Introduction to Lab 1

Real-Time Programming using Ada

Aleksandar Zeljić
<aleksandar.zeljic@it.uu.se>

4th September 2015

Lab 1: Real-Time Programming using Ada

- Lab goals:
  - Basic understanding of Ada
  - Handling of time and task synchronization
- Lab preparation:
  - *Work in 3-person groups*
  - Lab will be done on Friday, 11.9. in room 1515
  - *Make sure your Unix/Solaris login works!*
  - Have a look at "Ada Quick Start" on the lab homepage
    `http://www.it.uu.se/edu/course/homepage/realtid/ht15/lab1`
- Lab report:
  - Ada code to all 4 problems
  - Illustrative test-runs of all 4 programs
  - Via Student Portal
  - *Deadline: Tue, 15.9. at 23:59*

Ada: Some Facts

- The history:
  - Originally designed by/for US Department of Defense (read: Military)
  - Target: Embedded and real-time systems
  - Development: Late 1970s; first compiler in 1983
- The language:
  - Imperative, object-oriented language
  - Statically typed
  - Case insensitive
  - Concepts “time” and “task” direct language features
- The compilers:
  - Commercial and free implementations available
  - We use GNAT (part of GNU compiler collection)

First Example: Hello World!

Listing: hello_world.adb

```
1 -- File: hello_world.adb
2 with Ada.Text_IO;   -- Use package Ada.Text_IO
3 use Ada.Text_IO;    -- Integrate its namespace
4
5 procedure hello_world is
6     Message : constant String := "Hello World";
7 begin
8     Put_Line(Message);
9 end hello_world;
```

Note:

- Comments start with --
- Used package `Ada.Text_IO` for basic I/O (via procedure `Put_Line`)
- Filename equals procedure's name
- Assignment via ":=", comparison via "=" and "/="

First Example: Let's run it..

Example Run: hello_world.adb

```
$ gnatmake hello_world
gcc-4.3 -c hello_world.adb
gnatbind -x hello_world.ali
gnatlink hello_world.ali
$ ./hello_world
Hello World
$
```

(No surprises here.)

(Some) Basic Program Statements

- Basic statements and structures:
  - `null` – Does nothing
  - `:=` – Assignment
  - `if/then/else` – Conditional execution
  - `loop/exit (when)` – Loop execution

Example: Basic statements

```
1  x := y;  -- Assign y to x
2  if (x = y) then
3      null;
4  else
5      z := 0;
6      x := 42;  -- Note: Several statements!
7  end if;
8  loop
9      z := z + 1;
10     exit when z > 100;  -- Works also with only "exit"
11 end loop;  -- Loops may be nested and while/for annotated
```

Types

- Basic types: Boolean, Integer, Float, Character, String
- New types via subtype, mod, array, record
- New typenames via type

Example: Types

```
1 subtype Die is Integer range 1 .. 6;   -- One die
2 type Dice is array (0 .. 22) of Die;   -- 23 dice
3 type Complex is
4         record
5             r : Float;   -- real component
6             i : Float;   -- imaginary component
7         end record;
8 d: Dice;   -- Array access: d(1) := 3;
```

Programs and Procedures

- Main procedure: Procedure without arguments
  - Same name as the file name
- Procedures:
  - May use in/out parameters (in/out parameters are read-only/write-only)
  - Local variables
  - Local procedures/functions/tasks/...

Example: Procedure's Definition

```
1 procedure Name_of_Proc(var1: in Integer; var2: out String)
      is
2     -- local variables
3     -- definitions of local procedures/functions/tasks/..
4 begin
5     -- code of procedure
6 end Name_of_Proc;
```

Functions

- Functions are almost like procedures
- Use "return" keyword:
  - For the return type
  - For returning the result

Example: Function's Definition

```
1 function Name_of_Func(x1, x2: Float) return Float is
2     -- local variables
3     -- definitions of local procedures/functions/tasks/..
4 begin
5     return x1 * x2;
6 end Name_of_Func;
```

Text Output

- Output with linebreak: `Ada.Text_IO.Put_Line("Hej!")`
- Output without linebreak: `Ada.Text_IO.Put("Hej!")`
- String representation of `Y` of type `X`: `X'Image(Y)`
  - Uses `Image` attribute of type `X`

Example: Output of various types

```
1  use Ada.Text_IO;
2  i: Integer; d: Duration;
3  begin
4       i := 1 + 2;
5       Put_Line(Integer'Image(i));
6       --print non-string type
7       d := 10.0;
8       Put("Waiting time: ");
9       Put_Line(Duration'Image(d));
10 end Foo;
```

Time in Ada

- Package: `Ada.Calendar`
  - Type for *relative* time: `Duration`
  - Type for *absolute* time: `Time`
  - For getting present time: `Clock` (of type `Time`)

Example: Time

```
1 -- Assume: x: Duration; t, t1, t2: Time;
2
3 delay 10.0; -- Delays execution by 10s (Float literal!)
4 delay x;    -- Use Duration type here
5
6 t := Clock; -- Read present time
7 delay until t + 10.0;   -- Delays until t + 10 seconds
8 delay (t2 - t1);        -- Time difference is Duration
9 delay until t + x;      -- Time + Duration gives Time
```

Note: Delay *may be longer*! (granularity, no irq, scheduler..)

Time in Ada

Anything wrong with this code? What's the output?

Example: Periodic Loop?

```
1  -- Assume: Use package Ada.Text_IO and Ada.Calendar
2  procedure drift is
3      Message : constant String := "The time is";
4      Period : Duration := 1.0;
5      Start_Time: Time := Clock;
6  begin
7      loop
8          Put(Message);
9          delay Period;
10         Put_Line(Duration'Image(Clock - Start_Time));
11     end loop;
12 end drift;
```

An example run

Example: drifting

```
The time is 1.000089000
The time is 2.000209000
The time is 3.000300000
The time is 4.000390000
The time is 5.000481000
The time is 6.000569000
The time is 7.000658000
The time is 8.000748000
The time is 9.000837000
The time is 10.000928000
```

Problem: Delay too long, real period longer, causing drift

Time in Ada: Avoiding Drift

- Better: Avoid drift by using `delay until`

Example: Periodic Loop, revised version

```
1  -- Assume: Use package Ada.Text_IO and Ada.Calendar
2  procedure avoid_drift is
3      Message : constant String := "The time is";
4      Period : Duration := 1.0;
5      Start_Time : Time := Clock;
6      Next_Time: Time := Start_Time + Period;
7  begin
8      loop
9          Put(Message);
10         delay until Next_Time;
11         Next_Time := Next_Time + period;
12         Put_Line(Duration'Image(Clock - Start_Time));
13     end loop;
14 end avoid_drift;
```

An example run of the two versions

Example: drifting

```
The time is 1.000089000
The time is 2.000209000
The time is 3.000300000
The time is 4.000390000
The time is 5.000481000
The time is 6.000569000
The time is 7.000658000
The time is 8.000748000
The time is 9.000837000
The time is 10.000928000
```

Example: avoid drifting

```
The time is 1.000072000
The time is 2.000069000
The time is 3.000069000
The time is 4.000070000
The time is 5.000069000
The time is 6.000068000
The time is 7.000067000
The time is 8.000064000
The time is 9.000070000
The time is 10.000069000
```

Problem: Delay too long, real period longer, causing drift

Tasks: Implementation

Example: Task declaration and definition

```
1  -- Declaring a task
2  task My_Task is
3      --entry calls declarations
4  end My_Task;
5
6  -- Defining a task body
7  task body My_Task is
8      -- local declarations
9      begin
10         -- code of the entry calls
11     end My_Task;
```

Tasks: Communication and Synchronization

Using entry calls to communicate

Example: Task declaration and definition

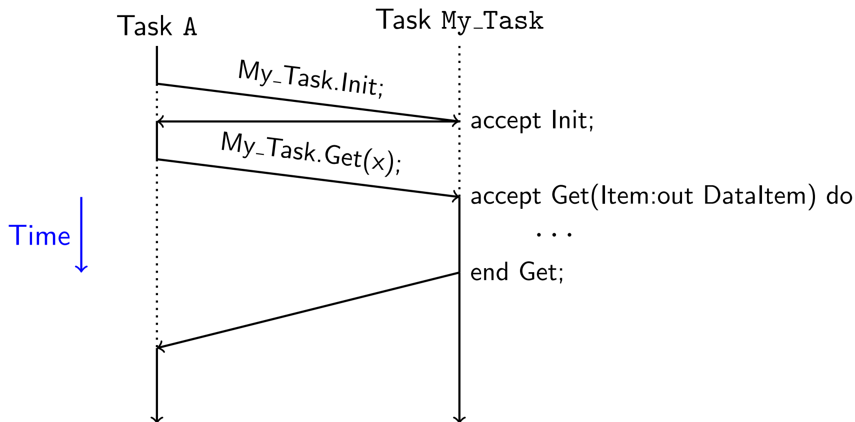
```
1  -- Declaring a task
2  task My_Task is
3      entry Init;
4      entry Get(Item : out DataItem);
5      entry Put(Item : in DataItem);
6  end My_Task;
7
8  -- Defining a task body
9  task body My_Task is
10 -- Declarations needed by task code go here.
11     begin
12     -- Task code goes here (see next slide!)
13     end My_Task;
```

Example task body

Example entry body

```
1 accept Init;  -- Wait for a call to 'Init' (using My_Task.
     Init)
2 accept Get(Item: out DataItem) do -- same, with parameters
3 ...
4 end Get;
```

Tasks: Concurrent Execution with Synchronization



What if we want to wait for one of the calls

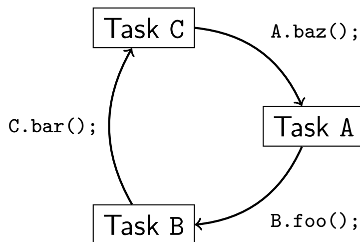
selective wait

```
1 --wait for one of the calls
2 select
3 accept Init;
4 ...   -- Statements after Init call
5 or
6 accept Get(...) do
7 ...   -- Statements for Get call (Caller blocked!)
8 end Get;
9 end select;
```

One can add timeout using *delay*

Tasks: Potential Deadlocks

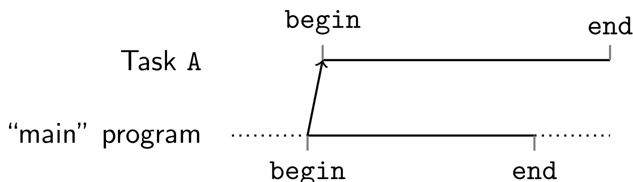
- Synchronization can cause *deadlocks*:



- A, B, C are waiting for each other!

Tasks: Lifetime Cycle

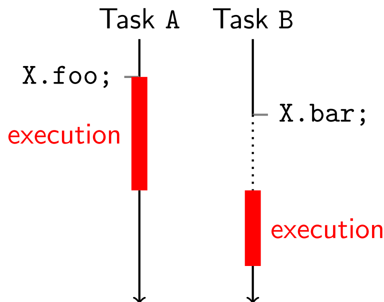
- Start running *as soon as they are created*
  - In scope of "main" procedure: At program start!
  - Otherwise: When scope is called
  - Use explicit synchronization if necessary (like an "Init" entry)
- End when control flow reaches "end"
- Program ("master task") does not end before all tasks are done (!)



- Tasks can also be defined as *types*
  - Variables of that type are the running instances
  - Start running as soon as variables are created

Protected Types

- Provide *mutual exclusion*:
  - For concurrent execution
  - Only one procedure/entry of instance can run at any time
  - Can implement mutually exclusive data access to an object `X`



- Note: Tasks can also provide mutually exclusive data access
  - Execute just one "request" at a time

Protected Types: Implementation

Example: Protected Types

```
1  protected Integer_Buffer is
2      entry Insert (i : in  Integer);
3      entry Remove (i : out Integer);
4      private
5          Buffer : Integer;
6          Empty  : Boolean := True;
7  end Integer_Buffer;  -- Note: could also be defined as a type
8
9  protected body Integer_Buffer is
10     entry Insert (i : in  Integer) when Empty is
11     begin
12         Buffer := i; Empty := False;
13     end Insert;
14     entry Remove (i : out Integer) when not Empty is
15     begin
16         i := Buffer; Empty := True;
17     end Remove;
18 end Integer_Buffer;
```

Lab Assignment

Or: What You Are Supposed To Do

- Part 1: Cyclic Scheduler
  - Implement a cyclic schedule of 3 procedures
  - Avoid "drift" of the schedule
- Part 2: Cyclic Scheduler with Watchdog
  - Add watchdog task to Part 1
  - `select` with `delay` may be useful
- Part 3: Process Communication
  - Implement Producer, Consumer and FIFO Buffer tasks
  - ..and communication between them
- Part 4: Data Driven Synchronization
  - Same as Part 3 but with Protected Type

The End

Questions?