8: Tasks – Concurrency in Ada

We focus on very simple example setting:

- a computationally intensive task is split into t different subtasks
- the subtasks are divided among $p \leq t$ different processes (Ada-**task**s), each performing about t/p subtasks
- ideally p-times faster than a sequential process (if p or more processors or cores are available!)

Our concrete example:

- subtask: compute the n-th value of Hofstadter's Q sequence

$$\begin{aligned} n \leq 2: \; Q(n) &= 1 \\ n > 2: \; Q(n) &= Q(n - Q(n-1)) + Q(n - Q(n-2)) \end{aligned}$$

- complete task: compute the values $Q(1), \ldots, Q(100)$

The Work to be Done!

```
package Hofstadter is

   function Q(N: Positive) return Positive;
   -- computes recursively (and slowly!!!)
   -- the N-th element of Hofstadter's Q sequence

end Hofstadter
```

```
package body Hofstadter is

   function Q(N: Positive) return Positive is
   begin
      if N < 3 then
         return 1;
      else
         return Q(N - Q(N-1)) + Q(N - Q(N-2));
      end if;
   end Q;

end Hofstadter;
```

Reading task

Please read
⟨http://en.wikibooks.org/wiki/Ada_Programming/Tasking⟩

8.1: Mute Tasks – Tasks Without Communication

```
with Hofstadter, Ada.Text_IO;

procedure Mute_Tasks is
  task Odd_Inputs;
  task Even_Inputs;

  task body Odd_Inputs is
  -- deal with half of the inputs
    ...
  end Odd_Inputs;

  task body Even_Inputs is
  -- deal with the other half of the inputs
    ...
  end Even_Inputs;

begin
  null; -- the environment task is trivial
end Mute_Tasks;
```

first few outputs:

```
x H(x)
1 1
3 2
5 3
2 1
4 3
6 4
7 5
8 5
9 6
10 6
11 6
12 8
13 8
14 8
15 10
16 9
17 10
...
```

The Implementations of the Two Tasks

```
task body Odd_Inputs is
   X: Positive;
begin for I in 0 .. 49 loop
        X := 1 + 2*I;
        Ada.Text_IO.Put_Line(X'Img & Hofstadter.Q(X)'Img);
      end loop;
end Odd_Inputs;

task body Even_Inputs is
   X: Positive;
begin for I in 1 .. 50 loop
        X := 2*I;
        Ada.Text_IO.Put_Line(X'Img & Hofstadter.Q(X)'Img);
      end loop;
end Even_Inputs;
```

Tasks with a “start” message

- note that the two tasks are *almost identical*
- a better approach: a single task type “worker”, depending on one or more parameters
- …
- but task start running and “doing” something immediately after their declaration
- so the first thing our “worker” will do is to just waiting for the “start” message.
- these tasks are still mute (i.e., they listen and wait for “start”, but they don’t talk to each other)

The solution

```
with Hofstadter, Ada.Text_IO;

procedure Mute_Workers is
   task type Worker is
      entry Start(Name: Character;
                  Start, Offset, Count: Positive);
   end Worker;

   task body Worker is
      ...
   end Worker;

   E, O: Worker;

begin
   O.Start('O', 1, 2, 50);
   E.Start('E', 2, 2, 50);
end Mute_Workers;
```

first few outputs:

```
    x H(x)
O   1 1
O   3 2
O   5 3
O   7 5
E   2 1
E   4 3
E   6 4
E   8 5
E   10 6
E   12 8
E   14 8
E   16 9
E   18 11
O   9 6
O   11 6
O   13 8
O   15 10
```

The solution (continued)

```
task body Worker is
   My_Name: Character := '?';
   Val, Diff, Cnt: Positive;
begin
   accept Start(Name: Character;
                Start, Offset, Count: Positive)
   do
      My_Name := Name;
      Val:= Start; Diff := Offset; Cnt := Count;
   end Start;

   for I in 1 .. Cnt loop
      Ada.Text_IO.Put_Line(My_Name & " " &
            Val'Img & Hofstadter.Q(Val)'Img);
      Val := Val + Diff;
   end loop;
end Worker;
```

```
O  17 10
E  20 12
O  19 11
O  21 12
E  22 12
O  23 12
E  24 16
O  25 14
E  26 14
O  27 16
E  28 16
O  29 16
E  30 16
O  31 20
E  32 17
O  33 17
E  34 20
```

Mute tasks – advantages and disadvantages

(+) A very simple way to parallelize computations!

(+) No synchronization issues, no deadlocks, ...

(−) Not suitable to all problems, even if the problem itself is parallelisable!

(+) With p machines, ideally a p-times speed-up!

(−) It is statically determined which machine has to process which subtask!

(−) In our case, each process had to deal with exactly half the subtasks. What, if certain subtasks are computationally more intensive than others? For a p-time speed-up, we may need some form of **load balancing**. But then, our tasks aren't mute, any more!

8.2: Protected Objects

- A “protected object” in Ada encapsulates a shared object, such that different processes (or **task**s) can safely access the object.
- The processes don’t need to handle mutex-variables. The shared object takes care for that by itself.
- Three types of protected operations:

  function
  - result depends on object state, object state remains constant
  - different **task**s may call **function**s of the same object in parallel

  procedure
  - object state may change
  - exclusive access: if a **task** executes a **procedure**, all other operations are blocked **task**, all other operations are blocked

  entry
  - like **procedure**,
  - with an additional “barrier”

Declaration of Protected Type

General Scheme

```
protected type ...  is
   -- protected operations
   procedure ...;
   entry ...;
   function ...  return ...;
private
   -- data (i.e., attributes)
   Data:  Some_Type;
end ...;
```

Declaration and usage of Protected Objects

i.e., objects of a protected type

```
P1:  Some_Prot_Type;
P2:  Some_Prot_Type;
A: array(1..3) of Some_Prot_Type;

X := P1.My_Function(...);
P2.My_Procedure(...);
A(1).My_Entry(...);
```

Implementation of a Protected Type

```
protected body ...  is

   procedure ...  is
      ...; -- declarations
   begin
      ...; -- statements
   end ...;

   entry ...  when ...  is
      ...;
   begin
      ...;
   end ...;

   function ...  return ...  is
      ...;
   begin
      ...;
   end ...;

end ...;
```

Example: Shared_Counter.One_Way_Counter

Spec

```
package Shared_Counter is

   protected type One_Way_Counter(N: Positive) is
     procedure Increment(Value:  out Positive;
                         At_End: out Boolean);
      -- Value will output the values 1, 2, ... N.
      -- After calling Val N times, Value will stick at N.
      -- At_End is True <=> Increment has previously been
      --                    called at least N times
   private
      Current: Natural := 0;
   end One_Way_Counter;
```

Example: Shared_Counter.One_Way_Counter

Implementation

```
package body Shared_Counter is

   protected body One_Way_Counter is

      procedure Increment(Value:  out Positive;
                          At_End: out Boolean) is
      begin
         At_End := Current >= N;
                -- true if and only if Current sticks at N
         if not At_End then
            -- Current does not yet stick at N
            Current := Current + 1;
         end if;
         Value := Current;
      end Increment;

   end One_Way_Counter;
```

Example: Shared_Counter.Two_Way_Counter

Spec

```
   protected type Two_Way_Counter(Init: Natural := 0) is
      entry Increment;  -- barrier at Natural'Last
      entry Decrement;  -- barrier at 0
      function Value return Natural;
   private
      Val: Natural := Init;
   end Two_Way_Counter;

end Shared_Counter;
```

Shared_Counter.Two_Way_Counter

Implementation

```
protected body Two_Way_Counter is

   entry Increment when Val < Natural'Last is
   begin
      Val := Val + 1; -- no Constraint_Error,
                      -- thanks to the barrier
   end Increment;

   entry Decrement when Val > 0 is
   begin
      Val := Val - 1; -- no Constraint_Error
   end Decrement;

   function Value return Natural is
   begin
      return Val;
   end Value;

end Two_Way_Counter;
```

What is the point of the barriers?

- As the comments indicate, thanks to the barrieres we don't need to worry about arithmetic over- or underflow.
- But wouldn't we get the same with the following code?

```
   TWC: Two_Way_Counter;
...
   If TWC.Value > 0 then
      TWC.Decrement;
   end if;
```

That would seem to work, even if Decrement where a **protected procedure**, rather than an **entry**.

What is the point of the barriers? (2)

- If TWC.Decrement is a **protected procedure**, and not an **entry**, the proposed code

```
   TWC: Two_Way_Counter;
...
   If TWC.Value > 0 then
      TWC.Decrement;
   end if;
```

suffers from a race condition:

after checking TWC.Value $>$ 0 and
before calling TWC.Decrement,

another task object might change the value of TWC – and perhaps set it to zero.

Making Decrement an entry

with the barrier Val>0 has the following effect

- If Val is zero, then all **task**s calling TWC.Decrement will wait until one **task** calls TWC.Increment.
- Then one of the waiting **task**s can execute TWC.Decrement. (Or some **task**s can call TWC.Value or one **task** can call TWC.Increment once more . . .)

The race condition is gone for good!

Potentially Blocking Operations

- The code performed by a protected operation should run fast.
- So-called *potentially blocking* operations are prohibited and must not be called within a protected operation:

  During a protected action, it is a bounded error to invoke an operation that is potentially blocking.

- Potentially blocking operations include
  - most forms of IO,
  - calling a protected operation, and
  - calling a subprogram with calls a potentially blocking operation.
- The Ada compiler and its run time system are not required to detect a potentially blocking operation.

  If the bounded error is detected, `Program_Error` *is raised. If not detected, the bounded error might result in deadlock or a (nested) protected action on the same target object.*

8.3: Unhandled exceptions in Ada tasks

- If a task, for whatever reason, fails to handle an exception, the task silently “dies”. I.e., it terminates without telling the rest of the world about it.
- This surely is better than promoting the exception to the parent task – otherwise writing a reliable parent task would be almost impossible.
- However, this is really annoying for beginners, trying to learn about Ada **task**s. (It surely has driven me crazy!)
- I have written a package Armageddon as a debugging aid.
- Armageddon uses the package Ada.Task_Identification introduced with Ada 2005. To understand how Armageddon works, one needs to know about protected objects and protected callbacks.
- On the other hand, actually using Armageddon is extremely simple!

The package Ada.Task_Identification

```
with Ada.Task_Identification, Ada.Exceptions;

package Ada.Task_Termination is

  type Cause_Of_Termination is
    (Normal, Abnormal, Unhandled_Exception);

  type Termination_Handler is access
    protected procedure
     (Cause:  in Cause_Of_Termination;
      T: in Ada.Task_Identifikation.Task_Id;
      X: in Ada.Exceptions.Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler
                (Handler:  in Termination_Handler);

   ...   -- more getter and setter for handlers

end Ada.Task_Termination;
```

Protected callbacks

- The **type** Termination_Handler **is**

  access protected procedure (⟨Parameters⟩);
- This is quite similar – but not assignment compatible – to

  access procedure (⟨Parameters⟩);
- I.e., the user can call a Termination_Handler like any other procedure, except that any other **task** trying to call the same Termination_Handler must wait.

The Armageddon package

Armageddon has a single object End_Of, of some (anonymous) protected type. This type's only operation is the procedure The_World.

```
with Ada.Task_Termination,
     Ada.Task_Identification,
     Ada.Exceptions;

package Armageddon is
   use Ada;

   protected End_Of is
      procedure The_World
        (C: in Task_Termination.Cause_Of_Termination;
         T: in Task_Identification.Task_ID;
         X: in Exceptions.Exception_Occurrence);
           -- T=which task? C=why?
           -- If C=Unhandled_Exception then X=which exception?
   end End_Of;

end Armageddon;
```

What does Armageddon do?

```
with Ada.Text_IO;

package body Armageddon is

   protected body End_Of is

      procedure The_World(C: ...; T: ...; X:...) is
         ...
      end The_World;

   end End_Of;

begin
   Task_Termination.Set_Dependents_Fallback_Handler
     (End_Of.The_World'Access);
end Armageddon;
```

Armageddon installs the procedure End_Of.The_World as termination handler. Whenever a task terminates, End_Of.The_World is called.

What happens at the “End_Of.The_World”?

Cheating: We call a potentially blocking operation in a protected procedure.

```
procedure The_World
   (C: in Task_Termination.Cause_Of_Termination;
    T: in Task_Identification.Task_ID;
    X: in Exceptions.Exception_Occurrence) is

    use Text_IO, Task_Termination;
    T_Img: String renames Task_Identification.Image(T);

begin
    case C is
        when Normal =>
            null;
        when Abnormal =>
            Put_Line("Abnormal termination of task " & T_Img);
        when Unhandled_Exception =>
            Put_Line("Unhandled exception in task " & T_Img);
            Put_Line(Exceptions.Exception_Information(X));
    end case;
end The_World;
```

How can you use Armageddon?

. . . not use but with

Just add

```
with Armageddon;
```

to your context clause.

You don’t need to do anything else. Really!

8.4: Calling Protected Operations

no “mute” tasks any more

```
with Hofstadter, Ada.Text_IO, Shared_Counter, Armageddon;

procedure Load_Bal_Workers is

   Counter: Shared_Counter.One_Way_Counter(100);

   task type Worker is
      entry Start(Name: Character);
   end Worker;

   task body Worker is
      ...
   end Worker;

   A, B: Worker;
begin
   A.Start('A');
   B.Start('B');
end Load_Bal_Workers;
```

The Implementation of a Worker

```
task body Worker is
   My_Name: Character := '?';
   Current: Positive;
   Done: Boolean;
begin
   accept Start(Name: Character) do
      My_Name := Name;
   end Start;

   loop
      Counter.Increment(Current, Done);
      exit when Done;
      Ada.Text_IO.Put_Line(My_Name & " " &
        Current'Img & Hofstadter.Q(Current)'Img);
   end loop;
end Worker;
```

8.5: Rendezvous

- We have already seen a (rather trival) example for a rendezvous: the environment task did sent a “start” message to each worker.
- Below we will give another example. We revise the problem to solve: *Find one value n with $Q(n) = 24$.*
- We will have several worker **task** objects and one consumer **task** object. (By default, the number of worker objects will be eight.)
- The consumer waits for a message “found n” from one of the workers, and then prints n.
- The workers compute $Q(i)$ for different i, as before.

The Main Program (1)

```
with Hofstadter, Ada.Text_IO, Shared_Counter, Armageddon;

procedure Search_Q is

   task type Consumer is
      entry Found(N: Positive);
   end Consumer;

   task body Consumer is
      ... -- see below
   end Consumer;

   Wait_For_Result: Consumer;
```

The Main Program (2)

```
   Counter: Shared_Counter.One_Way_Counter(Positive'Last);
     -- used for the workers' load balancing

   task type Worker is
      -- no entry
   end Worker;

   task body Worker is
      ... -- see below
   end Worker;

   W: array (1..8) of Worker;
begin
   null;
end Search_Q;
```

The Consumer Implementation

```
task body Consumer is
   Result: Positive;
begin
   accept Found(N: Positive) do
      Result := N;
   end Found;
   Ada.Text_IO.Put_Line(Positive'Image(Result));
end Consumer;
```

The Worker Implementation

```
task body Worker is
   Current: Positive;
   Done: Boolean;
   Result: Positive;
begin
   loop
      Counter.Increment(Current, Done);
      exit when Done;
      Result := Hofstadter.Q(Current);
      if Result = 24 then
         Wait_For_Result.Found(Current);
      end if;
   end loop;
end Worker;
```

The accept statement

- During rendezvous, a statement of the form
  accept Name (Parameter) **do** ... **end** Name;
  is executed.
- We could simplify the Consumer implementation into

```
task body Consumer is
begin
   accept Found(N: Positive) do
      Ada.Text_IO.Put_Line(Positive'Image(Result));
   end Found;
end Consumer;
```

  That is possible, but would that be bad programming style. Why?
- Neither client nor server continue before the **accept** is done.
- The parameters of **accept** Name usually are copied into local variables – else they are lost after **end** Name.

The select-accept-structure

```
select when ⟨ First_Guard ⟩ =>
        accept ⟨ First_Entry ⟩ ⟨ Parameter ⟩ do
           ⟨ Statements ⟩
        end ⟨ First_Entry ⟩

or when ⟨ Second_Guard ⟩ =>
        accept ⟨ Second_Entry ⟩ ⟨ Parameter ⟩ do
           ⟨ Statements ⟩
        end ⟨ Second_Entry ⟩

or
        ⟨ more alternatives ⟩
end select;
```

The select-accept-structure (2)

- one or more **accept**-alternatives,
- depending on guards (logical conditions)

```
select when ⟨ Guard ⟩ =>
        accept ⟨ E1 ⟩ ⟨ ... ⟩ do
           ⟨ Statements ⟩
        end ⟨ E1 ⟩
or when ⟨ Guard ⟩ =>
        accept ⟨ E2 ⟩ ⟨ ... ⟩ do
           ⟨ Statements ⟩
        end ⟨ E2 ⟩
or
        ⟨ more alternatives ⟩
end select;
```

- *at least one guard* must be True
- it is an error if no guard is True
- if more than one guard is True
  - any one of the **accept**-alternatives with a True guard will be taken
  - which one is undefined

- you can replace "**when** true => **accept** ..." by "**accept** ..."
- many further variations (e.g., the **terminate**-alternative)

A problem

Specification:

```
type Device_Type is range 0 .. 1023;

task type Poll_For_Status is
   entry Start (Dev : in Device_Type);
end Poll_For_Status;
```

Problem statement:

- “Poll_For_Status” is a background task
- which shall run endlessly
- and ask the device “Dev” every three seconds for its status.
- The main program (namely, the calling task) shall continue normally – that is what makes “Poll_For_Status” a background task.

The Implementation

Die Implementation:

```
task body Poll_For_Status is
begin
   accept Start (Dev : in Device_Type) do
      loop delay 3.0; -- every third second
         Send_Status_Req (Dev);
                        -- send status request to Dev
      end loop;
   end Start;
end Poll_For_Status;
```

Questions:

- The authors of this program has been very unhappy. Why?
- How would you change the program?