Consider a small but very popular museum of limited capacity. The fire marshal has ruled that no more than 100 persons can visit the museum at a given time. There are four entrances to the museum, protected by turnstiles. The goal is to count visitors in the museum and sell tickets only when the number of visitors is lower that the allowed capacity.

The software reading turnstile sensors is as follows:

```
package Turnstile is

-- This package provides an interface to the
-- Museum's turnstile sensors

   -- The Museum has four doors
   type Entrance_Type  is (North, South, East, West);
   -- A person can enter or leave through a turnstile
   type Direction_Type is (Enter, Leave);

   procedure Get (Entrance  : in  Entrance_Type;
                  Direction : out Direction_Type);
   -- The caller of this procedure is blocked until a
   -- person passes through the turnstile at the given
   -- entrance.  Its return value indicates whether the
   -- person has entered or left the Museum.

end Turnstile;
```

The program controlling sale of tickets is unreliable: it works “most of the time”, but sometime crashes on “Constraint Error”, sometimes it indicates that people are still in the museum at closing time, although the inspection by museum personnel indicates that the premises are empty, etc. In short, this program is unreliable. What is wrong with it?

```
with Turnstile;             use Turnstile;
with Ada.Exceptions;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
with Ada.Text_IO;           use Ada.Text_IO;
procedure Museum_V1 is

-- This program uses a global variable shared by five tasks
-- to monitor the number of visitors inside a museum
--
-- This version works "most of the time"
--
--     Sometimes at closing time, it indicates that
--     the museum is not empty when indeed it is empty
--
--     Sometimes an entrance monitor task terminates with
--     the exception Constraint_Error

   Maximum     : constant := 100; -- Museum capacity
   Population  : Natural  := 0; -- Current count of visitors

   -- A task type for monitoring turnstiles
   task type Entrance_Monitor
             (My_Entrance : Turnstile.Entrance_Type);

   task body Entrance_Monitor is
      Direction : Turnstile.Direction_Type;
   begin
      loop
         -- Wait until someone passes through my turnstile
         Turnstile.Get (My_Entrance, Direction);

         -- Update the number of people in the Museum
         case Direction is
            when Turnstile.Enter =>
               Population := Population + 1;
            when Turnstile.Leave =>
```

```
                Population := Population - 1;
            end case;
         end loop;
      exception
         when Except : others =>
            Put_Line
              (File => Standard_Error,
               Item => Entrance_Type'Image (My_Entrance) &
                  " turnstile task terminated with exception "
                  & Ada.Exceptions.Exception_Name (Except));
      end Entrance_Monitor;

      -- One task to monitor each of the four museum doors
      North_Door : Entrance_Monitor (Turnstile.North);
      South_Door : Entrance_Monitor (Turnstile.South);
      East_Door  : Entrance_Monitor (Turnstile.East);
      West_Door  : Entrance_Monitor (Turnstile.West);

      Museum_Full : Boolean := False;

   begin
      loop
         Put ("Current number of visitors is ");
         Put (Item => Population, Width => 1);
         New_Line;

         -- Check for change in ticket selling status
         if not Museum_Full and Population >= Maximum then
            Put_Line ("The Museum is full.  " &
                      "Suspend ticket sales.");
            Museum_Full := True;
         elsif Museum_Full and Population < Maximum then
            Put_Line ("The Museum is no longer full.  " &
                      "Resume ticket sales.");
            Museum_Full := False;
         end if;

         delay 1.0;  -- Update status every second
      end loop;
   end Museum_V1;
```

The reliable solution using Protected Objects:

```
with Turnstile;             use Turnstile;
with Ada.Exceptions;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
with Ada.Text_IO;           use Ada.Text_IO;
procedure Museum is

-- This program uses a protected object shared by five
-- tasks to monitor the number of visitors inside a museum

   Maximum : constant := 100;  -- The Museum capacity

   protected Population is -- The current number of visitors
      procedure Increment;
      procedure Decrement;
      function  Current return Natural;
   private
      Count : Natural  := 0;
   end Population;

   protected body Population is
      procedure Increment is
      begin
         Count := Count + 1;
      end Increment;

      procedure Decrement is
      begin
         Count := Count - 1;
      end Decrement;

      function Current return Natural is
      begin
         return Count;
      end Current;
   end Population;

   -- A task type for monitoring turnstiles
   task type Entrance_Monitor
             (My_Entrance : Turnstile.Entrance_Type);
```

```
   task body Entrance_Monitor is
      Direction : Turnstile.Direction_Type;
   begin
      loop
         -- Wait until someone passes through my turnstile
         Turnstile.Get (My_Entrance, Direction);

         -- Update the number of people in the Museum
         case Direction is
            when Turnstile.Enter =>
               Population.Increment;
            when Turnstile.Leave =>
               Population.Decrement;
         end case;
      end loop;
   exception
      when Except : others =>
         Put_Line
           (File => Standard_Error,
            Item => Entrance_Type'Image (My_Entrance) &
               " turnstile task terminated with exception "
               & Ada.Exceptions.Exception_Name (Except));
   end Entrance_Monitor;

   -- One task to monitor each of the four museum doors
   North_Door : Entrance_Monitor (Turnstile.North);
   South_Door : Entrance_Monitor (Turnstile.South);
   East_Door  : Entrance_Monitor (Turnstile.East);
   West_Door  : Entrance_Monitor (Turnstile.West);

   Museum_Full   : Boolean := False;
   Current_Count : Natural;

begin
   loop
      Current_Count := Population.Current;
      Put ("Current number of visitors is ");
      Put (Item => Current_Count, Width => 1);
      New_Line;

      -- Check for change in ticket selling status
      if not Museum_Full and Current_Count >= Maximum then
```

```
         Put_Line ("The Museum is full.  " &
                   "Suspend ticket sales.");
         Museum_Full := True;
      elsif Museum_Full and Current_Count < Maximum then
         Put_Line ("The Museum is no longer full.  " &
                   "Resume ticket sales.");
         Museum_Full := False;
      end if;

      delay 1.0;  -- Update status every second
   end loop;
end Museum;
```

Protected objects provide simple and elegant solution to the problem of protecting shared data or enforcing mutual exclusion when executing critical sections of code. Such sections are inside protected object(s).

Modus operandi:
Protected object is equipped with two locks:

- **Read-only lock**: allowing simultaneous reading accesses but barring write access
- **Read-Write lock**: enforcing mutual exclusion: only one modifier at a time, viz.:

```
protected Population is -- The current number of visitors
   procedure Increment;
   procedure Decrement;
   function  Current return Natural;
private
   Count : Natural  := 0;
end Population;

protected body Population is
   procedure Increment is
   begin
      Count := Count + 1;
   end Increment;

   procedure Decrement is
   begin
      Count := Count - 1;
   end Decrement;

   function Current return Natural is
   begin
      return Count;
   end Current;
end Population;
```

Functions act as readers, procedures act as writers.

NOTE: Lack of synchronization. A crowd of readers can starve writers for access to the protected object.

Consider the following generic package. We will use it in subsequent example.

```
generic
   type Element_Type is private;
package Bounded_Queue is

   type Queue_Type (Max_Size : Positive) is limited private;

   Overflow  : exception;
   Underflow : exception;

   procedure Clear (Queue : in out Queue_Type);

   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type);
   -- Overflow raised on attempt to Enqueue an element onto
   --          a full queue.  Queue is unchanged.

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type);
   -- Underflow raised on attempt to dequeue an element from
   --           an empty Queue.  Queue remains empty.

   function Full (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

private

   type Queue_Array is array (Positive range <>) of
Element_Type;
   type Queue_Type (Max_Size : Positive) is
      record
        Count : Natural  := 0;         -- # items in queue
        Front : Positive := 1;         -- First item index
        Rear  : Positive := Max_Size;  -- Last item index
        Items : Queue_Array (1 .. Max_Size);  -- Queue array
      end record;

end Bounded_Queue;
```

```
package body Bounded_Queue is

   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type) is
   begin
      if Queue.Count = Queue.Max_Size then raise Overflow;
      else
         Queue.Rear := Queue.Rear rem Queue.Max_Size + 1;
         Queue.Items (Queue.Rear) := Item;
         Queue.Count := Queue.Count + 1;
      end if;
   end Enqueue;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type) is
   begin
      if Queue.Count = 0 then raise Underflow;
      else
         Item := Queue.Items (Queue.Front);
         Queue.Front := Queue.Front rem Queue.Max_Size + 1;
         Queue.Count := Queue.Count - 1;
      end if;
   end Dequeue;

   function Full (Queue : in Queue_Type) return Boolean is
   begin
      return Queue.Count = Queue.Max_Size;
   end Full;

   function Empty (Queue : in Queue_Type) return Boolean is
   begin
      return Queue.Count = 0;
   end Empty;

   procedure Clear (Queue : in out Queue_Type) is
   begin
      Queue.Count := 0; Queue.Front := 1;
      Queue.Rear  := Queue.Max_Size;
   end Clear;

end Bounded_Queue;
```

Producer – Consumer Demo using Protected Objects with Entries:

```
with Ada.Integer_Text_IO;         use Ada.Integer_Text_IO;
with Ada.Text_IO;                 use Ada.Text_IO;
with Ada.Numerics.Float_Random;   use Ada.Numerics;
with Bounded_Queue;
procedure Producer_Consumer_Demo is

   Number_Of_Producers : constant := 5;
   Producer_Iterations : constant := 8;

   ------------------------------------------------------
   package Integer_Queue is new
           Bounded_Queue (Element_Type => Positive);

   ------------------------------------------------------
   protected type Bounded_Buffer (Max_Size : Positive) is
      procedure Clear;
      -- delete all of the items in the buffer
      entry Put (Item : in Positive);
      -- add a value to the buffer
      entry Take (Item : out Positive);
      -- remove a value from the buffer
   private
      Buffer : Integer_Queue.Queue_Type (Max_Size);
   end Bounded_Buffer;

   protected body Bounded_Buffer is
      procedure Clear is
      begin
         Integer_Queue.Clear (Buffer);
      end Clear;

      entry Put (Item : in Positive)
            when not Integer_Queue.Full (Buffer) is
      begin
         Integer_Queue.Enqueue (Queue => Buffer,
                                Item  => Item);
      end Put;

      entry Take (Item : out Positive)
            when not Integer_Queue.Empty (Buffer) is
```

```
   begin
      Integer_Queue.Dequeue (Queue => Buffer,
                             Item  => Item);
   end Take;
end Bounded_Buffer;

-- The buffer object.  Holds a maximum of 3 entries.
The_Buffer : Bounded_Buffer (Max_Size => 3);

-------------------------------------------------
protected ID_Generator is
   procedure Get (ID : out Positive);
   -- Returns a unique positive ID number
private
   Next_ID : Positive := 1;
end ID_Generator;

protected body ID_Generator is
   procedure Get (ID : out Positive) is
   begin
      ID := Next_ID;
      Next_ID := Next_ID + 1;
   end Get;
end ID_Generator;

-----------------------------------------------------

protected Random_Duration is
   procedure Get (Item : out Duration);
   -- Returns a random duration value
   -- between 0.0 and 1.0 seconds
private
   First_Call   : Boolean := True;
   My_Generator : Ada.Numerics.Float_Random.Generator;
end Random_Duration;

protected body Random_Duration is
   procedure Get (Item : out Duration) is
   begin
      if First_Call then
         -- On the first call, reset the generator
         Ada.Numerics.Float_Random.Reset (My_Generator);
```

```
            First_Call := False;
         end if;
         -- Get a random value and convert it to a duration
         Item := Duration (Float_Random.Random
(My_Generator));
      end Get;
   end Random_Duration;

   ------------------------------------------------------

   task type Producer;

   task body Producer is
      My_ID    : Positive;
      My_Delay : Duration;
   begin
      -- Get a unique ID for this task
      ID_Generator.Get (ID => My_ID);
      -- Put my ID into the bounded buffer 8 times
      for Count in 1 .. Producer_Iterations loop
         -- Put my ID into the buffer
         The_Buffer.Put (My_ID);
         -- Simulate the time to do the work
         -- to actually produce something
         Random_Duration.Get (My_Delay);
         delay My_Delay;
      end loop;
   end Producer;

   -- A number of producer tasks
   type Producer_Array is array (1 .. Number_Of_Producers)
                          of Producer;
   Producers : Producer_Array;

   ------------------------------------------------------

   Value          : Positive;
   Consumer_Delay : Duration;
begin
   -- Consume everything in the bounded buffer
   for Count in 1 .. Number_Of_Producers *
Producer_Iterations
```

```
   loop
      The_Buffer.Take (Value);
      Put (Item => Value, Width => 2);
      New_Line;
      -- Simulate the time to do the work
      -- to actually consume something
      Random_Duration.Get (Consumer_Delay);
      delay Consumer_Delay / Number_Of_Producers;
   end loop;
end Producer_Consumer_Demo;
```

Producer – Consumer Relationship using Atomic Operations:

```
procedure Atomic_Demo is

   Value : Natural := 0;
   pragma Atomic (Value);

   task type Writer;
   task body Writer is
      My_Value : Natural := 0;
   begin
      loop
         -- Create My_Value
         -- My_Value := ...
         --Write My_Value to the shared global
         Value := My_Value;
         delay 0.1;
      end loop;
   end Writer;

   task type Reader;
   task body Reader is
      My_Value : Natural;
   begin
      loop
         -- Read the shared global
         My_Value := Value;
         -- Do something with My_Value
         -- ...
         delay 0.2;
      end loop;
   end Reader;

   type Writer_Array is array (1 .. 2) of Writer;
   type Reader_Array is array (1 .. 8) of Reader;

   The_Writers : Writer_Array;
   The_Readers : Reader_Array;

begin
   Value := 25;
end Atomic_Demo;
```

Protected Entries: Similar to protected procedures, but also equipped with **barriers**, for synchronization. Rules:

- A barrier evaluating to True is said to be open;
- A barrier evaluating to False is said to be closed;
- A task calling a protected entry with a closed barrier is blocked until the barrier becomes open.
- The barrier is evaluated when an entry is first called.
- Then it is re-evaluated when something happens which might change its value, i.e. after the completion of an entry call or a protected procedure body.
- Barrier should not test values outside of its protected object.

Example: **Producer – Consumer Demo**:

```
with Ada.Integer_Text_IO;         use Ada.Integer_Text_IO;
with Ada.Text_IO;                 use Ada.Text_IO;
with Ada.Numerics.Float_Random;   use Ada.Numerics;
with Bounded_Queue;
procedure Producer_Consumer_Demo is

   Number_Of_Producers : constant := 5;
   Producer_Iterations : constant := 8;

   ------------------------------------------------------
   package Integer_Queue is new
           Bounded_Queue (Element_Type => Positive);

   ------------------------------------------------------
   protected type Bounded_Buffer (Max_Size : Positive) is
      procedure Clear;
      -- delete all of the items in the buffer
      entry Put (Item : in Positive);
      -- add a value to the buffer
      entry Take (Item : out Positive);
      -- remove a value from the buffer
   private
      Buffer : Integer_Queue.Queue_Type (Max_Size);
   end Bounded_Buffer;
```

```
protected body Bounded_Buffer is
   procedure Clear is
   begin
      Integer_Queue.Clear (Buffer);
   end Clear;

   entry Put (Item : in Positive)
         when not Integer_Queue.Full (Buffer) is
   begin
      Integer_Queue.Enqueue (Queue => Buffer,
                             Item  => Item);
   end Put;

   entry Take (Item : out Positive)
         when not Integer_Queue.Empty (Buffer) is
   begin
      Integer_Queue.Dequeue (Queue => Buffer,
                             Item  => Item);
   end Take;
end Bounded_Buffer;

-- The buffer object.  Holds a maximum of 3 entries.
The_Buffer : Bounded_Buffer (Max_Size => 3);

-------------------------------------------------
protected ID_Generator is
   procedure Get (ID : out Positive);
   -- Returns a unique positive ID number
private
   Next_ID : Positive := 1;
end ID_Generator;

protected body ID_Generator is
   procedure Get (ID : out Positive) is
   begin
      ID := Next_ID;
      Next_ID := Next_ID + 1;
   end Get;
end ID_Generator;

-------------------------------------------------------
```

```
protected Random_Duration is
      procedure Get (Item : out Duration);
      -- Returns a random duration value
      -- between 0.0 and 1.0 seconds
   private
      First_Call   : Boolean := True;
      My_Generator : Ada.Numerics.Float_Random.Generator;
   end Random_Duration;

   protected body Random_Duration is
      procedure Get (Item : out Duration) is
      begin
         if First_Call then
            -- On the first call, reset the generator
            Ada.Numerics.Float_Random.Reset (My_Generator);
            First_Call := False;
         end if;
         -- Get a random value and convert it to a duration
         Item := Duration (Float_Random.Random
(My_Generator));
      end Get;
   end Random_Duration;

   ------------------------------------------------------
   task type Producer;

   task body Producer is
      My_ID    : Positive;
      My_Delay : Duration;
   begin
      -- Get a unique ID for this task
      ID_Generator.Get (ID => My_ID);
      -- Put my ID into the bounded buffer 8 times
      for Count in 1 .. Producer_Iterations loop
         -- Put my ID into the buffer
         The_Buffer.Put (My_ID);
         -- Simulate the time to do the work
         -- to actually produce something
         Random_Duration.Get (My_Delay);
         delay My_Delay;
      end loop;
   end Producer;
```

```
   -- A number of producer tasks
   type Producer_Array is array (1 .. Number_Of_Producers)
                          of Producer;
   Producers : Producer_Array;

   ------------------------------------------------------
--
   Value          : Positive;
   Consumer_Delay : Duration;
begin
   -- Consume everything in the bounded buffer
   for Count in 1 .. Number_Of_Producers *
Producer_Iterations
   loop
      The_Buffer.Take (Value);
      Put (Item => Value, Width => 2);
      New_Line;
      -- Simulate the time to do the work
      -- to actually consume something
      Random_Duration.Get (Consumer_Delay);
      delay Consumer_Delay / Number_Of_Producers;
   end loop;
end Producer_Consumer_Demo;
```

Restrictions:

A *critical section* is a group of instructions that MUST NOT be executed concurrently by multiple tasks. Bodies of protected operations are critical sections.

Thou shall not execute a potentially blocking operation within a protected operation. Those include:

- Delay statements;
- Calls to protected object entries;
- Creation or activation of a task;
- Calls to subprograms containing potentially blocking operations.

Violation of this may result in **bounded error**, which may not be detected at compile time.

Useful patterns: Barriers
(Not to be confused with barriers inherent to protected objects)

```
package Barriers is

   protected type Barrier (Group_Size : Positive) is
      -- wait until Group_Size tasks are waiting at this
entry
      entry Wait;
   private
      Gate_Open : Boolean := False;
   end Barrier;

end Barriers;
```

implementation:

```
package body Barriers is

   protected body Barrier is
      entry Wait
            when Wait'Count = Group_Size or Gate_Open is
      begin
         if Wait'Count > 0 then
            -- The first task released opens the gate
            -- for the rest. Tasks released before the
            -- last task keep the gate open.
            Gate_Open := True;
         else
            -- The last task released closes the gate
            Gate_Open := False;
         end if;
      end Wait;
   end Barrier;

end Barriers;
```

Example of use:

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
with Barriers;              use Barriers;
procedure Barrier_Demo is
-- This program demonstrates the barrier. It creates horse
-- tasks dynamically without deallocating completed task
-- memory.

   Starting_Gate : Barrier (Group_Size => 5);

   task type Horse (My_ID : Positive);
   task body Horse is
   begin
      Starting_Gate.Wait;
      Put (My_ID);
      New_Line;
   end Horse;

   type Horse_Ptr is access Horse;
   Racer : Horse_Ptr; ID    : Positive   := 1;

begin
   -- Each iteration of this loop brings two horese to the
starting gate
   -- As soon as five horses have arrived, the gate is
opened.  The sixth
   -- horse to arrive must wait for the next race, even if
the first five
   -- are still leaving the startng gate.
   loop
      Put_Line ("Press enter to bring two horses to the
starting gate");
      Skip_Line;
      Racer := new Horse (ID);
      ID := ID + 1;
      Racer := new Horse (ID);
      ID := ID + 1;
   end loop;

end Barrier_Demo;
```

Useful patterns: Broadcasts:

We want to send a message to a number of tasks waiting for it. As in radio, tasks waiting for a message must be 'tuned in' and waiting to receive it. Those which tune in too late miss the message (which may be re-broadcast later):

```
generic
   type Message_Type is private;
package Broadcasts is

   protected type Broadcast is
      procedure Send (Message : in Message_Type);
      -- Send a message to all waiting tasks
      entry Tune_In (Message : out Message_Type);
      -- Wait for a message
   private
      The_Message  : Message_Type;
      Have_Message : Boolean := False;
   end Broadcast;

end Broadcasts;
```

Implemented as:

```
package body Broadcasts is

   protected body Broadcast is

      procedure Send (Message : in Message_Type) is
      begin
         -- Do something only if tasks are waiting
         if Tune_In'Count > 0 then
            The_Message  := Message;
            Have_Message := True;
         end if;
      end Send;

      entry Tune_In (Message : out Message_Type)
            when Have_Message is
      begin
         Message := The_Message;
         if Tune_In'Count = 0 then
            Have_Message := False;
         end if;
      end Tune_In;

   end Broadcast;

end Broadcasts;
```