Australian National University

PROTECTION

Week 4 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer based on material by Alistair Rendell

Pre-Laboratory Checklist

- ❑ **You mastered tasks in basic form.**
- ❑ **You know what happens if concurrent tasks are accessing shared data in an uncoordinated way.**

Hurdle Lab – you need to get this right!

Objectives

In this lab you will finally see the smooth and elegant solutions to the shared-data-among-concurrent-entities problem. Rejoice and enjoy as you will leave shared variables based synchronization behind you. The lab is only loosely based on the *Massive Attack* album *Protection* or its remix version *No Protection*. There will be a lot of talk about protected and unprotected forms of accessing your data though. By the end of this lab, your solutions to shared memory based communication methods will have improved by two decades of research.

Your tutor needs to believe that you understood the material and solved exercise 1-3 on your own, i.e. you need to attend your lab. Solutions need to be uploaded by the end of week four.

Interlude: Protected Objects

In a nutshell, **protected objects** encapsulate data and provide concurrency-safe access to it. In terms of syntax, this will look like this (from the Ada Reference Manual, section 9.4):

```
protected_type_declaration ::=
   protected type defining_identifier
        [known_discriminant_part] [aspect_specification] is
      [new interface_list with] protected_definition;

single_protected_declaration ::=
   protected defining_identifier [aspect_specification] is
      [new interface_list with] protected_definition;

protected_definition ::=
      { protected_operation_declaration }
 [ private { protected_element_declaration } ]
   end [protected_identifier]
```

```
protected_operation_declaration ::=
   subprogram_declaration | entry_declaration | aspect_clause

protected_element_declaration ::=
   protected_operation_declaration | component_declaration

protected_body ::=
   protected body defining_identifier [aspect_specification] is
      { protected_operation_item }
   end [protected_identifier];

protected_operation_item ::=
   subprogram_declaration | subprogram_body | entry_body | aspect_clause
```

A few things to take note of:

- Something you will likely not use for some time are aspects on protected objects. Similar to tasks you can add further specifications to nail down what behaviour you need, e.g.:

  ```
  protected Object
     with Priority => Priority'Last is
  ```

  which would control the so-called ceiling priority of this protected object – which will become important in real-time systems.
- Protected objects can also be used to implement an interface in the common, object-oriented style. Those implementations will become final and no further derivations are possible. This is due to the **inheritance anomaly**[1] which prevents unconstrained, object-oriented inheritance to be applied in concurrent systems. This could look for instance like this:

  ```
  protected type Protected_Queue is new Queue_Interface with
     overriding entry Enqueue (Item :     Element);
     overriding entry Dequeue (Item : out Element);
        …
  ```

  Next week we will see that task types can implement the same interface:

  ```
  task type Protected_Queue_Task is new Queue_Interface with
     overriding entry Enqueue (Item :     Element);
     overriding entry Dequeue (Item : out Element);
  ```

The rest is much easier to be seen in a few examples. Compare the "unprotected" package on the left with the protected object on the right:

```
generic
   type Element is (<>); -- any discrete type
   Init : Element;       -- initial value
package Unprotected_Element_Generic is
   package Unprotected_Element is
      function  Get return Element;
      procedure Set (E : Element);
      procedure Inc;
      procedure Dec;
   private
      Store : Element := Init;
   end Unprotected_Element;
end Unprotected_Element_Generic;
```

```
generic
   type Element is (<>); -- any discrete type
   Init : Element;       -- initial value
package Protected_Element_Generic is
   protected Protected_Element is
      function  Get return Element;
      procedure Set (E : Element);
      entry Inc;
      entry Dec;
   private
      Store : Element := Init;
   end Protected_Element;
end Protected_Element_Generic;
```

1 S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In Research Directions in Concurrent Object-Oriented Programming, pages 107–150. 1993.

Rather small syntactical differences:

- `package` is replaced by `protected` for the module which encapsulate the data (`Store`) which potentially needs protection.
- Some of the `procedures` are replaced by `entries`. At this point it not quite clear why we would change some procedures but not others, but it will come in a moment.

Let's look at the implementations to gain the full picture:

```
package body Unprotected_Element_Generic is
   package body Unprotected_Element is
      function Get return Element is (Store);
      procedure Set (E : Element) is
      begin
         Store := E;
      end Set;
      procedure Inc is
      begin
         Store := Element'Succ (Store);
      end Inc;
      procedure Dec is
      begin
         Store := Element'Pred (Store);
      end Dec;
   end Unprotected_Element;
end Unprotected_Element_Generic;
```

```
package body Protected_Element_Generic is
   protected body Protected_Element is
      function Get return Element is (Store);
      procedure Set (E : Element) is
      begin
         Store := E;
      end Set;
      entry Inc when Store < Element'Last is
      begin
         Store := Element'Succ (Store);
      end Inc;
      entry Dec when Store > Element'First is
      begin
         Store := Element'Pred (Store);
      end Dec;
   end Protected_Element;
end Protected_Element_Generic;
```

Now we can see what's behind the entry idea: The two entries on the right (as opposed to the procedures on the left) have conditions attached to them (called guards). The package on the left is wide open to access by any task at any time – no task will ever be delayed or blocked while accessing any of the operations. At least that means it's fast … yet …

On the right side, the keyword `protected` as well as the specific structure around the `entry` definitions provide a set of rules which restrict access to those operations under certain conditions:

- `procedures` enforce **mutual exclusion** with all other operations in the same protected object. This means that maximally one task can enter a procedure inside a protected object (also called a `protected procedure`) at any one time and no other task can be inside the same protected object while this one task is operating inside the protected object.
- `entries` enforce the same constraints as procedures, yet are also associated with a guard (boolean expression) which needs to be fulfilled before a task is allowed to enter (otherwise it is blocked and neatly queued up until the guard opens up again). Those guards are being re-evaluated when a task leaves a procedure or entry within the same protected object – i.e. if a task leaves e.g. the `Dec` entry, then both guards will be re-evaluated and a potentially waiting task on the `Inc` entry can now gain permission to enter next.
- `functions` are side-effect free with respect to the protected data, i.e. the compiler will treat any write access to the protected data from within a protected function as an error. Given that, it is perfectly fine to grant many tasks concurrent access to protected functions, as long as this happens in mutual exclusion with procedures and entries.

In short: there can only ever be maximally one task inside a protected object if this task entered via a `protected procedure` or `entry`. Alternatively, multiple tasks can use one or multiple `protected functions` concurrently.

Exercise 1: Check it out

Let's test the above by hammering both of the structures with multiple tasks:

```
with Ada.Text_IO;                    use Ada.Text_IO;
with Protected_Element_Generic;
with Unprotected_Element_Generic;

procedure Count_Up is

begin
   for i in 1 .. 20 loop
      declare
         package Protected_Natural is new Protected_Element_Generic
           (Element => Natural, Init => 0);
         package Unprotected_Natural is new Unprotected_Element_Generic
           (Element => Natural, Init => 0);

         use Protected_Natural;
         use Unprotected_Natural;

         task type Count_up_by (Difference : Natural) is
            entry Done;
         end Count_up_by;

         task body Count_up_by is

         begin
            for i in 1 .. Difference loop
               Unprotected_Element.Inc;
            end loop;

            for i in 1 .. Difference loop
               Protected_Element.Inc;
            end loop;

            accept Done;
         end Count_up_by;

         Counters : array (1 .. 10) of Count_up_by (1_000);

      begin
         for t of Counters loop
            t.Done;
         end loop;

         Put_Line ("Protected value:" & Natural'Image (Protected_Element.Get) &
              " - Unprotected value:" & Natural'Image (Unprotected_Element.Get));
      end;
   end loop;
end Count_Up;
```

Download and run this program to see what happens. So far so obvious, right? Now for the more interesting part of your tests:

- What will happen if you exchange the two inner for-loops (if anything)?
- What will happen if you increment both values in the same for-loop (if anything)?
- What will happen if you count only to 10 inside each task (if anything)?

The results may surprise you – yet what would you expect to be the one thing which will never change with any of the above alternations? If you find a configuration for which the unprotected version appears to work: what can you do with that observation?

Submit your findings as a few lines of plain text explanations (keep it brief) to the *Submission-App* under "Lab 4 Checked".

Exercise 2: Canons on sparrows

This is like asking you to make a Ferrari look like an original Fiat Cinquecento, but sometimes I ask you to do simple things as well: Program a semaphore by means of a protected object. While semaphores are very simple structures, they need to be implemented spot-on or they are useless.

Here is a test framework for you to see your semaphores in classical action:

```
with Ada.Text_IO;  use Ada.Text_IO;
with Id_Dispenser;
with Semaphores;   use Semaphores;

procedure Philos is

   No_of_Philos : constant Positive := 5;
   Meditation   : constant Duration := 0.0;

   type Table_Ix is mod No_of_Philos;

   Forks : array (Table_Ix) of Binary_Semaphore (Initially_Available => True);

   package Index_Dispenser is new Id_Dispenser (Element => Table_Ix);
   use Index_Dispenser;

   task type Philo;
   task body Philo is

      Philo_Nr : Table_Ix;

   begin
      Dispenser.Draw_Id (Id => Philo_Nr);
      Put_Line ("Philosopher" & Table_Ix'Image (Philo_Nr) & " looks for forks.");
      Forks (Philo_Nr).Wait; delay Meditation; Forks (Philo_Nr + 1).Wait;
      Put_Line ("Philosopher" & Table_Ix'Image (Philo_Nr) & " eats.");
      Forks (Philo_Nr).Signal; Forks (Philo_Nr + 1).Signal;
      Put_Line ("Philosopher" & Table_Ix'Image (Philo_Nr) & " dropped forks.");
   end Philo;

   Table : array (Table_Ix) of Philo; pragma Unreferenced (Table);

begin
   null;
end Philos;
```

Philosophers sitting around a table and using two forks (one on the left and one the right of each philosopher). They can only eat if they acquired two forks and will put both down after being well fed.

This is unfortunately again incomplete and requires two modules from your side: The `Semaphores` package which needs to provide at least a binary semaphore (following the suggested syntax) as well as a package called: the `Id_Dispenser`. It is used in a way such that each task (philosopher) can draw its own unique id starting with the first value of the type provided in the instantiation of this generic package. Start by implementing this additional package. Reconstruct the details of the package from the way it is used here. Hint: you need to be able to guarantee that each task actually receives a unique id – irrespective of when tasks are requesting theirs.

Now run the above and play with different Meditation delays for the philosophers between picking up the left and the right fork. What happens? Are your semaphores working or are they broken? Can you make sure everybody gets to eat and the number of forks stays constant?

Submit your `Semaphore.zip` archive to the *SubmissionApp* under "Lab 4 Cinquecento" for detailed code review by us.

Exercise 3: Tasks in lockstep

The protected object mechanisms can not only be used to share data, but also to synchronize actions. Here comes a pattern which (attempts) to synchronize all tasks and release all of them once all tasks are ready to proceed. The `Count` attribute on entries comes in handy for this purpose as it indicates how many tasks are currently waiting on this particular entry. Here we open the entry `Synchronize` based on the number of waiting tasks:

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

procedure Synchronized_Action is

   No_Of_Tasks : constant Positive := 5;

   protected type Blockers (Group_Size : Positive) is

      entry Synchronize;

   end Blockers;

   protected body Blockers is

      entry Synchronize when Synchronize'Count = Group_Size is

      begin
         null;
      end Synchronize;

   end Blockers;

   Blocker : Blockers (No_Of_Tasks);

   task type In_Synchronized_Stages;

   task body In_Synchronized_Stages is

   begin
      Put_Line ("Task " & Image (Current_Task) & " starting up");
      delay 1.0;

      Blocker.Synchronize;

      Put_Line ("Task " & Image (Current_Task) & " in stage 1");
      delay 1.0;

      Blocker.Synchronize;

      Put_Line ("Task " & Image (Current_Task) & " in stage 2");
   end In_Synchronized_Stages;

   Staged_Tasks : array (1 .. No_Of_Tasks) of In_Synchronized_Stages;

begin
   null;
end Synchronized_Action;
```

Now I have to admit that this program looks better than it runs: It "hangs" - or at least it seems to stall for all but the most patient users. The idea of the program is that all tasks re-synchronize between stages, such that no task should every work in a later stage while some other task is still busy in an earlier stage.

Repair this program such that it indeed shows the intended behaviour and runs to completion. Hint: What is the temporal sequence in which you expect to see outputs on the screen? What is the rhythm in which outputs actually turn up? How can this possibly happen? … now you show see the bug (right?) and can repair it.

Submit your `Synchronized_Action.zip` archive to the *SubmissionApp* under "Lab 4 Lockstep" for code review by us.

This third exercise concludes your hurdle assessment.

Exercise 4: Multicast server

Let me throw in one more concept for you and you can program already rather interesting devices. If tasks are waiting for something to happen it can well be different things for different tasks. You can of course program as many guarded entries as there are different guards. This can be rather labour intensive as often those entries are almost identical besides a tiny variation in the guard. To address this issue, you can formulate what is called an entry family: It looks like a single entry, but it actually produces many entries, each one with their own guard and waiting queue.

Let's learn about this on a concrete example: Assume that tasks want to wait until there is a message for them. Obviously we could program a universal guard which opens if any messages arrive and tasks can then check wether they actually got mail – *or*: we can write an entry family such that each task can wait for a message which is actually for it. Have a look at this:

```
generic
   type Message is private;
   Postboxes : Positive := 3;
package Message_Server is

   subtype Box      is Positive range 1 .. Postboxes;
   type Boxes       is array (Positive range <>) of Box;
   type Box_Content is record
      Available : Boolean := False;
      Data      : Message;
   end record;
   type Box_Contents is array (Box) of Box_Content;

   protected Post_Office is

      procedure Send      (To : Box;   Data :     Message);
      procedure Multicast (To : Boxes; Data :     Message);
      procedure Broadcast             (Data :     Message);
      entry     Receive        (Box)  (Data : out Message);

   private
      Store           : Box_Contents;
      Check_Multicast : Boolean := True;
   end Post_Office;
end Message_Server;
```

All seems familiar with the only unusual spot being the additional "`(Box)`" thingy in front the `entry` `Receives`'s parameter list. This expresses that there are in fact not one `Receive` entry, but as many as the discrete type in parenthesis has elements (in this case: three). This enables users of this `Post_Office` to queue up for the right `Box` instead of being woken up every time a new message arrives for somebody. Nifty, isn't it?

This is how the implementation side of such a construct looks like:

```
package body Message_Server is
   protected body Post_Office is
      procedure Send (To : Box; Data : Message) is
      begin
         Store (To) := (Available => True, Data => Data);
      end Send;
      procedure Multicast (To : Boxes; Data : Message) is
      begin
         for B in To'Range loop
            Send (To (B), Data);
         end loop;
      end Multicast;
      procedure Broadcast (Data : Message) is
      begin
         for B in Box loop
            Send (B, Data);
         end loop;
      end Broadcast;
      entry Receive (for B in Box) (Data :out Message) when Store (B).Available is
      begin
         Store (B).Available := Receive (B)'Count > 0;
         Data                := Store (B).Data;
         Check_Multicast     := True;
      end Receive;
   end Post_Office;
end Message_Server;
```

There is an important detail to be observed here: Guards can never use entry parameters, but they can use the value of the family enumerator (`B` in this case), as this is actually a static value (think about it for a moment to confirm this to be true).

Now we use this new package for instance like this:

```
with Ada.Text_IO;    use Ada.Text_IO;
with Id_Dispenser;
with Message_Server;
procedure Multi_Cast is
   package Positive_Dispenser is new Id_Dispenser (Element => Positive);
   Number_Of_Tasks : constant Positive := 3;
   type Colour is (Red, Green, Blue, Yellow, Magenta, Cyan);
   package Colour_Server is new Message_Server (Message   => Colour,
                                                Postboxes => Number_Of_Tasks);
   use Colour_Server;
   task type Client;
   Clients : array (Box) of Client;
   task body Client is
      Data : Colour;
      Id   : Box;
   begin
      Positive_Dispenser.Dispenser.Draw_Id (Id);
      for i in 1 .. 3 loop -- collecting the three messages per task
         Post_Office.Receive (Id) (Data);
         Put_Line ("Task" & Box'Image (Id) & " received: " & Colour'Image (Data));
      end loop;
   end Client;
begin
   -- Broadcast data (resulting in 1 message per task)
   Post_Office.Broadcast (Cyan);
   delay 0.1;
   -- Send individual data (1 message per task)
   declare
      Paint : Colour := Colour'First;
   begin
      for i in Clients'Range loop
         Post_Office.Send (i, Paint);
         Paint := Colour'Succ (Paint);
      end loop;
   end;
   delay 0.1;
   -- Multicast data to some and individual to the rest (1 message per task)
   declare
      Recipients : constant Boxes := (1, 3);
   begin
      Post_Office.Multicast (Recipients, Magenta);
      Post_Office.Send      (2,          Yellow);
   end;
end Multi_Cast;
```

Now run the `Multi_Cast` program. It will hopefully behave as you expect it it to, but let me add some challenge: Comment-out the delay statements and try again. What happens? Why does that happen?

What you will find is called a **race condition**, i.e. the behaviour of the program becomes non-deterministic and depends on which task gets to a specific spot first. This is not always bad, but let's fix it here anyway. We need to avoid that some parts of the program are getting ahead, before other tasks have caught up. To this end you need to make the sending operations con-

ditional such that previous messages are not overwritten before they have been read. I leave this to you how you manage this in detail (there are many options), but let me give you some inspirations. The guard for the broadcast entry could for instance look like this:

```
entry Broadcast (Data : Message)
   when (for all B in Box => not Store (B).Available) is
```

The case is more complicated for the multicast, so let me suggest an option for this one too:

```
entry Multicast (To : Boxes; Data : Message) when Check_Multicast is
   begin
      if (for some i in To'Range => Store (To (i)).Available) then
         Check_Multicast := False;
         requeue Multicast;
      else
         …
```

This is using a technique which you did not see before: **`requeue`**. Here it simply sends a task back onto the same entry for the case that not all Mailboxes which are addressed are already empty. The boolean qualifier `(for some i in To'Range => Store (To (i)).Available)` expresses in standard math notation: $\exists i : \mathrm{Store}\,(\mathrm{To}\,(i).\,\mathrm{Available})$ or in English: some message still needs to be read first. You will have noticed that in such a schema, `Check_Multicast` has to be set to `True` someplace - where?. You should now be able to figure out the rest and to find a way to also synchronize the `Send` method properly. If you succeed, then your program should run through without the **`delay`** statements.

Submit your working `Multi_Cast.zip` archive to the *SubmissionApp* under "Lab 4 Multicast" for a detailed code review by us.

MAKE SURE YOU LOGOUT
TO TERMINATE YOUR SESSION!

Outlook

Next week you will make your tasks listen to multiple channels at the same time and handle many more interesting cases of interacting tasks.