ARP Assignment 2019-2020

Chiara Terrile
ID: 4337786
chiaraterrile97@gmail.com

Abstract—**This report briefly summarizes the design of a multi-process communication together with the result obtained during the experiments. This is based on Linux system calls implemented via Posix language.**
In addiction to the explanation of the code present in the repository, the reader can find in this report the instructions for the usage of the aforementioned repository.

I. INTRODUCTION

A. Architecture

For the multi-process communication we have the following architecture (see Fig. 1). The main processes are four : P, S, G and L, all connected via socket or pipes.
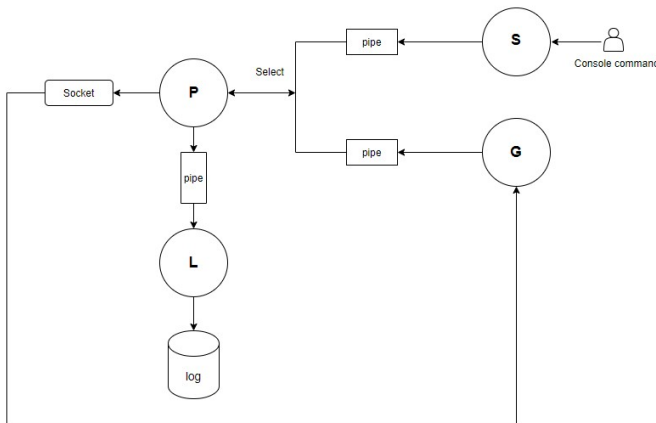Below the reader can find the description of the working principle of these processes.

Fig. 1. Architecture of the processes communication

B. Processes features

- **Process P** : this process receives messages from S or G process and if both of them are sending messages, via *select()* system call, P decides which of them to listen. Once read a message, P writes on a pipe the content and sends it to L process.
  If the message is coming from G, process P replies with a new message that is sent through a socket after a certain amount of time (defined by the user in the configuration file).
- **Process G** : this process reads the message coming from process P (the one sent via socket), and replies with the same content through a pipe between G and P.

- **Process S** : this process receives Posix signals from the user, the signals received can trigger three different behaviours : stop, resume or dump log (each of them will be better explained later on).
  Once received one of these signals, S sends a message to P process through a pipe with the information about the Posix signal.
- **Process L** : this process reads from the pipe where P has written, and log the content in a logfile depending on what it receives.

II. IMPLEMENTATION

In the folder given to the user, there are different files, described below :

- *config.txt* : this file contains the configuration values that can be set by the user. An example of these values can be the following :

```
localhost
8080
1000
1
```

  Where the first one is the *IP address* of the machine, followed by a *port number* (for the socket communication), a *waiting time* (that is the time waited by P before replying to G with a new message) and a *reference frequency*.
- *G_process.c* which is a C script that manages the task of G process (i.e. reading from the socket and sending the message through the pipe).
- *main.c* which is a C script that manages the other processes and the execution of the previous script.

In the next subsection the two C scripts aforementioned will be analyzed more in detail.

A. G_process.c

In this script there are two main parts : the server side of the socket communication and the task of writing on the pipe between G and P.
For the socket part, the port number for reading is passed as argument whenever this code is executed (by the main.c script).
The type of message exchanged in both parts is of type *token*:

```
typedef struct {
   double value;
   double timestamp;
}token;
```

Where we have the value of the token, that is a double number, and a timestamp, that corresponds to the time instant in which the message is sent or received (from the G side, is the time instant in which the message is sent again to P via pipe).
Once read the message from the socket, the value of the token is stored, a new timestamp is computed and then a message of type token is sent again to P.

B. main.c

In this script are managed all the processes together with the task of reading from the *config.txt* file and the logging action.
To explain how this is achieved, the main functions and parts of the script will be surveyed.

- *WriteLog* : this function takes in input the message received by L from P, and a flag that indicates if the message is coming from G process or from S, and then write a *LogFile.log* that will be created everytime the main file is launched.
  In case of a message coming from G, in the log will be written something like this :

```
timestamp : 1632393216.289048 from G
    process , received message :0.533.
timestamp : 1632393216.289968 from P
    process, sent token : 0.567.
```

  otherwise, if the message is coming from S, and the content is indeed, the name of the signal, we will have the following log :

```
timestamp : 1632393216.996628 .
From S process signal: SIGUSR1.
```

- *PrintLogFile* : this function simply print the content of the log file in the terminal whenever it is called, and it will be used when process S receives the signal for dumping log.
- *signal_handler* : this function manages the reception of Posix signals. It checks which signal has been received from the user, and then actuates a certain behaviour, writing at the same time on the pipe between P and L the name of the signal received, together with the timestamp in which this happens.
  To get into this, we have to introduce a new type of variable, that is the one of the message sent :

```
typedef struct {
  char* sig_name;
  double timestamp;
}message;
```

  The type pf signals managed are :
  - SIGUSR1 that once received, stops all the processes that are active, in particular P, L and G.
  - SIGUSR2 that once received, resumes all the processes.

- SIGCONT that once received, print the content of the logfile with the *PrintLogFile* function.
- *ReadFile* : this function reads the content of the configuration file and stores the values into variables that will be used in the code.

The remaining part is the *main*, which is composed of several parts :
- pipe creation : in this part are created named pipes, that are the one between P and G, P and S, P and L.
- initialization of the client side of the socket between P and G.
- select part : where is managed the selection of the pipe from which read through the *select()* system call.
  In case of two pipes active (the one from G and the one from S), the message coming from S has an higher priority.

  If the message read is from G, the value is stored, and then starting from the previous one, a new token is computed in the following way [1] :

$$NewToken = ReceivedToken + DT * RF \quad (1)$$

  Where DT is the time difference between the timestamp when the token is sent by G, and the time instant when it is received by P. The other parameter RF is the reference frequency that is set in the configuration file.
  Once computed the new token value, a flag message is sent to process L to communicate that the type of message that is arriving is of type *token*, so that L will be able to call *WriteLog* function in the proper way. Then it proceeds sending the new token with also an updated timestamp value again through the pipe between P and L.

  If the message, instead, is coming from S, the message is stored, then a flag message is sent to L to advertise that a message of type *message* is arriving, and then its value is written on the pipe.
- process L : here L reads from the pipe between P and L. The first message read, is always a flag that can assume these values :
  - **1** : if the message is coming from G
  - **0** : if the message is coming from S
  Once understood from which process the message is arriving, L reads the rest of the message and save its value.
  If the message is coming from G, there will be two messages to read, the first one is the token written in the fifo between G and P, and the second is the value of the new token, both of them with the relative information about timestamps.

---

[1] The original formula has been substituted, since it didn't provide anything interesting, while the new formula shows a linear pattern that can be considered as a uniform rectilinear motion of the token

In case of a message coming from S, there will be only a message containing the name of the signal received and the relative timestamp.

In both situations (message from G or message from S), the process L writes through the function *WriteLog* what it has received.

- G execution : in this part is executed the code related to the server of the socket between P and G, and this is done through *execvp()* system call.

III. INSTRUCTIONS

To use this package, the reader needs to follow this instructions.

In the folder we have two C files, that needs to have the relative executable file. If these two files are not there, the user has to compile them in the following way :

```
$ gcc G_process.c -o G -lm
$ gcc main.c -o main -lm
```

Then to launch the program, execute the main file :

```
$ ./main
```
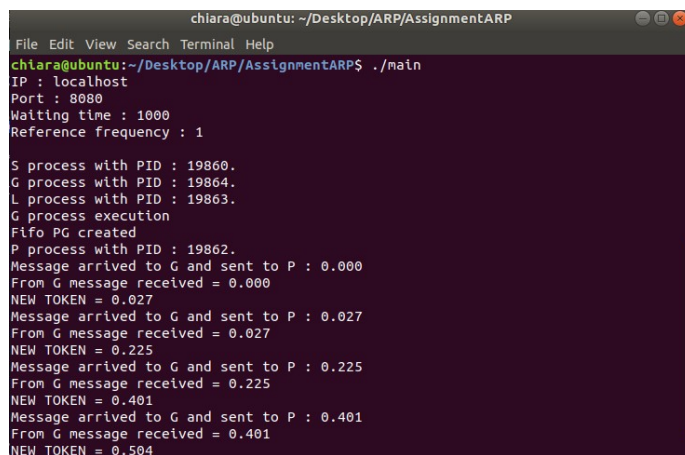
Whenever the user wants to send a signal to S process :

```
$ kill <SIG_NAME> <S_PID>
```

Where SIG_NAME is one of the signals managed by the signal handler (SIGUSR1, SIGUSR2 or SIGCONT) depending on what the use wants to do, and S_PID is the process ID of S that is printed on the shell whenever the program is launched and S is created, as weel as the other processes.

IV. RESULTS
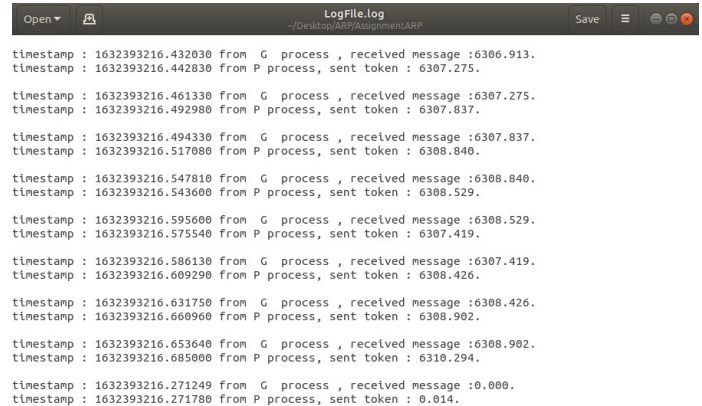
Once launched the main program, in the shell should appear something similar to what shown in Fig. 2.

Fig. 2. shell

At the same time is created the *Logfile.log* that in a normal situation (when there are no signals coming from the user) is structured as in Fig. 3.

Fig. 3. *LogFile.log* : no signals case

While, if the user sends a SIGUSR1 signal followed by a SIGUSR2 signal the resulting log will be as the one in Fig.4, where we can see that when is received the first signal, the logging process, as well as the other processes, is interrupted and resumed with the arrival of the second signal.

Fig. 4. *LogFile.log* received SIGUSR1 and SIGUSR2 case

3

If, instead, the user sends a signal SIGCONT, the execution of the other processes will be interrupted, allowing the process to print the content of the log in the shell (see Fig. 5).

Fig. 5. *LogFile.log* received SIGCONT case

Then when the entire *LogFile.log* has been printed, a command to resume (SIGUSR2) is waited as show in Fig. 6.

Fig. 6. *LogFile.log* received SIGCONT case

The data in the *LogFile.log* in the first situation (Fig. 3) can be filtered and used to obtain a graph showing the value of the generated tokens with respect to the timestamps (Fig. 7).
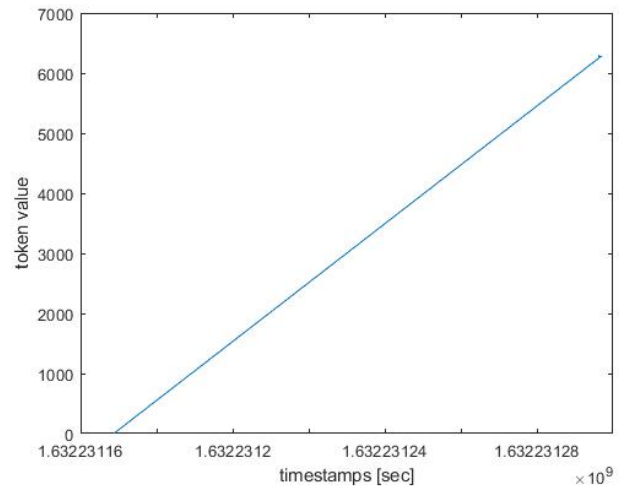

Fig. 7.

From this figure we can notice the linear pattern of the new tokens' generation, that can be brought back to a uniform rectilinear motion of the token that is travelling between the processes.

The user can find a MATLAB folder inside the package, containing the code used to generate this plot together with the filtered data used for this example.