TASK LIFE SPANS

Week 5 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer

Pre-Laboratory Checklist

- ❑ **You have read this text before you come to your lab session.**
- ❑ **You can define basic tasks.**
- ❑ **You can control the memory based synchronization needs of your tasks precisely.**

Objectives

This lab introduces you to the life span of tasks. Tasks are depending on each other while following exactly specified rules about their creation and destruction which you will learn and exercise in this lab. You will also put your new knowledge to use in order to write your first performance code, letting all CPUs in your computer work hard on one problem.

Interlude: Tasks – the full story

You saw the basic task syntax last week. Here comes the full story (from the Ada Reference Manual, section 9.1):

```
task_type_declaration ::=
   task type defining_identifier [known_discriminant_part] [aspect_specification]
   [is [new interface_list with] task_definition];

single_task_declaration ::=
   task defining_identifier [aspect_specification]
   [is [new interface_list with] task_definition];

task_definition ::=
      {task_item}
   [ private
      {task_item}]
   end [task_identifier]

task_item ::=
   entry_declaration | aspect_clause

task_body ::=
   task body defining_identifier [aspect_specification] is
      declarative_part
   begin
      handled_sequence_of_statements
   end [task_identifier];
```

There are a few new features introduced in those full definitions. Here are the details:

- Aspects of a task define attributes of a task which will come in handy to control in detail what is supposed to happen. For example you can define priorities, deadlines, memory usage, hard-assigned processors etc:

  ```
  task Neatly_Specified_Task
     with Priority          => Priority'Last,
          Relative_Deadline => Clock + Some_Delay,
          CPU               => 2;
  ```

  You won't need any of those aspects for a while, but keep a mental marker that there are control knobs for you, if you need them. Also take note, that in case that the Ada runtime runs on top of a desktop operating system (like on your computer) some of these aspects may be ignored by the underlying operating system. Ada-based systems which do require predictable control over those aspects will run on top of a real-time operating system or "bare bone", i.e. without an operating system.
- Another feature is that you can use a task definition to implement a interface, in the style of object oriented programming which are you familiar with from previous courses. Yet, there is an important restrictions to this derivation method: tasks (as other concurrent entities which you will learn about later) which implement a set of interfaces are final. No further task type can be derived from such an implementation. This is a solution to the **inheritance anomaly**[1] which occurs otherwise when you combine concurrency with object orientation.

  So in essence, you can chose to implement existing interfaces with tasks e.g. like this:

  ```
  task type Protected_Queue_Task is new Queue_Interface with

     overriding entry Enqueue (Item :     Element);
     overriding entry Dequeue (Item : out Element);

  end Protected_Queue_Task;
  ```

- You will also have noticed the option to declare `private` entries for a task (which are invisible to external tasks). This seems silly as the whole point of declaring those entries appears to be that other tasks can communicate with this task. You will nevertheless find an important use for those private entries in more complex server tasks: Clients will need to be "requeued" to internal entries – potentially multiple times – in order to allow external calls to be handled in multiple stages.
- For now we will only need discriminants and public entries to declare our tasks. This could for instance look like this:

  ```
  subtype UId is Natural range 0_000_000 .. 9_999_999;

  task type Student (Id : UId := UId'First) is

     entry Do_Laboratory (Lab_No    : Positive);
     entry Do_Assignment (Assign_No : Positive);

  end Student;
  ```

  … which declares a task type for a typical student who will only react to the outside world when laboratory work or assignments need to be done. An instance of such a student can be given a uid, but if omitted the uid for this new task will be set to zero:

  ```
  Me        : Student (Id => 5324532);
  Anonymous : Student; -- Anonymous will automatically gain uid zero
  ```

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.

Exercise 1: Your tasks in scope

Tasks which are created by means of a static declaration are bound to the scope from within they are created. If everything goes to plan, then tasks which are created within a certain scope need to complete before this scope can itself complete. Let's investigate what happens in this simple code snippet:

```
   …
   Task_Instance : Task_Type;
   …
begin
   …
end;
```

The activation of `Task_Instance` occurs before the first statement after the `begin` is executed. At the `end` of the scope, the task which contains this snippet is *delayed* until `Task_Instance` is completed and terminates. This easy to understand relationship between the creator task and the created task (usually referred to as "parent task" and "child task" resp.) can handle the vast majority of real-world concurrency jobs – so make sure you fully understand what's going on here. Take note that it is not important where Task_Type is defined (as long as it is visible from within this scope).

Things will take a more interesting turn if tasks are created by means of dynamic allocation statements (i.e. by using `new`). Read the following program which contains all possible options where and how tasks can be created:

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

procedure Task_Scopes is

   Start_Up_Time : constant Time := Clock;

   procedure Put_Line_Time (S : String) is

   begin
      -- Prefixes the time since startup as seconds with two decimal places
      Put (Float (Clock - Start_Up_Time), 1, 2, 0); Put_Line (" seconds: " & S);
   end Put_Line_Time;

   task type Outer_Type;
   type Outer_Type_Ptr is access Outer_Type;

   task body Outer_Type is

   begin
      delay 0.6; Put_Line_Time ("-- End of an outer task");
   end Outer_Type;

   task Static_Outer_Task;

   task body Static_Outer_Task is

   begin
      delay 0.1; Put_Line_Time ("Start of Static_Outer_Task");
      declare

         task type Inner_Type;
         type Inner_Type_Ptr is access Inner_Type;

         task body Inner_Type is

         begin
            delay 0.6; Put_Line_Time ("-- End of an inner task");
         end Inner_Type;
```

```
          task Static_Inner_Task;
          task body Static_Inner_Task is
          begin
             delay 0.2; Put_Line_Time ("Start of Static_Inner_Task");
             declare
--                 Inner_Task_Instance : Inner_Type;
--                 Outer_Task_Instance : Outer_Type;
--                 Dynamic_Inner_Instance : Inner_Type_Ptr := new Inner_Type;
--                 Dynamic_Outer_Instance : Outer_Type_Ptr := new Outer_Type;
             begin
               delay 0.3; Put_Line_Time ("End of Static_Inner_Task declare block");
             end;
             delay 0.1; Put_Line_Time ("End of Static_Inner_Task");
          end Static_Inner_Task;
       begin
          delay 0.4; Put_Line_Time ("End of Static_Outer_Task declare block");
       end;
       delay 0.1; Put_Line_Time ("End of Static_Outer_Task");
    end Static_Outer_Task;
 begin
    delay 0.2; Put_Line_Time ("Start of main scope");
    delay 0.2; Put_Line_Time ("End of main scope");
 end Task_Scopes;
```

Dynamically allocated tasks are bound to the scope in which the pointer-type to the task is declared (for instance: `type Outer_Type_Ptr is access Outer_Type;`). This can be anywhere between the declaration of the task type and the allocation statement itself. Note that a dynamically allocated task can therefore also live longer than the pointer variable referring to it.

Now your actual exercise: Read the above code carefully and write down the sequence of texts which appear on the terminal for all four cases (uncomment exactly one of the four lines in the innermost declare block for each case).

Now download and run the program. Is everything happening according to plan? Understand exactly why the actual output and your notes differ (if you didn't get it spot on in one).

Exercise 2: Concurrent Mergesort

Mergesort lends itself in an obvious way to concurrent implementations. With every split step you have two completely independent subproblems at your hand which can be solved concurrently. You could now implement this naïvely and actually branch off into two new tasks with every split step. Yet the ultimate goal of a concurrent sorting algorithm us not to show off how concurrent it will become but to enhance the performance of the algorithm. In this case concurrent operations will only speed things up as long as there are free CPU cores available. Beyond this point things will slow down again as there is overhead introduced with the creation of new tasks – yet with no further performance benefit as the new tasks will ultimately run in sequence. Thus concurrent Mergesort tries to exploit the physical concurrency of your system and will consider the number of physical CPU cores available before it branches into concurrent tasks.

In Ada you have a package `System.Multiprocessors` which provides you with actual number of cores which have been found on your system. This package provides a type which you could use to assign a specific task to a specific CPU, but much more importantly it provides a function `Number_Of_CPUs` telling you the .. well. For the keen: this is actually the number of CPU cores, i.e. not the number of hyperthreads.

I prepared all that for your convenience:

```
generic
   type Element       is private;
   type Index         is (<>);
   type Element_Array is array (Index range <>) of Element;
   with function "<" (Left, Right : Element) return Boolean is <>;
procedure Concurrent_Mergesort (Sort_Field : in out Element_Array);
```

which goes with the implementation:

```
with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO;    use Ada.Text_IO;
with CPU_Counter;    use CPU_Counter;
procedure Concurrent_Mergesort (Sort_Field : in out Element_Array) is
   procedure Mergesort (F : in out Element_Array) is
   begin
      if F'Length > 1 then
         declare
            Middle : constant Index := Index'Val (Index'Pos(F'First)+ F'Length/2);
            subtype Low_Range  is Index range F'First .. Index'Pred (Middle);
            subtype High_Range is Index range Middle  .. F'Last;
            F_Low  : aliased Element_Array := F (Low_Range);
            F_High : aliased Element_Array := F (High_Range);
            Gained_Agent : Boolean := False;
         begin
            if CPUs_Potentially_Available then
               CPU_Count.Try_Check_One_Out (Gained_Agent);
            end if;
            if Gained_Agent then
               null; --> Replace this with concurrent operations
            else
               Mergesort (F_Low);
               Mergesort (F_High);
            end if;
            declare
               Low          : Low_Range  := Low_Range'First;
               High         : High_Range := High_Range'First;
               Low_Element  : Element    := F_Low  (Low);
               High_Element : Element    := F_High (High);
            begin
               Merge : for i in F'Range loop
                  if Low_Element < High_Element then
                     F (i) := Low_Element;
                     if Low = F_Low'Last then
                        F (Index'Succ (i) .. F'Last) := F_High(High..F_High'Last);
                        exit Merge;
                     else
                        Low  := Index'Succ (Low); Low_Element  := F_Low (Low);
                     end if;
                  else
                     F (i) := High_Element;
                     if High = F_High'Last then
                        F (Index'Succ (i) .. F'Last) := F_Low (Low .. F_Low'Last);
                        exit Merge;
                     else
                        High := Index'Succ (High); High_Element := F_High (High);
                     end if;
                  end if;
               end loop Merge;
            end;
         end;
      end if;
   end Mergesort;

begin
   Mergesort (Sort_Field);
end Concurrent_Mergesort;
```

This is blistering fast … it also does not sort anything. The essential concurrency section has been left out, as you happen to be enrolled in a course about concurrency. Before you go conceptually overboard: There are only a handful of lines missing here! If you find yourself writing another page of code, you are certainly on the wrong track.

You can safely ignore the rest of the algorithm (I promise it works fine) as your only job is to find a concurrent alternative to the sequential, recursive call sequence as you know it from the sequential version of Mergesort:

```
else
   Mergesort (F_Low);
   Mergesort (F_High);
end if;
```

Now step back and consider what exactly you need to express here. How can you express this as simply as possible?

Remember what the previous exercise demonstrated and then consider where do you need to wait for what before you can proceed with the rest of the algorithm.

Once you re-connected the logic, run the provided test framework and see what sorting times you achieve. For sanity check, go into the package `CPU_Counter` and manually overwrite the number of `Available_CPUs` with `1`. The performance should change proportional to the number of cores which live in your system … like in "should slow down proportionally". If it doesn't (or speeds up) then you need to go back to the drawing board.

If you want to see what your computer can do then change the scenario variable `Specific_Build_Modes` to `Performance` and let the bits fly. While this is not a particularly optimised piece of code, you will probably have a hard time to outperform this speed with any other language – feel welcome to go for an assembler version to attempt to outperform what you measured here, if you are so inclined.

After all the fun, submit the archive `Concurrent_Mergesort.zip` of the working version of your program to the *SubmissionApp* under "Lab 5 Concurrent Mergesort" for a detailed code review by your fellow students (and us). You will receive the submission of another student right after your submission (via the *SubmissionApp*) and we would like you to read this alternative solution and provide feedback. Make concrete suggestions for code improvement or alternatives if you see sections which can be done differently. Basically: provide the quality feedback which you expect to see for your own solution.

**Make Sure You Logout
to Terminate Your Session!**

Outlook

Next week you will make your tasks communicate directly.