Laboratory Assignment 6: Synchronization and communication of threads from different processes

Table of contents

1 Objectives

This practice aims to reinforce our knowledge of the POSIX system's thread and process synchronization mechanisms and usage schemes. In practice, we will make use of: mutex, condition variables, semaphores, and POSIX shared memory objects.

The archive p6files.tar.gz contains several files that can be used as a starting point for developing this practice, as well as some makefiles for the compilation of the different projects.

2 Exercise

This exercise consists of solving the problem of the tribe of savages on the problem sheet, using different processes to simulate each of the savages and the cook. The classical problem statement is as follows:

Savages from a tribe take food from a cauldron with M servings of missionary stew. When a savage wants to eat, a portion from the cauldron is served unless it is empty. If it is empty, he must notify the cook to replenish another M ration, and then his ration may be served. A cook and an arbitrary number of savages behave as follows.

```
//Cook:
while (!finish) {
  putServingsInPot()
}

//Savages:
for (i = 0; i < NUMITER; i++){
  getServingFromPot()
  eat()
}
```

The following restrictions must be met:

- Savages cannot invoke getServingFromPot() if the cauldron is empty.
- The cook can only invoke putServingsInPot() if the cauldron is empty.

To simulate this problem, we will create two programs:

- A *cook.c* program that creates the necessary shared resources and then executes the cook() function. The user only needs to create a process that executes this program. This program will register a handler for the SIGTERM and SIGINT signals. When the signals are captured, the program should terminate, clearing all the shared resources created in the system.
- A program *savage.c* tries to open the shared resources created by the *cook.c* program. If it does not find them, it will warn the user with an error message asking to run the *cook.c* program first. Once the shared resources are opened, it will execute the savages() function, which will try to eat NUMITER times from the cauldron and then terminate. The user can create as many processes running this program as he wants, in different terminals or the same terminal, by launching the processes on the *background* (for example, you can use a bash for loop and launch several processes running this program).

Students are advised to use named semaphores for synchronizing the different processes and a shared memory region to hold the variable representing the cauldron's contents.

The *putServingsInPot()*, *getServingsFromPot()*, and *eat()* functions, in addition to implementing the necessary synchronization, must display messages on the terminal to check the operation of the program, indicating in each message the id of the corresponding process.