Concurrent Programming in Ada

Andreas Rottmann

University of Salzburg, Austria

Department of Computer Science

arott@cosy.sbg.ac.at

OVERVIEW

CONCURRENCY AS A LANGUAGE FEATURE

Should concurrency be implemented as a language feature or should an OS interface be used?

Language defined

- ➜ More readable and maintainable programs
- ➜ Increased portability
- ➜ There may be no OS

Operating System defined

- ➜ Easier to compose programs of different languages
- ➜ Difficulties in mapping the language model to OS model
- ➜ OS standardize

A BIT OF ADA HISTORY

Two Ada standards:

Ada 83

- Concurrency as intrinsinc language feature
- Had some deficiencies

Ada 95

- Protected objects
- Better *preference control* (`requeue`)
- Better timing (`delay until`)
- OOP
- Core Language + Annexes

Clocks & Time

There are tow Ada packages providing time representation:

- `Ada.Calendar` — Wall clock time
- `Ada.Real_Time` — Monotonic, low-level time representation

The **`delay`** statement

- **`delay`**
- **`delay until`**

Delay Example

```
-- delay for (at least) 10 seconds
delay 10.0

-- delay until 10 seconds after the start of First_Action
Start := Clock;
First_Action;
delay until Start + 10.0;
Second_Action;

-- Note that this is not the same as:
Start := Clock;
First_Action;
delay (Start + 10.0) - Clock;
Second_Action;
```

Tasks

A *task* is a concurrently executing part of the program. Tasks are introduced using *task types*. Thus, *a task is a type instance*.

Syntax

```
-- task type
task type My_Task(Discriminant1 : Disc1_Type; D2 : Disc2_Type) is
   -- visible part
   entry Action1(Parameter : Par_Type);
   entry Action2(Parameter : Par_Type);
private
   --private part
   ...;
end My_Task;
```

```ada
-- task body
task body My_Task is
begin
   -- sequence of statements
   ...;
end My_Task;

-- declaration
T : My_Task;
```

```ada
-- anonymous tasks
task A_Task;

task body A_Task is
begin
   ...;
end A_Task;
```

Task hierarchies

Activation

- ➜ After **begin**, but before any statements
- ➜ All child tasks must have been activated

Finalization

- ➜ At **end**
- ➜ Unhandled exception

```
task Parent; -- parent task, using anonymous task type

task body Parent is

   Child1 : A_Task_Type;
   Child2 : Another_Task_Type;

begin
   -- the next line will be executed *after* Child1 and Child2 have
   --  been activated
   Some_Statement;
end;
```

THE RENDEZVOUS

A synchronization (and communication) point where two tasks "meet".

In Ada:

- A *task entry* declares a rendezvous point
- The **accept** statement specifies actions to be executed when the entry is called

```
accept Entry_Name(Family_Index)(P : Parameter) do
   -- sequence of statements
exception
   --exception handling part
end Entry_Name;
```

Primes by sieve

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Sieve1 is

   task type Sieve is
      entry Pass_On(Int : Integer);
   end Sieve;

   type Sieve_Ptr is access Sieve;

   function Get_New_Sieve return Sieve_Ptr is
   begin
      return new Sieve;
   end Get_New_Sieve;

   Limit : constant Positive := 1000;
   Num : Positive;
   S : Sieve_Ptr := Get_New_Sieve;
```

Primes by sieve, contd.

```
task body Sieve is
   Next_Sieve : Sieve_Ptr;
   Prime, Num : Natural;

begin
   accept Pass_On(Int : Integer) do
      Prime := Int;
   end Pass_On;
   Put(Prime); -- output prime number
   New_Line;
   loop
      accept Pass_On(Int : Integer) do
         Num := Int;
      end Pass_On;
      exit when Num rem Prime /= 0;
   end loop;
```

Primes by sieve, contd.

```
      -- pass the number on to a new sieve task
      Next_Sieve := Get_New_Sieve;
      Next_Sieve.Pass_On(Num);
      loop
         accept Pass_On(Int : Integer) do
            Num := Int;
         end Pass_On;
         if Num rem Prime /= 0 then
            Next_Sieve.Pass_On(Num);
         end if;
      end loop;
   end Sieve;

begin
   Num := 3;
   while Num < Limit loop
      S.Pass_On(Num);
      Num := Num + 1;
   end loop;
end Sieve1;
```

THE `select` STATEMENT AND THE RENDEZVOUS

There are four forms of the **`select`** statement:

① Selective accept

② Conditional entry call

③ Timed entry call

④ Asynchronous select

Selective accept

➜ Wait for more than one rendezvous

➜ Time out if no rendezvous happened in a specified time period

➜ Check if a rendezvous is immediately possible

➜ Terminate if no clients can possibly call its entries

Waiting for more than one rendezvous

```
task type Server is
   entry Service1(...);
   entry Service2(...);
end Server;

task body Server is
begin
   loop
      select
         accept Service1(...) do
            -- handle request for Service1
         end Service1;
      or
         accept Service2(...) do
            -- handle request for Service2
         end Service2;
      end select;
   end loop;
end Server;
```

GUARDED ALTERNATIVES

Guard

- ➜ Boolean expression
- ➜ Evaluated when the select statement is executed
- ➜ If true, the alternative is considered

```
select
   when Boolean_Expression =>
      accept Service1(...) do
         -- handle request for Service1
      end Service1;
or
   ...
end select;
```

Delay alternative

Often necessary to time-out; for instance a server might have to do some periodic task:

```
task body Server is

   Period := Time_Span := To_Time_Span(10.0);
   Next_Cycle : Time := Clock + Period;

begin
   loop
      -- do periodic maintainance
      select
         accept Do_Maintaince_Now;
      or
         delay until Next_Cycle;
         Next_Cycle := Next_Cycle + Period;
      end select;
   end loop;
end Server;
```

THE ELSE PART

Can be used to only engage in a rendezvous if there are entry calls on it:

```
task body Server is
begin
   loop
      select
         accept Service1(...) do
            -- handle request for Service1
         end Service1;
      else
         -- do something in reaction that Service1 was not requested
         ...;
      end select;
      accept Service2(...) do
         -- handle request for Service2
      end Service2;
   end loop;
end Server;
```

THE TERMINATE ALTERNATIVE

- Single **terminate** statement
- Task will complete if there are no clients that can possibly call its entries

```
task body Server is
begin
   loop
      select
         accept Service1(...) do
            -- handle request for Service1
         end Service1;
      or
         terminate;
      end select;
   end loop;
end Server;
```

Protected Objects

- ➜ For shared resources
- ➜ Encapsulate data, only accessible via protected subprograms and entries
- ➜ Mutual exclusion is guaranteed for data access
- ➜ Have a body
- ➜ Similar to the “monitor” concept

```
-- A protected type
protected type A_Protected_Type is
   procedure Operation1(P : Parameter);
   entry Entry1(P : Parameter);
private
   -- place here private operations and items here
end A_Protected_Type;
```

Protected entries

- Guarded by a boolean expression (barrier) inside the body of the protected object
- Callers suspended until barrier becomes true and no other tasks active inside protected object
- Provide condition synchronization

Using Protected Objects

Semaphore

```
protected type Semaphore(Initial : Natural := 0) is
   entry Wait; -- P operation
   procedure Signal; -- V operation
private
   Value : Natural := Initial;
end Semaphore;

protected body Semaphore is
   entry Wait when Value > 0 is
   begin
      Value := Value - 1;
   end Wait;

   procedure Signal is
   begin
      Value := Value + 1;
   end Signal;
end Semaphore;
```

Bounded Buffer

```
Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0..Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
   entry Get(Item : out Data_Item);
   entry Put(Item : in Data_Item);
private
   First : Index := Index'First;
   Last : Index := Index'Last;
   Number_In_Buffer : Count := 0;
   Buf : Buffer;
end Bounded_Buffer;
```

Bounded Buffer, contd.

```
protected body Bounded_Buffer is
   entry Get(Item : out Data_Item) when Number_In_Buffer /= 0 is
   begin
      Item := Buf(First);
      First := First + 1;
      Number_In_Buffer := Number_In_Buffer - 1;
   end Get;

   entry Put(Item : in Data_Item) when Number_In_Buffer /= Buffer_Size is
   begin
      Last := Last + 1;
      Buf(Last) := Item;
      Number_In_Buffer := Number_In_Buffer + 1;
   end Put;
end Bounded_Buffer;
```

THE END

Thanks for Your Attention!