TASKS

Week 3 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer based on material by Alistair Rendell

Pre-Laboratory Checklist

- ❑ **Skills: You can edit, load and run Ada programs (you remember `gps&`).**
- ❑ **You understand basic, static type concepts.**
- ❑ **You understand in full what information hiding and exception handling means.**
- ❑ **You know some aspects of generic programming (static polymorphism).**

Objectives

The objective of this exercise is for you to create and handle your first concurrent entities – called tasks here. Creating tasks in Ada is syntactically easy, and you will also learn in this lab about the basic features of a task and what you can do with it.

Interlude: Most basic tasks

In Ada each task corresponds to a data object, called a task object. Associated with each task object is a sequence of statements to be executed and an optional interface for synchronization and/or communication with other tasks.

Like other data objects, task objects belong to types, in this case task types. All the task objects in the same task type share certain characteristics:

- Every task independently executes the same sequence of statements.
- Every task instance has its own copy of local variables (which are initialized in the same way).
- Every task instance has its own set of independent interfaces, even though they all share the same syntactical form as given by the task type.

All task types are `limited`! That is tasks cannot be copied or tested for equality.

Like for procedures or functions (and other complex objects which we will learn about later), tasks are declared in two parts: a task (type) declaration and a task body. As with function definitions before we can place the declaration part into a specification package and the body part into the implementation (body) package. Just like with functions, the task (type) declaration specifies the interface to the task (type) (its **entries**), while the task body contains the statements executed by this task or by tasks of this type. As with any scoped programming language, you can add declarations to the body of a task, which will be local to this task (type).

The following syntax is used to declare a task (type):

```
task [type] identifier [discriminant-part] [is
   entry-declaration
   ...
   entry-declaration
end identifier];
```

You will have noticed that I mention “type” always in parenthesis and it is optional for the task declaration. Leaving it out is simply a shortcut to declare a single task instance without needing to introduce a task type first. Our very first example below will be so simple that we won’t need an explicit task type. In fact we won’t even need any interface to the first tasks and so we can declare the most simple kind of tasks (a singleton task) just with:

```
task identifier;
```

The body part of a task looks very similar to a procedure, yet there are no parameters which could be passed:

```
task body identifier is
   declarative-part
begin
   sequence-of-statements
exception
   handler
   ...
   handler
end identifier;
```

Tasks are “running” on declaration (or allocation as we will see later). To be more precise, Ada guarantees that a declared task is up and running before the first statement after the next `begin`. In the above case of a singleton task, the single task will be in state “running” right after it has been declared with `task identifier;`. All declarations inside a task body are local to the specific instance of a task (of course in case of a singleton task, there is only one) and thus data which need to be (memory) shared between tasks need to be declared outside of both communicating tasks. There will be a lot more on this data sharing subject a little later, as it is a crucial part of concurrent systems.

Exercise 1: Let there be tasks

Let’s have a look at a very simple example and make our first observations what all this actually means.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Two_Tasks is
   task First_One;
   task Second_One;
   task body First_One is
   begin
      Put (“Hello .. “);
      Put_Line (“and goodbye world from task 1”);
   end First_One;
```

```
      task body Second_One is

      begin
         Put (“Hello .. “);
         Put_Line (“and goodbye world from task 2”);
      end Second_One;

   begin
      Put (“Hello .. “);
      Put_Line (“and goodbye world from Two_Tasks”);
   end Two_Tasks;
```

Before you go any further, make a prediction of what will happen if you run this program and explain it to your neighbour.

Once you agree with your neighbour that this is exactly what is to be expected, download this simple program from the course-site and run it[1]. What actually happened? Run it again. What happened this time? Is your computer broken? Does your neighbour see the same thing?

Maybe it would help to look at the watch while things are progressing along. Here is a simple program which will print out the current time in a loop:

```
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;

procedure Timestamp is

   No_of_Iterations     : constant Positive := 10;
   Delay_Per_Iteration : constant Duration := 1.0;

   Start_Up_Time        : constant Time     := Clock[2];

   subtype Repeat_for is Positive range 1 .. No_of_Iterations;

begin
   for i in Repeat_for loop[3]

      declare
         Now : constant Time := Clock[4];

      begin
         Put (Day_Number'Image   (Day     (Now)) & “.”);
         Put (Month_Number'Image (Month   (Now)) & “.”);
         Put (Year_Number'Image  (Year    (Now)) & “ at”);
         Put (Day_Duration'Image (Seconds (Now)) & “ seconds - since start up:”);
         Put (Duration'Image (Now - Start_Up_Time));
         New_Line;
         delay Delay_Per_Iteration;
      end;
   end loop;
end Timestamp;
```

Run this time-stamping program (it is part of your previous download). What do you observe?

- How are the two different “second” measurements different?
- What is the unit of `Duration`?
- Everything in this program seems to be declared as constants – can this be right?

1 Run this on the command line (terminal) instead of from inside GPS. Depending on the operating system, GPS may bundle outputs and won’t show you intermediate results.
2 Apparently `Clock` is function which will return different values at different times … hence the name …
3 Of course I could have just written `for i in 1 .. 10 loop`, yet I thought to be cleaner in style from the very beginning and avoid constants (other than obvious corner values like `0` or `1`) inside code sections as much as possible.
4 This call to `Clock` will be evaluated every time this declare block is entered and the `Now` constant is declared, i.e. with every loop iteration. Why am I doing this? Simply because constants are nicer than variables, in case you will ever need to run your programs through a certification process or formally verify your code.

Now change the line:

```
delay Delay_Per_Iteration;
```

to:

```
delay until Start_Up_Time + i * Delay_Per_Iteration;
```

Run your program again and try to spot the difference. You may want to open two terminal windows side-by-side with one version running in each window to spot the differences more easily. The difference is vital for real-time systems as you may learn later. As a general reflex, programmers coming from a high-integrity or real-time background will always prefer absolute delay expressions (implemented by the `delay until` statement in Ada) if they have a choice.

Now to your final job in this exercise: Create the `Two_Tasks_Timestamped` program such that your tasks are not only friendly (as in the original `Two_Tasks` program), but also print out the time stamps when they were producing those texts. (Copy most of the Two_Tasks program and amend it with appropriate time-stamp outputs.)

Then answer the following questions:

- In the first shot, your screen output was probably somewhat of a mess. Why was that? Can you repair that?
- How many tasks are running in your `Two_Tasks` program? As I ask like that: the answer is obviously not two – but how many are actually there?

Submit your code to the *SubmissionApp* under “Lab 3 Two tasks timestamped“. We will check out most of your code and will attempt to give you feedback on all levels of coding.

Exercise 2: Sharing tasks (the naïve way – don’t do this at home)

We said above, in order to share data between tasks, the data needs to be declared outside of those tasks. So let’s declare a variable `Sum` outside the scope of a simple task type and then instantiate a few tasks that work on this variable:

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

procedure Counter_Test is

   Sum : Natural := 50;

   task type Counter (Id : Positive; Goal : Natural)[5];

   task body Counter is

   begin
      while Sum /= Goal loop
         Sum := (if Sum > Goal then Sum - 1 else Sum + 1)[6];
         Put (Natural’Image (Sum));
         delay 0.0; -- try leaving out this delay statement!
      end loop;

      New_Line;
      Put_Line (“Counter task” & Positive’Image (Id) &
                “ terminates with sum being:” & Natural’Image (Sum));
   end Counter;
```

5 Observe that this task type has discriminants so that we can later instantiate a few of those tasks with different values for those discriminants.

6 Conditional expressions make code neater and more compact. Your functional programming background will probably help you writing code in such succinct forms in imperative languages as well.

```
      Counter_1 : Counter (1, 30);
      Counter_2 : Counter (2, 40);
      Counter_3 : Counter (3, 60);
      Counter_4 : Counter (4, 70);
   begin
      New_Line;
      Put_Line ("Counter_Test terminates with sum being:" & Natural'Image (Sum));
   end Counter_Test;
```

It is important that you answer the questions below before you run this program:

- Will this program terminate?
- What will it display (if anything)?
- Very important: what will be the value (or the possible values) for the final "`Counter_Test terminates with sum being: ..`" output)?

Discuss your answers with other people in the lab who are just doing the same exercise – otherwise your tutor.

Now download and run the program. How close were you? Do a number of tests and see whether the value at the end of the each task is actually always identical with its goal value. You will find that this will not always be the case. If you can answer why this is you have understood the essence of this lab.

Exercise 3: Synchronize just enough

For this advanced exercise, take the `Counter_Test_Synchronized` program (which is included in the previous download, yet is just a direct copy of the `Counter_Test` program) and make sure that the value of sum (as it is displayed at the end of each task) is actually the desired goal value. This will be very easy once you learn about the tools to protect data against improper mutual access (protected objects or synchronous message passing as it will be introduced in upcoming labs). Yet the point of this exercise is to implement this with nothing but atomic shared variables. It is painful and educational to do so (this is what your parents always referred to as *character building*). Start with only two tasks and then see whether you can implement a mutual exclusion method which could handle a random number of tasks. You will likely make mistakes in at least some of your attempts – so did I. You succeeded if you amended the `Counter_Test_Synchronized` program to always read out and display the correct results at the end – and also convinced yourself that you will never ever do it like that again. If you still believe this is a great way of handling concurrency, come to us for more exercises.

Submit your code to the *SubmissionApp* under "Lab 3 Atomic sync counter" and receive the code for the same module from a fellow student (all anonymously via the *SubmissionApp*). The code is run as you wrote it – no additional tests are being added. Check out how your colleague solved this problem and provide feedback for him/her. Be specific and constructive with your feedback. If you think it doesn't work then give an example where it breaks.

**MAKE SURE YOU LOGOUT
TO TERMINATE YOUR SESSION!**

Outlook

Next week you will be handling communicating tasks and learn more about how long tasks live.