Lab 1: Ada programming

11/14/2023

Attempt 1

In Progress
NEXT UP: Submit assignment

Add comment

Unlimited Attempts Allowed
12/22/2023

Details

Assignment Goals

The goal of this assignment is to gain basic understanding of the Ada programming language and some of its constructs. Some problems are related to the real-time aspects of programming in Ada, other problems should make the student familiar with the Ada programming style.

All problems address generic aspects of real-time programming.

Hand In

Your submission must consist of the following:

- A report with answers to questions asked in each part.
- Programs solving the individual assignments (well commented, well indented).
- Any additional pointers we need to compile and run your code.

This lab is graded as Pass or Fail.

Language and Compiler

This assignment shall be solved in Ada95 and compiled using the GNU Ada compiler, `gnat`. This compiler is already installed in all of the department's Linux computer rooms, and also in the Windows computer rooms. You can also download the free **GNAT Community Edition (https://www.adacore.com/download)** to your own machine, or use this **Ubuntu Linux virtual machine (https://drive.google.com/file/d/1pa-DG6nPbL6qjQkL1ENdxiUrlgJwOtC1/view?usp=sharing)** (6.1 GB, don't try to download it to the department's machines) with all the software needed for all the labs (for installation instructions for the VM, refer to the slides of the introduction session to this lab).

Submit assignment

We also provide you with a **Quick Ada Tutorial (https://uppsala.instructure.com/courses/80399/pages/quick-ada-tutorial)** . *Please note that the exercises there are just for your own training, and not part of this lab assignment!*

Please download and use the code templates we provide for the different parts of this lab from **here (https://uppsala.instructure.com/courses/80399/files/5769858?wrap=1)** **(https://uppsala.instructure.com/courses/80399/files/5769858/download?download_frd=1)** .

Part 1: Cyclic scheduler

Write a cyclic scheduler which manages three procedures F1, F2, and F3. We demand the following for the executions:

1. F1 shall be executed every second
2. F2 starts when F1 terminates
3. F3 shall execute every other second, starting 0.5 seconds after F1's start

The execution times for the functions are not known. However, you can assume that F1 and F2 together execute for less than 0.5 seconds, and that F3 does not take more than 0.5 seconds to execute.

Let the functions print an informative message when they execute, e.g.

```
F1 executing, time is now: 0.00001
F2 executing, time is now: 0.00004
F3 executing, time is now: 0.50008
F1 executing, time is now: 1.00008
F2 executing, time is now: 1.00010
...
```

Make sure that the printed time has a resolution of at least 1/1000 sec.

Examine the code from the template, you can already compile it and see the output. From the output, you should see that the start times of F1, F2, and F3 have drifts accumulated with time going by, which is NOT allowed in your final output.

Try to modify it in places indicated in the template.

Note: Some amount of jitter in the start times of the functions cannot be avoided. However, the start times of the functions are not allowed to drift further and further away from the schedule!

Questions:

1. Explain the difference between relative delay, and absolute delay based on your own understanding.
2. Suppose we know the exact execution times of F1, F2, and F3. Is it possible to use relative

Submit assignment

Hints:

- Use the package Ada.Calendar for access to the clock
- Use the package Ada.Text_IO for printing messages
- There is no need for tasking in this part of the lab

Part 2: Cyclic scheduler with watch-dogs

Modify F3 from part 1 so that it occasionally takes more than 0.5 seconds to execute.

Augment the cyclic scheduler from part 1 with a watchdog task to monitor F3's execution time. When F3 exceeds its deadline (0.5s), the watchdog task should immediately print a warning message. I.e., 0.5s after start of F3, either F3 has finished or the watchdog has printed a message. The watchdog should let F3 finish, even if it misses its deadline.

The watchdog task should be started at the same time as (or just before) F3 starts executing, and from that point on measure the time that F3 uses.

When F3 misses its deadline, the cyclic executive should re-synchronize so that F1 is started at whole seconds.

There is a skeleton code file provided for this part, so you shouldn't reuse the one for the first part of the lab. Examine the code and understand how Ada defines a task. Try to modify it in places indicated by the comments. You can also reuse part of your solution of part 1.

Questions:

1. How did you synchronize the watchdog task with the start and the end of F3's execution?
2. Explain how your watchdog task monitors the execution time of F3.
3. Explain the way you re-synchronize the cyclic executive when F3 misses its deadline.

Hints:

- The *select*, *accept*, and *delay* statements of Ada might be useful here.
- The packages *Ada.Numerics.Discrete_Random* and *Ada.Numerics.Float_Random* are useful for creating random numbers

Part 3: Process Communication

Create three tasks:

1. A task that acts as a first in, first out (FIFO) buffer for integers. The buffer should block any task which makes calls that could overflow or underflow the buffer. The buffer should at least be able to buffer 10 integers

Submit assignment

3. A task that pulls integers out of the buffer at irregular intervals (i.e. a *consumer* of values), and summarizes them. When the sum is above 100 the task should terminate the program. The information that the buffer and the producer should terminate should be spread using synchronization (not using global variables, the producer should not count, ...). You are not allowed to use *abort* or *terminate* commands.

In your solution, the buffer should not print any messages, but the producer and consumer should print messages containing the integers that are given to/taken from the buffer.

Questions

1. Show by a concrete scenario that the producer-consumer-buffer program using blocking rendezvous communication can have deadlocks and explain the mistake that can cause it.

Note

- You should not implement semaphores nor use the Semaphore_Package. Synchronization and mutual exclusion should be implemented by the task which manages the FIFO buffer. The producer and consumer shouldn't communicate directly, except for the termination synchronization.
- You should not throw away any number produced by the producer, even if the buffer is full at the moment. But you can throw away numbers, if any, produced by the producer after the buffer task receives termination signal from the consumer.

Hints

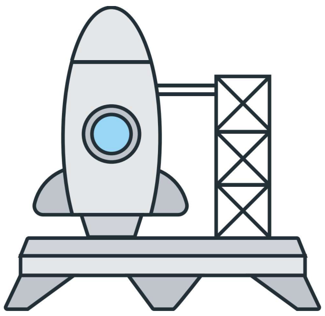
- To detect errors, make sure your test run includes situations where both the producer and the consumer get blocked on the buffer. (Not at the same time, but each one eventually.)
- The statements *accept* and *when* might be useful for the buffer task.

Part 4: Data driven synchronization

Re-implement the FIFO buffer in the previous part as a protected shared object.

Questions

1. Does the producer-consumer-buffer program using protected object suffer from any potential deadlock? Explain why or why not.



Choose a file to upload

or

Webcam Photo

Canvas Files

☐ Härmed försäkrar jag att detta är mitt eget arbete. Jag förstår att dokumentet kommer att plagiatkontrolleras med Ouriginal. I hereby declare that this is my own work. I understand that the document will be checked for plagiarism with Ouriginal.

Submit assignment