ARP Assigment 2019/2020 V2.0

Advanced Robot Programming - Università Degli Studi di Genova

Steven Palma Morera
S4882385
imstevenpm.study@gmail.com

***Abstract*—The aim of the present report is to summarize the procedure followed to design the network simulation of multiple Posix processes using basic system calls in Linux described as in [1]. The results obtained from the repository developed are briefly discussed in order to confirm that the objectives solicited in [2] were successfully achieved. The reader can also find the instructions for use for the repository. [3].**

***Index Terms*—Posix, Linux, basic system calls**

I. Introduction and Requirements

A. Introduction

The network of Posix processes desired is shown in the Fig.1, where each circle corresponds to a Posix process and each arrow represents a communication between them using either pipes or sockets (0).

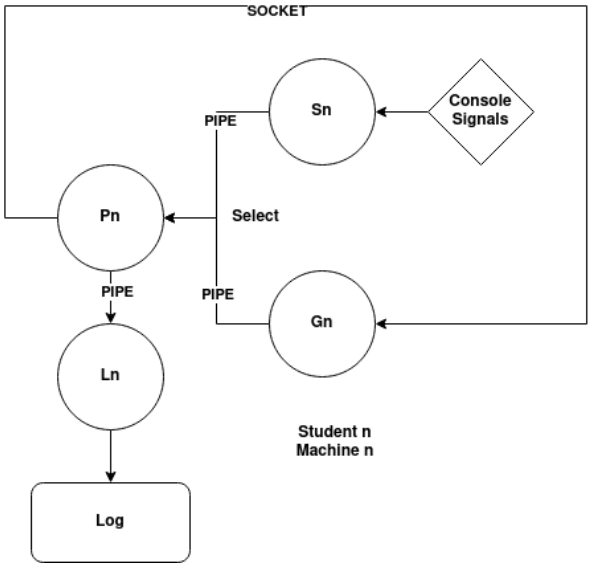

Fig. 1: Network of Posix processes desired. Source: [1]

B. Specific Requirements

- Process Pn: (1) Receives messages from Gn and Sn, computes and sends a new message to Gn after a time delay chosen by the user or sends log messages for Ln to save them. (2) Pn should use a "select" system call to decide whether it listens to messages coming from Sn or Gn via pipes.
- Process Ln: (1) Logs data end events in a log file. (2) It communicates only with Pn using a pipe. (3) It logs an entry each time a communication between processes happens.
- Process Gn: (1) Receives messages from Pn via a socket and dispatches them to Pn via a pipe. (2) It should be initialized by the system called "execv".
- Process Sn: (1) Receives console messages as Posix signals. (2) The console messages should perform the following actions: Start (Pn starts receiving messages), Stop (Pn starts receiving messages), Log (Ln outputs the contents of the Log file)
- Config file: (1) A configuration file where the user can specify important parameters of the network before running it, like port number, IP address, reference value and a time delay.
- Messages: (1) The messages communicated between the processes should have a token value computed in each iteration and a timestamp of when Pn sends the token to Gn.
- Log file: (1) Holds a series of text lines in couples of the form: timestamp,from G/S/P,value of message.
- The network simulation should be designed to run in different and multiple machines.

II. Development and structure

In order to accomplish the requirements specified in the past section, the repository from [3] was developed containing the following files, a further study of the SnPnLn_StevenPalma.c and Gn_StevenPalma.c scripts will follow next.

- Readme.md: Shows general information and meta data about the project.
- Bash_StevenPalma.bash: a script for compiling and executing the project.
- Config_StevenPalma.config: a config file the user uses to edit the important parameters of the network stated in the past section.
- SnPnLn_StevenPalma.c: a C language script that contains all the programmatic functionalities of the Posix processes Sn, Pn and Ln.
- Gn_StevenPalma.c: a C language script that contains all the programmatic functionalities of the Posix process Gn.

A. SnPnLn_StevenPalma.c: Structure and design decisions

This script is composed mainly by six functions, a main, a function task for each Posix process and two signal handlers.

1) Main: In the main function, the processes are created using the system call "fork" and a PID and a task function is assigned to Sn, Pn and Ln. Sn is the parent process of P and G, while P is the parent process of L. For some processes and argument is passed to their task function, like a PID or a file descriptor to their proper functioning. Finally, the Gn process is executed from Sn using the system call "execvp", for completeness of the requirements.

2) Sn: In the Sn Task function, the system call "signal" is used to declare the callbacks for the SIGUSR1 and SIGUSR2 console signals. The signal handler for the SIGUSR1 signal will be responsible of starting and stoping the communication of messages of Pn, this is done by toggling the value of bool variable and sending it through a nammed pipe to Pn using the system calls "open", "write" and "close". The signal handler for the SIGUSR2 will be responsible of printing the contents of the Log_StevenPalma.log file generated by Ln. First, it notifies Pn that a request for doing so has arrived by writing a variable to a nammed pipe. Then, it opens, gets, prints and closes the log file.

3) Ln: While Sn and Gn communicate with Pn using a named pipe, Ln communicates with Pn using a unnamed pipe instead. This decision was made only for educational purposes. Each time an interaction between processes happens, Pn communicates it to Ln via the unnamed pipe. For this, Ln is constantly hearing the unnamed pipe using the "select" system call for performance optimization. Once there is data in the unnamed pipe, Ln reads it with the system call "read", then it takes the absolute time from EPOCH with "gettimeofday" system call, and it finally opens the Log_StevenPalma.log file and "fprintf" the contents of the message that Pn sent in the right format specified in the past section.

4) Pn: Pn is the central process of the network so it does more work than the other.

- First, It initializes all the elements needed for the communication, like the named pipes using the "mkfifo" system call and the socket and then it reads the contents from the Config_StevenPalma.config file.
- Second, in a while loop and using a "select" system call, it constantly listens for new messages from the named pipes -that is, from Gn and Sn-. If a message was sent from the first named pipe (Sn), then it either changes the local value of the start/stop variable and registers the communication writing in the unnamed or it only registers that a request for printing the log file has been made.
- Third, if a message was sent from the second named pipe (Gn), that means that a new token has to be generated. For that, it first checks if the start/stop variable is on, if it is then, it reads from the named pipe using the "read" system call and saves the message received (a token and a timestamp) into a struct variable, it then sends the event to Ln via the unnamed pipe. Later, it initializes all the parameters for the socket communication with "socket", "inet_pton" and "connect" system calls. Once it is connected to the other end of the socket, it waits a time delay chosen by the user in the Config_StevenPalma.config file and then takes the absolute current time and computes the new token value. Finally, it sends the new message with the system call "send" through the socket, registers the communication to Ln through the unnamed pipe and closes all the open file descriptors.
- Since the formula provided in [1] and [2] doesn't seem to work properly, the designer decided to implement an entirely new formula for computing the token in the Pn process. The formula used is the on from Eq.1 and can be thought as a recursive uniform rectilinear motion of a body. Where the Token value at time i represents the distance traveled by a body in the given time interval and the Reference Value chosen by the user in the Config_StevenPalma.config file represents the constant velocity of the body.

$$\begin{aligned} x_i &= x_{i-1} + v\Delta t \\ Token_i &= Token_{i-1} + RV(Time_i - Time_{i-1}) \end{aligned} \tag{1}$$

This formula was chosen because it uses the same elements from the one given in [1] and [2] - previous token, difference between timestamps and reference value- but with a different, functional and easy to evaluate meaning.

B. Gn_StevenPalmapipe t.c: Structure and design decisions

This script is only composed by a main function. It first reads the port parameter from the Config_StevenPalma.config file, then it declares the initial variables for a socket communication with the "socket", "bind" and "listen" system calls. Gn will work as the server while Pn the client, this was done because since originally from [1] Pn was the one sending the message to Gn+1 so Gn only needed to listen the requests. It now initializes the first message to be send and waits 10 seconds before doing so. In a while loop, it sends the message through the named pipe using the system calls "open", "write" and "close" - this functionality was added because of [2] - , then it waits for a reply from the Pn socket end with the system call "accept". Once a socket connection is established, it reads the content of the message, stores it in the proper format and closes all the file descriptors used. Finally, at the beginning of the while loop it sends the message back again through the named pipe.

III. Instructions for use

The following instructions were tested using Ubuntu Focal Fossa 20.04.1 LTS on August 6th, 2020. For a more recent guide, check the Readme.md file from the repository online.

1) Open a new terminal in a chose directory and clone the repository given in [3] by running:

```
git clone https://github.com/
imstevenpm/ARP_Assigment.git
```

2) In the same shell, execute the bash script by running:

```
./Bash_StevenPalma.sh
```

A new terminal will show up and the SnPnLn and Gn executables will be created in the current directory.

3) Before continuing, the user can set the parameters from the Config_StevenPalma.config file to a different ones. In the new terminal, execute the SnPnLn executable by running:

```
./SnPnLn
```

The network should initialize by showing on shell the PID of each Posix process created. Also, a time delay of 10 seconds is waited before Gn sends the first message to Pn and the communication cycle starts.

4) After the 10 seconds, messages from all the Posix processes will appear in the shell indicating their current status. The network is up and running.
5) Notice that the Log_StevenPalma.log file is created as well as the FIFO files for the named pipes in the current directory of the user's machine.
6) From here, -in a different shell- the user can send the console signals to Sn for displaying the Log_StevenPalma.log content by running:

```
kill -SIGUSR2 <Sn PID>
```

or for starting/stopping Pn from receiving messages by running:

```
kill -SIGUSR1 <Sn PID>
```

7) To end the execution of the network the user can type CTRL+C

IV. Results and Analysis

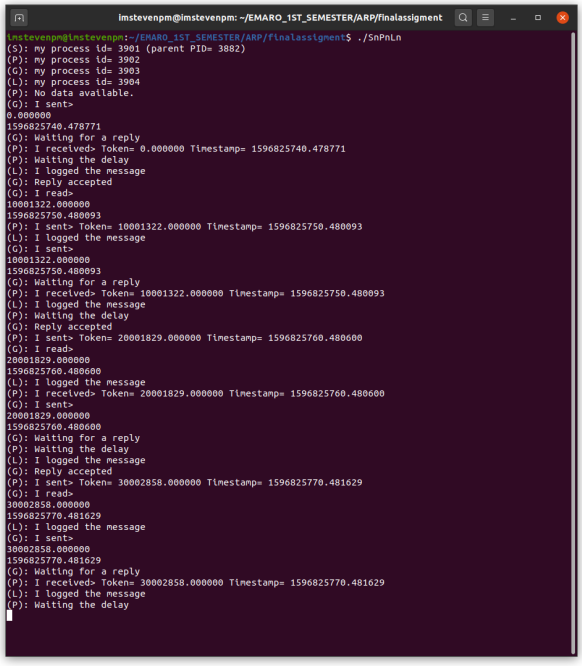

Fig. 2: Pn and Gn exchanging messages in the network. Source: Own elaboration

After following the steps and using the tools developed, the next results were obtained. In Fig.2 It is shown the usual output in terminal when the network is up and running. As it can be seen, first the PID of each process is displayed and after a few seconds the communication cycle starts. First, Gn sends a message through the named pipe, and waits to a reply through the socket. Second, Pn receives through the named pipe the message Gn has just sent, waits the time delay and then computes a new token and sends a new message to Gn through the socket. Third, Gn listens to that new messages and the communication cycle continues. Finally, Ln is constantly registering the events. In the Fig.2 the reader can follow these steps up to the fourth cycle of communication.

Also from the Fig.2 It can be noticed the reasoning of the tokens generated. Notice how the first token is equal to 0, this is because is the initial message sent by Gn; however, notice that in the subsequent tokens an increment of approximately 10000000 happens each time in each cycle. The extra increment -usually under 1000- is the actual delay caused by the communication between the processes, if only if the Reference Value is equal to 1. This is the expected outcome since the time delay used in the Config_StevenPalma.config file is 10000000 and the Reference Value is equal to 1, so each token generated in the communication cycle follows effectively the Eq.1. The reader can now experiment using different time delays and a different Reference Value.

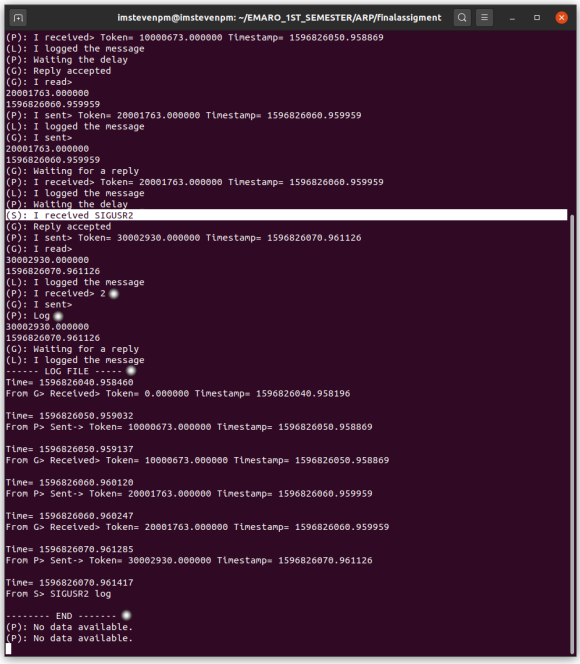

Fig. 3: Sn receiving the Console Signal for printing the Log file. Source: Own elaboration

In the Fig.3, Fig.4 and Fig.5 the capacity of the network to react to the console signals is evaluated. First, in the Fig.3, the log file was printed after a proper console signal for printing the log file was sent. It can be noticed when Sn received

the signal and once Pn finished the current task, the Log file generated is printed just as desired and expected.

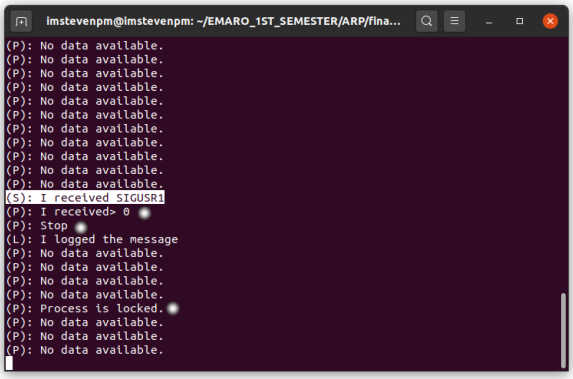

Fig. 4: Sn receiving the Console Signal for stopping the communication between Pn and Gn. Source: Own elaboration

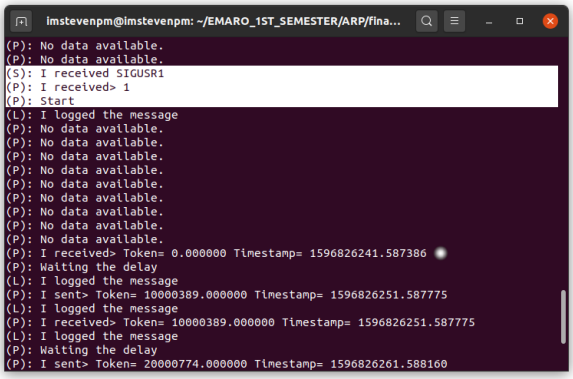

Fig. 5: Sn receiving the Console Signal for restarting the communication between Pn and Gn. Source: Own elaboration

Second, in the Fig.4, the communication cycle was stopped after the proper console signal for start/stop the communication was sent. It can be noticed when Sn received the signal and how Pn reacts accordingly rejecting any new messages request from the named pipe shared with Gn, just as desired and expected. Third, in the Fig.5, -similarly- the communication cycle was restarted after the proper console signal for star/stop the communication was sent. It can be noticed when Sn received the signal and how Pn reacts accordingly accepting again messages from Gn, just as desired and expected.

In the Fig.6, It can be seen the Log file generated by the processes, registering nicely all the events that happened in the network.

Finally, in the Table.I a summary of the accomplished requirements obtained from the network designed is shown. Notice how every requirement was successfully developed; however, those in yellow have important observations that shall be discussed in the next section.

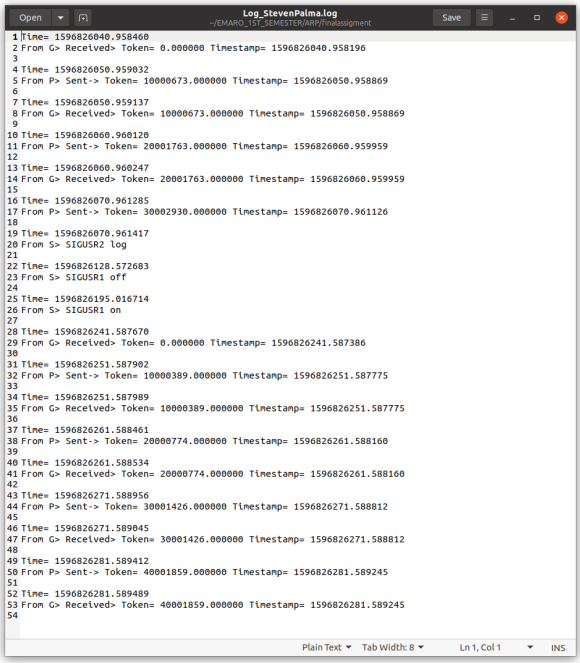

Fig. 6: Log file created. Source: Own elaboration

Requirement	Working	Fully working
Network (0)	-	✓
Pn (1)	✓	-
Pn (2)	-	✓
Ln (1)	-	✓
Ln (2)	-	✓
Ln (3)	-	✓
Gn (1)	✓	-
Gn (2)	-	✓
Sn (1)	-	✓
Sn (2)	✓	-
Config file (1)	✓	-
Messages (1)	-	✓
Log file (2)	-	✓

TABLE I: Requirements check table obtained for the network developed. Source: Own Elaboration

V. Additional Observations

- For Pn (1), the requirement is marked as yellow because the application of the network defined in [1] and [2] of computing the token as a sinusoidal wave wasn't accomplished because the formula didn't work properly. However; using the new formula proposed by the designer, the system works fine.
- For Gn (1), the network was designed originally for the requirements of [1] so It isn't well optimized for [2]. This results in Gn sometimes not communicating properly with Pn, however a simply restart of the network solves it.
- For Sn (2), the signals can be sent and the processes will do what required; however, after a console signal was sent, the communication cycle stops since Pn listens first in the select to messages coming from Sn. This causes that, after a console message is sent, Pn doesn't receive

anymore messages from Gn since Gn is still waiting for the reply from the socket that Pn should have sent when Gn sent the message, but Pn won't do it because it listened to the named pipe from Sn and not the one from Gn. Since Gn is waiting the reply, It won't send any message and Pn won't receive any neither. This didn't happen when the network was used for the application described in [1] since Gn and Gn-1 were different and independent. In order to solve this, each time a console signal is sent and the communication cycle is wanted to be restarted again, then the user first must kill the current Gn process running:

```
kill -9 <Gn PID>
```

And then execute a new Gn process by running:

```
./Gn
```

- For Config file (1), the parameters are well read and used in the network; however, the IP address wasn't used since there wasn't any other additional machine to be tested on. This means that even if the network should work between different machines, It wasn't tested. Also, the designer noticed that the network is sensitive to the time delay for the Pn process specified in the Config file. This value should be well tuned accordingly to the timeouts of the selects used and the initial delays of the network. This didn't happen when the network was tested for [1], since this time delay was an add-on from [2].
- Regarding the files created after the bash file and the processes are executed, it is advisable for the user to delete all the files generated each time a new network is about to be started (Log file, executables and pipes). since for example, the Log file is never emptied, so It might have values from past experiments. This was done like this because no requirement was given about what to do with the log file when the network finishes.
- Regarding the messages displayed in the shell when the network is running, sometimes the messages get overlapped. This happens because each process is executing simultaneously so It is expected that their print functions to get overlapped in the shell. However; this is something that only happens in the shell and It doesn't happen in the Log file where the event are registered without overlapping.

VI. Conclusions

- A network simulation of multiple Posix processes shown in the Fig.1 was successfully developed.
- Basic systems calls and Posix concepts in Linux were used to accomplished Its programmatic design.
- A full repository for the project was created ([3]) for further updates and optimizations. Hence, It is advisable for the reader to check it constantly.

References

[1] Zaccaria, R. 2019 ARP Assignment V1.0, 2019. Retrieved from: https://2019.aulaweb.unige.it/pluginfile.php/147205/mod_resource/content/2/assign_specifications.pdf

[2] Zaccaria, R. ARP Assignment academic year 2019-2020 (V2.0), 2020.

[3] Palma, S. ARP_Assigment, 2020. Retrieved from: https://github.com/imstevenpm/ARP_Assigment