Practical assignment on Concurrent Programming 2025/2026

José Proença

Department of Computer Sciences, Faculty of Sciences, University of Porto

Groups

This is a group assignment. Groups should have at most 2 members – if you fail to find a partner please let us know. You can join a group using a shared spreadsheet: `https://docs.google.com/spreadsheets/d/1dwH-LwGZIRH9VDvvbYXB75-E5nUp4bvxM7FM8MewgRI/edit?usp=sharing`

What to submit

A single ZIP file with the *PDF report* that answers the proposed questions **and** the *source code* of the developed *implementations* in CAAL/PseuCo and in Scala. Use different folders with the implementations of different questions, include a `readme.txt` explaining how to compile and run each implementation, and **do not include compiled code** (e.g., the `target` folders for Scala).

AI support

Include in your report a small section explained if any AI was used, and how it was used.

Presentation

Present your encodings and implementations in a 8 minute presentation, to be scheduled for **June 2** - **June 3**, 2026.

Deadline

23h59m of **May 31**, 2026 (Sunday)

Sending money using locks

Exercise 1. [5 pt] Recall your first deadlock example tackled when using `synchronised` blocks, where money is transferred between different accounts. Figure 1 represents a simplified diagram where three participants use a send€ action to represent an atomic transfer of money. This scenario does not include locks, which can lead to deadlocks when improperly implemented.

1.1. Model in CCS (using Pseuco or CCS-CAOS) a simplified version with two participants always trying to send money to each other. Ignore the amount of money. **Use locks in a naive way**, i.e., such that the deadlocks seen in the bank transfer example from our lessons can occur. You should include:

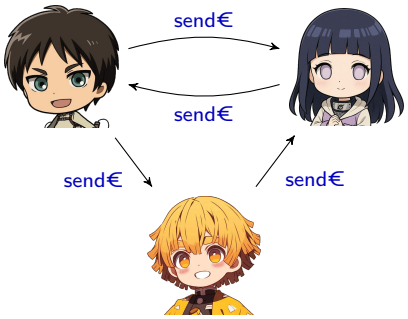

Figure 1: Example scenario where different participants send money to each other

- the CCS code in a separate file in the zip submission;
- a screenshot of the resulting LTS;
- an explanation of which **parallel processes** and which **actions** you used in your model.

1.2. Adapt your CCS model from the previous exercise to **fix the deadlock**, using a similar approach to the solution presented in lessons. Submit the CCS code of this variation as a separate file, and explain (and prove) if the two systems are weakly bisimilar.

1.3. Model in CCS the scenario from Figure 1 **using locks in a naive way** (where the deadlocks can occur). For example, in this example the bottom participant (Zenitsu) can only send to the right participant (Hinata), but the right participant (Eren) can send to both other participants.
Include the code for Pseuco or CCS-CAOS in your zip file.

1.4. Implement a generalisation of the scenario above using **value passing CCS**, with three equivalent participants that can always send messages to any other participant. Include the full CCS code for Pseuco in your zip file.

Simulate dependent processes in a server

Exercise 2. [8 pt] It is not possible to execute an OS process in a browser via JavaScript. A possible workaround is to implement a server that receives requests from a browser, with the instructions to be executed, and having the browser delegating these instructions to the server.

In this exercise we will only simulate such executions on a server. You will start with both a complete JavaScript client and with a working *skeleton* of a server in Scala. The client sends requests to a running server, which provides responses. Your goal is to implement the rest of this server. All provided files can be found in `https://fm-dcc.github.io/cp2526/exercises/process-simulator.zip`, which include:

- `client/index.html` – where you can open a website running JavaScript that can request tasks to be simulated in a given server;
- `server/src` – where you can find the source-code to a basic server with placeholders that you need to fill;
- `server/build.sbt` – where you have a configuration that you can use to run the server.

Running and using the server. In a Linux system, open a terminal in the `server` folder and run it using the command `sbt run`. Then using an internet browser (e.g., Chrome) open the website "`client/index.html`".

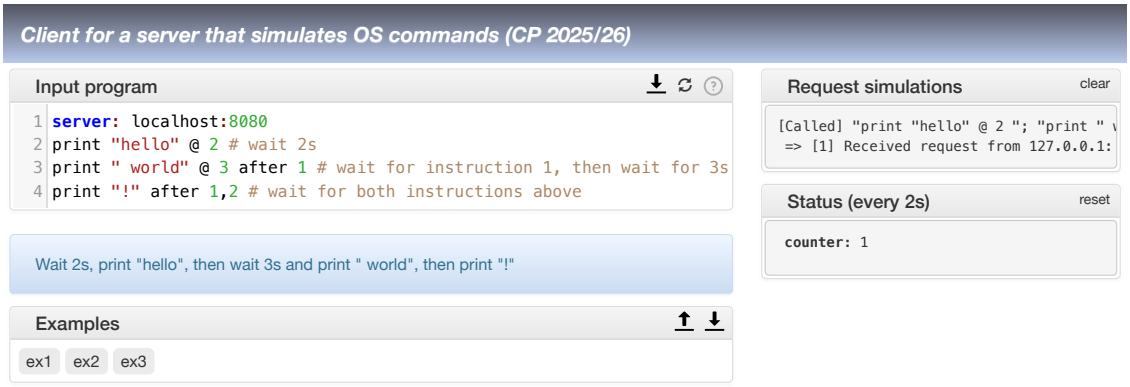

Figure 2: Screenshot of the client's view in a web browser

A screenshot of the website from `client/index.html` is in Fig. 2, using example `ex2`. You can provide a list of instructions in the left window, called "Input program". Click the header "Request simulations" to expand that window and to send requests to the server mentioned in the input program. Each block under a `server` will trigger a single request with the list of instructions, and multiple such blocks can exist. The confirmation from the server that the request was received will be appended to that window.

After expanding the widget "Requested simulations" the first time, every time you press the refresh button next to the "Input program" header you will resend the list of requests, and append the replies to the window. You can press the "clear" button to delete the content of this window.

If you click on the "Status (every 2s)" header, you will trigger this window to expand. While expanded, the client will send a request for the state of the first server every 2 seconds, displaying the reply once it is received.

What is an instruction? Each of these instructions has three parts by a fixed order:

- a `print` instruction (mandatory);
- a `@` delay, representing how long the print should take since it the task starts executing, simulating the time to process of a more complex task (optional, default is `0`); and
- an `after` dependency, consisting of a comma-separated list of indices of all instructions that must complete before the instruction can start to execute (optional, default is the empty list).

See the examples in the client to better understand how to write an instruction.

What to implement. You will have to support the execution of our simple instructions in a single server assuming many requests can be made at the same time (possibly by different clients). The requirements of this server are listed below.

1. You should implement a thread pool with a **fixed** number of threads to manage the pending instructions.
2. The code that is executed (concurrently) when receiving a request should be as small as possible.

3. If an instruction is ready to execute and there is an unused thread, the instruction should be executed as soon as possible.
4. Shared data structures should be used to model the state of the server.
5. Three different kinds of requests can be made to the server:
   - **run** a given block of instructions,
   - **ask for the state** of the server, and
   - **reset** the state of the server.
6. The result of the printing instruction should be stored by the server in its internal state (sent to the client when requested).

You should include in your zip file all your source code. You can use different folders for different variations of the server if you want. The main files that you need to modify are:

- `server/src/main/scala/cp/serverSim/Routes.scala` (to specify how to react to requests) and
- `server/src/main/scala/cp/serverSim/ServerState.scala` (to specify the internal state of the server).

2.1. For now ignore the content of the instruction. Observe that the **original counter** should count the number of requests received, but it is not thread-safe (it may provide unexpected results).

Provide an example of an "input program" that is likely to fail with the provided server, and explain why it fails. Then propose an efficient solution for this counter that will not fail with your example program.

2.2. For now ignore the "`after`" clause. Implement the thread pool to run the instructions, and use a thread-safe shared structure **using lock-free programming** to store the result of the print instructions with a time stamp. This structure should be displayed in the "Status" widget of the client.

Provide an example of an "input program" that would be likely to fail with a non-thread safe implementation of the shared structure, explain why it would fail, and present the result with your correct implementation.

2.3. Considering the full instructions (including the "`after`" clause), provide three examples that illustrate that the dependency mechanism is working properly in a concurrent scenario.

2.4. Extend your implementation with some functionality that can be efficiently implemented using a `@volatile` variable. This functionality does not need to be useful – only didactic. Explain why `@volatile` is needed, and provide an example of an "input program" that can fail without this annotation, but always succeeds with the annotation.

Selling tickets with actors

Exercise 3. [4 pt] You will implement an actor that manages tickets to be sold. It can receive tickets to sell, or it can sell the tickets (if it has tickets in stock to be sold). You can find below a simple skeleton of this system.

3.1. Complete the template below such that a ticket office can be told to sell (`ToSell`) a given number of tickets more, can be purchased (`Buy`) a given number of tickets. When the number of requested tickets is less than the number of available tickets the purchase should fail, and an error message should be logged. Submit your implementation with code to test your application.

```scala
object TicketOfficeTest extends App {

  case class ToSell(n:Int)
  case class Buy(n:Int)
  //...

  class SellerActor extends Actor {
    val log =
        Logging(context.system, this)
    //...
    def receive: Actor.Receive = {
      // case ...
    }
  }
```

```scala
  // Testing the system

  val sys = akka.actor.ActorSystem("TicketSys")
  val ticketOffice =
    sys.actorOf(Props[SellerActor],"main office")

  ticketOffice ! ToSell(2000)
  for (x <- 0 until 101) do
    ticketOffice ! Buy(20)
  log("Tried to buy many ${20*101} tickets.")
  ticketOffice ! "Bye"
  Thread.sleep(3000)
  sys.terminate()
}
```

3.2. Modify the ticket office implementation with a delegation functionality: If the number of available tickets is higher than a threshold (say 100), the ticket office should create a child actor and send a fix number of tickets to sell (less than the threshold, say 50). These child actors should report to their parent once they fail to have enough tickets for a given purchase, returning the possible remaining tickets.

Include in your code a way to test this functionality, and include in your report: (1) a diagram with the actor hierarchy, and (2) a sequence diagram with a possible exchange of messages produced by testing code.

Presentation

Exercise 4. [3 pt] Give a short 8-minute presentation summarising your CCS encodings and your Scala implementations. You do not need to submit slides and all members of the group should talk when giving the presentation. The presentation will be scheduled for June 2 - June 3, 2026. If for some reason you do not expect to be available (e.g., you are abroad), please let us know as soon as possible.