1 Semaphores

Consider a shared data structure that can be both read from and written to. The readers and writers operate upon shared data that consist of two integer values, x and y. The data structure is in a consistent state if y’s value is double the value of x. Show how semaphores can be used to allow many concurrent readers or a single writer but not both. Make sure that readers that access the shared data structure see it in a consistent state. If necessary, modify your solution to make it starvation-free.

A semaphore implementation along with an unsynchronised version (i.e., with race conditions that cause inconsistent state) of the readers/writers is given at

http://www.iste.uni-stuttgart.de/ps/Lehre/SS2011/V_RTP/a5/semaphore.adb
http://www.iste.uni-stuttgart.de/ps/Lehre/SS2011/V_RTP/a5/rw.adb

You are asked to insert appropriate semaphore calls to eliminate the race conditions.

2 Parent/Child and Guardian/Dependant

For every task in the Ada program given at

http://www.iste.uni-stuttgart.de/ps/Lehre/SS2011/V_RTP/a5/guardians.adb

indicate its parent and guardian (or master in Ada terminology), and if appropriate its children and dependants. Also indicate the dependants of the Guardians and Hierarchy procedures.

3 Concurrency

The following program tries to dispatch tasks to available processors. However, sometimes the system gets stuck and only a reset seems to help. Inspect the program and explain what is wrong.

```ada
procedure Dispatcher is
  type Proc_Num is mod 10;

  task type Big_Task (Size : Positive);
  -- Size is the number of required
  -- processors for this task

  protected type Processor is
    entry Allocate;
    function Is_Allocated return Boolean;
    procedure Release;
  private
    Allocated : Boolean := False;
  end Processor;

  type Big_Task_Array is array (1..5) of Big_Task (3);
  type Processor_Array is array (Proc_Num) of Processor;

  Task_Set : Big_Task_Array;
  Processors : Processor_Array;

  function Next_One return Proc_Num is
    -- Return the next free processor
    I : Proc_Num := Processors'FIRST;
    begin
      loop
        if not Processors (I).Is_Allocated then
          return I;
        end if;
        I := I + 1;
      end loop;
    end Next_One;

  task body Big_Task is
    Procs : array (1..Size) of Proc_Num;
    begin
      loop
        -- Allocate all processors
        for I in Procs'RANGE loop
          Processors (Procs (I)).Allocate;
        end loop;

        -- ... utilise processors ...
        for I in Procs'RANGE loop
          Processors (Procs (I)).Release;
        end loop;
      end loop;
    end loop;
  end Big_Task;

  protected body Processor is
    entry Allocate when not Allocated is
      begin
        Allocated := True;
      end;

    function Is_Allocated return Boolean is
      begin
        return Allocated;
      end;

    procedure Release is
      begin
        Allocated := False;
      end;
  end Processor;

begin
  null;
end Dispatcher;
```

4 Concurrent Execution

The following two program fragments are given:

```ada
Start := Ada.Real_Time.Clock;
First_Action;
delay until Start + 10.0;
Second_Action;
```

```ada
Start := Ada.Real_Time.Clock;
First_Action;
delay (Start + 10.0) - Ada.Real_Time.Clock;
Second_Action;
```

`First_Action` and `Second_Action` are calls to procedures that execute sequentially. Both program fragments are executed in a concurrent environment. Do both fragments behave the same? If your answer is no, please describe an execution scenario that shows the different behaviour.