IL2206 EMBEDDED SYSTEMS

Laboratory 1A : Concurrent Software Development in Ada

Version 1.0.0 (September 4, 2025)[1]

1 Objectives

The programming language Ada has been developed for embedded and real-time systems. Concurrency is supported directly by the language. Ada offers powerful communication mechanisms like rendezvous and the concept of protected object. Support for real-time systems is provided by the real-time annex. The objective of the laboratory is to introduce the students to Ada and its features for concurrency. For more information on Ada consult the KTH library, which has many books on the Ada programming language.

2 Preparation

It is very important that students are well-prepared for the labs, since both lab rooms and assistants are expensive and limited resources, which shall be used efficiently. The laboratory will be conducted by groups of two students. However, each student has to understand and explain the developed source code and the laboratory tasks to pass the laboratory.

⚠ Documentation

All program code shall be well-structured and well-documented. The language used for documentation is English.

2.1 Installation of Ada

Please check the installation instructions for Ada on the Canvas page.

[1]no changes from preliminary version 0.1.2

2.2 Code Skeletons

KTH provides code skeletons for the laboratory, which can be downloaded from the course page in Canvas.

2.3 Modular Types and Attributes in Ada

Ada provides a *modular* integer data type, which can be used to implement a circular buffer. This data type is used in the code skeletons for Section 3.2 provided for the laboratory. In case an addition or other operation yields a result that is larger or smaller than the highest or smallest value that the data type can represent, then "wrap-around semantics" are used. The following examples illustrates the usage of this data type, and also shows how *attributes* like `First` and `Last` can be used to access certain properties of the defined data type. Attributes play an important role in Ada and can be used in different circumstances.

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

procedure Modular_Types is

   type Counter_Value is mod 10;
   package Counter_Value_IO is
      new Ada.Text_IO.Modular_IO (Counter_Value);

   Count : Counter_Value := 5;

begin
   Put("First value of type Counter_Value: ");
   Counter_Value_IO.Put(Counter_Value'First,1);
   Put_Line(" ");
   Put("Last value of type Counter_Value: ");
   Counter_Value_IO.Put(Counter_Value'Last,1);
   Put_Line(" ");
   for I in 1..5 loop
      Count := Count + 6;
      Counter_Value_IO.Put(Count);
      Put_Line("");
   end loop;
end Modular_Types;
```

Execution of the code gives the following output.

```
ada> ./modular_types
First value of type Counter_Value: 0
Last value of type Counter_Value: 9
 1
 7
 3
```

```
9
5
```

2.4 Executing Ada Programs

Ada programs will in general utilise more than one core on your desktop computer, which is the setup for this laboratory. However, when using the real-time annex or if you want to define the number of cores that shall be used, special care has to be taken when executing the program as stated in the following box.

⚠ Controlling the Execution of Ada Programs

- If programs shall run on a single core, it has to be enforced by the user. In Linux the user can use the command `sudo taskset -c 0 ./program_name` to enforce execution of a single core.
- If you use the real-time annex in Ada, the programs need to be run in *supervisor mode* for the correct scheduling and timing!

NOTE! Since this lab does not use the real-time annex, can run on several cores, and does not require a well-defined scheduling algorithm, it can be executed in normal user mode instead of supervisor mode.

3 Laboratory Tasks

3.1 Semaphore

The Ada language does not directly provide library functions for a semaphore. However, semaphores can be implemented by means of a protected object. Skeletons for the package are available on the course page. Use the package specification `Semaphore` in the file `semaphores.ads` and modify the corresponding package body in the file `semaphores.adb` so that the package implements a counting semaphore.

◯ 3.1 completed

3.2 Producer-Consumer Problem

The producer-consumer problem is a very relevant problem in the design of embedded systems. The problem is illustrated in the Figure 1 and can be formulated as follows. There are m *producer* and n *consumer* processes, which are connected to a single *buffered communication channel*. Producers write data to the communication channel, consumers read data from the communication channel. For a reliable communication, the following synchronisation properties have to be fulfilled:

- A consumer cannot read a data element from an empty buffer

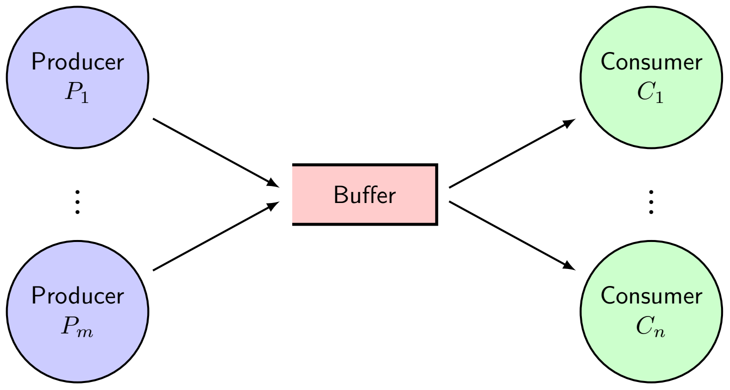

Figure 1: Producer-Consumer Problem

- A producer cannot write a data element into a full buffer

In the above formulation of the producer-consumer problem, consumers can read data from any sender and are not concerned from which sender the data comes from. When data is read, it is also removed from the buffered channel.

(a) Develop a solution for the producer-consumer problem by means of a *protected object* that uses a buffer of a fixed size. Use the package `Buffer` from the package specification file `buffer.ads` and package body file `buffer.adb`, which together with an initial code skeleton `producerconsumer_prot.adb` for the main program implementation is available on the course web page. Use the protected object to implement the producer consumer problem and save the final implementation in the file `producerconsumer_prot.adb`.

Modular Types in Ada

The buffer implementation uses modular types (Section 2.3). These enable define data types, which allow a 'wrap-around' if the maximum value (or minimum value) is reached and the next value (or previous value) should be calculated. For instance, if a variable `V` is implemented as a modular integer type between 0 and 5, and the current value is 5, then adding 1 to the variable `V` would give the result 0.

Please study the executable example on modular types in Ada, which is available in the Canvas page for this laboratory, to understand how modular types can be used.

Note: The main procedure and its corresponding source code file need to have the same name. Thus the main procedure needs to have the name `ProducerConsumer_Prot`.

◯ 3.2-a completed

(b) Develop a solution for the producer-consumer problem using the *rendezvous* mechanism. Use the same buffer structure as in Task a, but implement the circular buffer as an own server task. Save your implementation as `producerconsumer_rndvzs.adb`. An initial skeleton for `producerconsumer_rndvzs.adb` can be found on the course web page.

◯ 3.2-b completed

(c) Develop a solution for the producer-consumer problem using your *semaphore* implementation from Task 3.1. Use the same buffer structure as in Task a, but implement the circular buffer as a shared variable. An initial skeleton for `producerconsumer_sem.adb` can be found on the course web page. In order to use the semaphore package it shall be installed in the same directory as `producerconsumer_sem.adb`. It can then be accessed by the following code.

```
with Semaphores;
use Semaphores;
```

Use *two semaphores* `NotFull` and `NotEmpty`, which shall be used to block a) tasks that want to write to a full buffer, and b) tasks that want to read from an empty buffer, and *another semaphore* `AtomicAccess` to ensure mutual exclusive access to the buffer data structure. Your code shall use the code for the buffer provided in the skeleton `producerconsumer_sem.adb` and shall not be based on a protected object or rendezvous.

Draw also the diagram that illustrates your solution extending the figure above.

◯ 3.2-c completed

4 Examination

Demonstrate the programs that you have developed for the laboratory staff during your laboratory session. Be prepared to explain your program in detail.

Each student in a student group shall be able to explain all parts of the laboratory and the written code. In order to pass the laboratory the student must have completed all tasks of Section 3 and have successfully demonstrated them for the laboratory staff.