310
Concurrent and Distributed Systems

Study period:	15 minutes
Time allowed:	3 hours
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.

Name (family name first):

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. [10 marks] General Concurrency

(a) [4 marks] Give two advantages and two disadvantages associated with programming a concurrent application to run on a shared-memory multiprocessor environment (compared with a distributed memory message-passing environment).

(b) [2 marks] What basic facilities are expected to be provided by a concurrent programming language? What more advanced programming constructs might be provided? Give at least three.

(c) [4 marks] In the context of a concurrent computer program what is an invariant? Give a definition or a meaningful example. Explain why some invariants which are crucial for the correctness of a concurrent program can be hard to prove. Illustrate your answer with an example.

2. [25 marks] Synchronization

(a) [5 marks] Assume that there are three threads, `X`, `Y`, and `Z`, that repeatedly and continuously print "`X`", "`Y`", and "`Z`" respectively. Use semaphores to coordinate the printing such that the number of "`X`"s printed is always less than or equal to the sum of "`Y`"s and "`Z`"s printed.

(b) [5 marks] Using a shared variable of type integer and two binary semaphores construct a general counting semaphore.

(c) [6 marks] Consider the following C code (very carefully!):

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void){

  int array1[2];
  int *array2;
  int status;

  array2 = (int*)malloc(2*sizeof(int));

  array1[0]=10;
  array1[1]=11;
  array2[0]=20;
  array2[1]=21;

  printf("Initial addresses of arrays: %p %p\n", array1, array2);

  if (fork() == 0){

    printf("1st array1 is at %p: and holds: %d %d\n",array1,array1[0],array1[1]);
    printf("1st array2 is at %p: and holds: %d %d\n",array2,array2[0],array2[1]);
    array1[1]=31;
    array2[1]=41;

  } else {

    printf("2nd array1 is at %p: and holds: %d %d\n",array1,array1[0],array1[1]);
    printf("2nd array2 is at %p: and holds: %d %d\n",array2,array2[0],array2[1]);
    wait(&status);
    printf("3rd array1 is at %p: and holds: %d %d\n",array1,array1[0],array1[1]);
    printf("3rd array2 is at %p: and holds: %d %d\n",array2,array2[0],array2[1]);

  }
  return 0;
}
```

The code compiles and runs without problems. The initial part of the output on the terminal reads (`%p` is a pointer type printing out the address in hex-code):

```
Initial addresses of arrays: 0x3000 0x5000
```

(question continued on next page)

Detail the remaining output as produced by the rest of the code. If you are unable to specify the output exactly, state why. Explain what you assume in your answer about the underlying runtime system.

(d) [9 marks] The following questions concern monitors in concurrent systems.

(i) [5 marks] Specify a monitor in its most generic form and give one specific example of its usage.

(ii) [4 marks] Monitors are considered a mixture of high-level and low-level programming constructs. Explain which parts of a monitor these claims are referring to, and why.

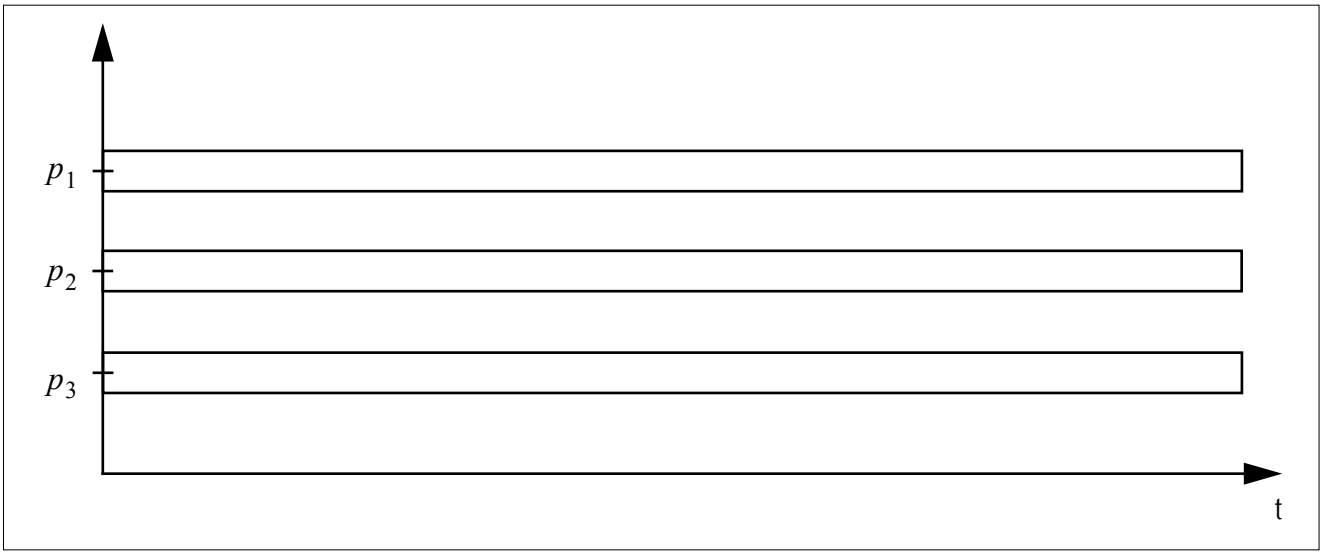
3. [18 marks] Message Passing

(a) [5 marks] What is the "requeue" facility (as in Ada95) and when is it essential? Give an example of a concurrent system which requires a requeue facility.

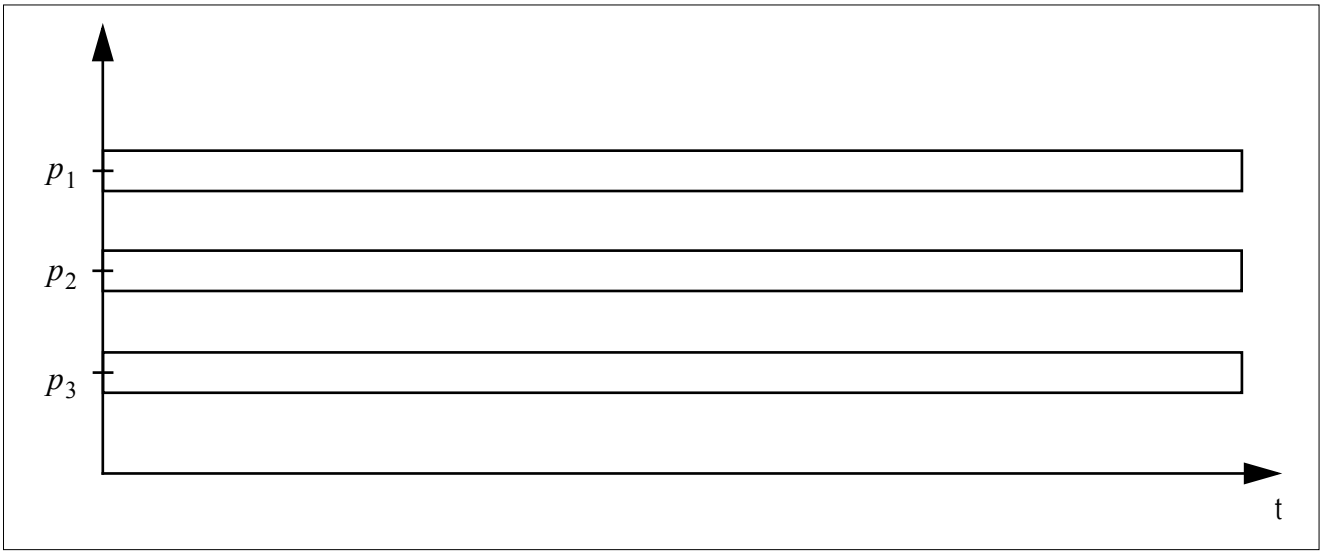
(b) [8 marks] Consider three processes `p1`, `p2`, `p3`, that will communicate with each other using send and receive message passing calls. The following series of events are supposed to take place concurrently:

```
Process p1          |  Process p2          |  Process p3
--------------------------------------------------------------
   A                |     D                |     G
receivefrom (p3)    |  sendto (p1)         |  receivefrom (p2)
   B                |     E                |     H
receivefrom (p2)    |  sendto (p3)         |  sendto (p1)
   C                |     F                |     I
```

(i) [2 marks] Detail a possible time-line of events assuming that the message passing facility is *asynchronous* (indicate the events `A-I`, and the messages by send-events, receive-events, and connecting arrows).



(ii) [2 marks] Detail a possible time-line of events assuming that the message passing facility is *synchronous* (indicate the events `A-I`, and the messages by send-events, receive-events, and connecting arrows).



(iii) [4 marks] Now consider that all three processes repeat the events given above in infinite loops and there is an underlying *asynchronous* communication system:

```
Process p1          |  Process p2          |  Process p3
--------------------------------------------------------------------
loop                |  loop                |  loop
   A                |     D                |     G
  receivefrom (p3)  |    sendto (p1)       |    receivefrom (p2)
   B                |     E                |     H
  receivefrom (p2)  |    sendto (p3)       |    sendto (p1)
   C                |     F                |     I
end loop            |  end loop            |  end loop
```

Detail what will happen to the progress of the two remaining processes if one of the processes dies unexpectedly (or is explicitly terminated). Differentiate the cases involving termination of processes `p1`, `p2`, or `p3` (i.e. only one process will terminate in every test-run). Explain what assumptions about the asynchronous communication system you made in your answer.

(c) [5 marks] Assume two completely different computer architectures (in terms of processors, languages, operating systems, etc.) which are connected via a common communication system. What aspects do you need to consider in the design of a working and reliable synchronous communication protocol (formats, synchronization, buffers, …) between these two computers? Explain what (if any) assumptions you make about the communication system itself.

4. [6 marks] Scheduling

(a) [6 marks] You are required to design a CPU-scheduler which minimizes the maximal turnaround time of a task set without deadlines.

(i) [4 marks] Describe your chosen implementation in the case of unknown computation times and give reasons for your decision(s).

(ii) [2 marks] Now assume that every task provides with every scheduling request the expected amount of CPU time which this task will need. Is it useful to change the scheduler and to incorporate this new information? (You still need to minimize the maximal turnaround time.) If so: in which ways, if not: why not?

5. [18 marks] Safety and Liveness

(a) [5 marks] Name and describe in reasonable detail five (!) ways to ensure that a concurrent system is safe (from deadlocks).

(b) [13 marks] The following Ada program (page 18) is syntactically correct and will compile without warnings. It produces output on the terminal with every successful `Write` or `Read` call on the protected object `Shared_Object`.

(i) [1 mark] How many tasks do you find in this code? (Do not forget to count the 'main' program.) Give their names.

(ii) [2 marks] Does the set of tasks constitute a deterministic system (in the sense that it produces exactly the same output with every test run)? Give reasons for your answer. Does your answer depend on assumptions about the underlying hardware or run-time environment? If so, what are those assumptions?

(iii) [10 marks] Will/might the set of tasks **terminate**, **deadlock**, **livelock**, or **run indefinitely**? Detail your answer with a calling sequence which leads to this situation. What output(s) do you expect to see on the terminal? (If you declared the system non-deterministic in the question above, you might need to come up with multiple possibilities and you also need to provide multiple calling sequences – one for each principal case.)

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

procedure Communicating_Processes is

   task Producer_Client;
   task Consumer_Client;

   task Producer is
      entry Available;
      entry Do_Something;
   end Producer;

   task Consumer is
      entry Available;
      entry Do_Something;
   end Consumer;

   protected Shared_Object is
      entry Write;
      entry Read;
   private
      Filled : Boolean := False;
   end Shared_Object;

   task body Producer_Client is
   begin
      Producer.Do_Something;
   end Producer_Client;

   task body Consumer_Client is
   begin
      Consumer.Do_Something;
   end Consumer_Client;

   task body Producer is
   begin
      loop
         select
            accept Available;
         or
            accept Do_Something do
               Shared_Object.Write;
               select
                 Consumer.Available;
                 Consumer.Do_Something;
               else
                  null;
               end select;
            end Do_Something;
         or
            terminate;
         end select;
      end loop;
   end Producer;

   task body Consumer is
   begin
      loop
         select
            accept Available;
         or
            accept Do_Something do
               Shared_Object.Read;
               select
                  Producer.Available;
                  Producer.Do_Something;
               else
                  null;
               end select;
            end Do_Something;
         or
            terminate;
         end select;
      end loop;
   end Consumer;

   protected body Shared_Object is

      entry Write when not Filled is
      begin
         Filled := True; Put ("Write");
      end Write;

      entry Read when Filled is
      begin
         Filled := False; Put ("Read");
      end Read;

   end Shared_Object;

begin
   null;
end Communicating_Processes;
```

6. [23 marks] Distributed Systems

(a) [4 marks] This question addresses issues associated with virtual (logical) times in distributed systems. If you find two logical times *C(a)* and *C(b)* attached to events *a* and *b* in different processes, then what can you conclude if:

(i) *C(a) = C(b)*

(ii) *C(a) > C(b)*

Alternatively, if you know something about the relation between the events *a* and *b* in a distributed system, what can you conclude about their logical times *C(a)* and *C(b)* if:

(iii) *a* happened concurrently with *b*

(iv) *a* always triggers *b*

(b) [4 marks] Fetching a global snapshot in a distributed system can be hard to achieve.

(i) [2 marks] Can you suggest a distributed systems in which it is (relatively) easy to fetch a global snapshot (i.e. you can do so within a constant and short timespan)? Describe such a system.

(ii) [2 marks] If you do not have assurances like the ones you detailed in the first part of this question, and you need to construct a distributed algorithm to implement a global snapshot, what is the issue you need to take special care of? (Which potential danger can render your global snapshot useless?) Give a precise answer.

(c) [4 marks] Suggest a practical distributed system where you would implement a two-phase locking transaction scheduler, and one practical distributed system where you would implement an optimistic transaction scheduler. Give reasons for your decisions.

(d) [11 marks] The following program (page 24) implements a distributed, symmetrical server configuration. For the sake of simplicity it is assumed that all job identifiers are unique. A client can address any of the servers in the group and will receive any results directly from this server. The actual processing of a job is done by a free server in the group (not necessarily the server communicating with the client). Client-job processing is done in the local function `Processing` and the decision about the availability of a server is done locally in the function `Server_Available` (both not detailed in the given implementation). All servers run exactly the same program.

(i) [2 marks] Read the actual implementation below (which compiles warning-free and implements a ring of identical servers). What is the communication effort (number of entry-calls) per job? (Do not count requeues.)

(ii) [3 marks] Can this system deadlock? If you think that it can deadlock, describe the chain of dependencies and suggest a way to resolve the problem. (No limitations here; you can employ any construct/method you see fit.)

(iii) [6 marks] How will this server cluster behave in high-load situations? Detail your answer by addressing overall performance, reliability, scalability, and responsiveness. Assume message delays and processing delays. Distinguish between the situation where the clients call all of the servers equally, or when some servers are frequently preferred over others. If you detect shortcomings, suggest improvements.

```
procedure Processes_In_Ring is

   Ring_Size : constant Positive := 12;

   type Ring_Range  is mod Ring_Size;
   type Job_Type    is mod 42;
   type Ring_Packet is record
                          Job, Job_Result : Job_Type;
                          Done            : Boolean;
                       end record;

   task type Server is
      entry Set_Server_Id (Id      : in Ring_Range);
      entry Job           (New_Job : in Job_Type; Job_Result : out Job_Type);
      entry Forward_Job   (Packet  : in Ring_Packet);
   private
      entry Job_Internal  (Job     : in Job_Type; Job_Result : out Job_Type);
   end Server;

   Servers : array (Ring_Range) of Server;
```

(cont. on next page)

```
task body Server is
      type Job_SetT is array (Job_Type'Range) of Boolean;
      Job_Set               : Job_SetT    := (others => False);
      Circulation_Complete  : Boolean     := False;
      Server_Id             : Ring_Range;
      Current_Packet        : Ring_Packet;

      function Processing (Job : in Job_Type) return Job_Type is (...)
      function Server_Available return Boolean is (...)

   begin
      accept Set_Server_Id (Id : in Ring_Range) do
         Server_Id := Id;
      end Set_Server_Id;
      loop
         select
            accept Job (New_Job : in Job_Type; Job_Result : out Job_Type) do
               Current_Packet := (Job        => New_Job,
                                  Job_Result => Job_Type'First,
                                   Done       => False);
               Job_Set (New_Job) := True;
               requeue Job_Internal;
            end Job;
            Servers(Server_Id + 1).Forward_Job (Current_Packet);
         or
            when Circulation_Complete =>
               accept Job_Internal (Job: in Job_Type; Job_Result: out Job_Type) do
                  Job_Result    := Current_Packet.Job_Result;
                  Job_Set (Job) := False;
               end Job_Internal;
               Circulation_Complete := False;
         or
            when not Circulation_Complete =>
               accept Forward_Job (Packet : in Ring_Packet) do
                  Current_Packet := Packet;
               end Forward_Job;
               Circulation_Complete := Job_Set (Current_Packet.Job);
               if not Current_Packet.Done
                and (Circulation_Complete or Server_Available) then
                  Current_Packet.Job_Result := Processing (Current_Packet.Job);
                  Current_Packet.Done       := True;
               end if;
               if not Circulation_Complete then
                  Servers(Server_Id + 1).Forward_Job (Current_Packet);
               end if;
         or
            terminate;
         end select;
      end loop;
   end Server;

begin
   for Id in Ring_Range loop Servers(Id).Set_Server_Id (Id); end loop;
end Processes_In_Ring;
```

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 [ ]