Operating Systems

CSI3131 Assignment 1

Winter 2020

Process/thread creation and inter-process communication

You must submit your assignment on-line with Virtual Campus. This is the only method by which we accept assignment submissions. Do not send any assignment by email. We will not accept them. We are not able to enter a mark if the assignment is not submitted on Virtual Campus! The deadline date is firm since you cannot submit an assignment passed the deadline. You are responsible for the proper submission of your assignments and you cannot appeal for having failed to do so. A mark of 0 will be assigned to any missing assignment.

Assignments must be done individually. Any team work, and any work copied from a source external to the student (including solutions of past year assignments) will be considered as an academic fraud and will be forwarded to the Faculty of Engineering for imposition of sanctions. Hence, if you are judged to be guilty of an academic fraud, you should expect, at the very least, to obtain an F for this course. Note that we will use sophisticated software to compare your assignments (with other student's and with other sources...). This implies that you must take all the appropriate measures to make sure that others cannot copy your assignment (hence, do not leave your workstation unattended).

Goals

Practise:

1. process creation in Linux using fork() and exec() system calls
2. thread creation in Linux with the pthread API
3. inter-process communication using pipes

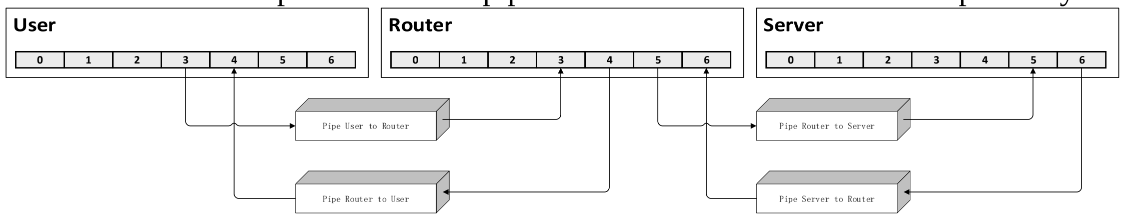
Posted: Jan 30, 2020
Due: Feb 13, 2020, 23:59

Description (Please read the complete assignment document before starting.)

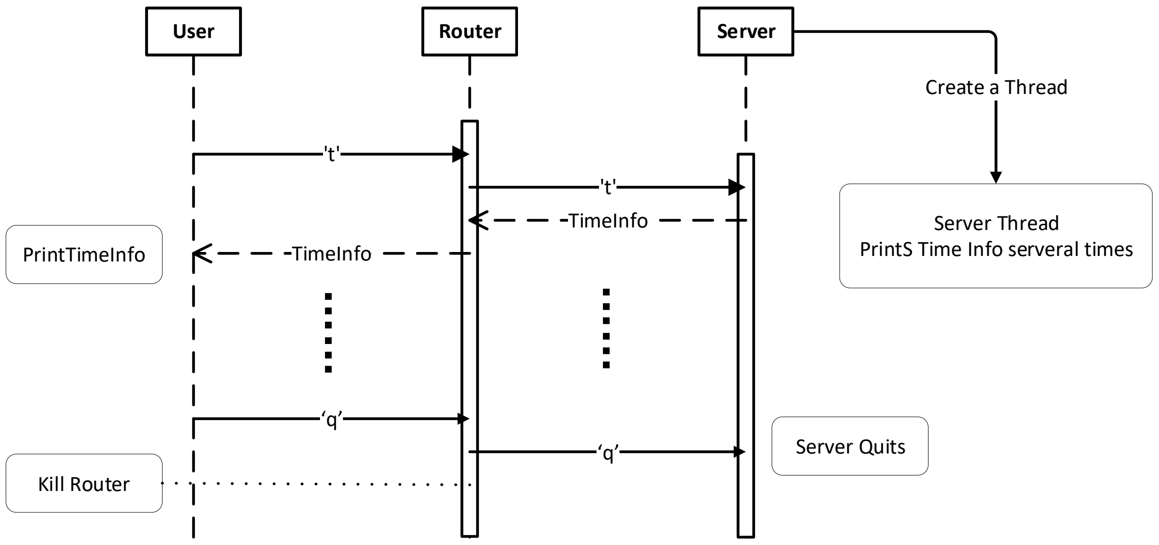
Two programs have been provided: *user.c* and *router.c*. Your task shall be to complete these two programs such that the execution of the program *user* creates three processes as follows:

1. *User* process: this is the original process invoked when *user* is run. It is responsible for creating the other two processes after which is simulates *User* that asks time to Server (simulated by the Server process).
2. *Server* process: this is a child process of the *User* process that simulates responding the commands from *User* and displaying time in a thread.
3. *Router* process: This process is executed by *User* and forwards any kind of data from User and Server process to each other.
4. Message Flow: *User* sends 't' to *Server* and *Server* responds with current time. *User* sends 'q' to *Server* and *Server* should be ended by itself. These commands should be forwarded by *Router* process. *Server* and *User* should not communicate with each other directly by any means.

The *User* and *Server* processes use 2 pipes to communicate with router respectively.

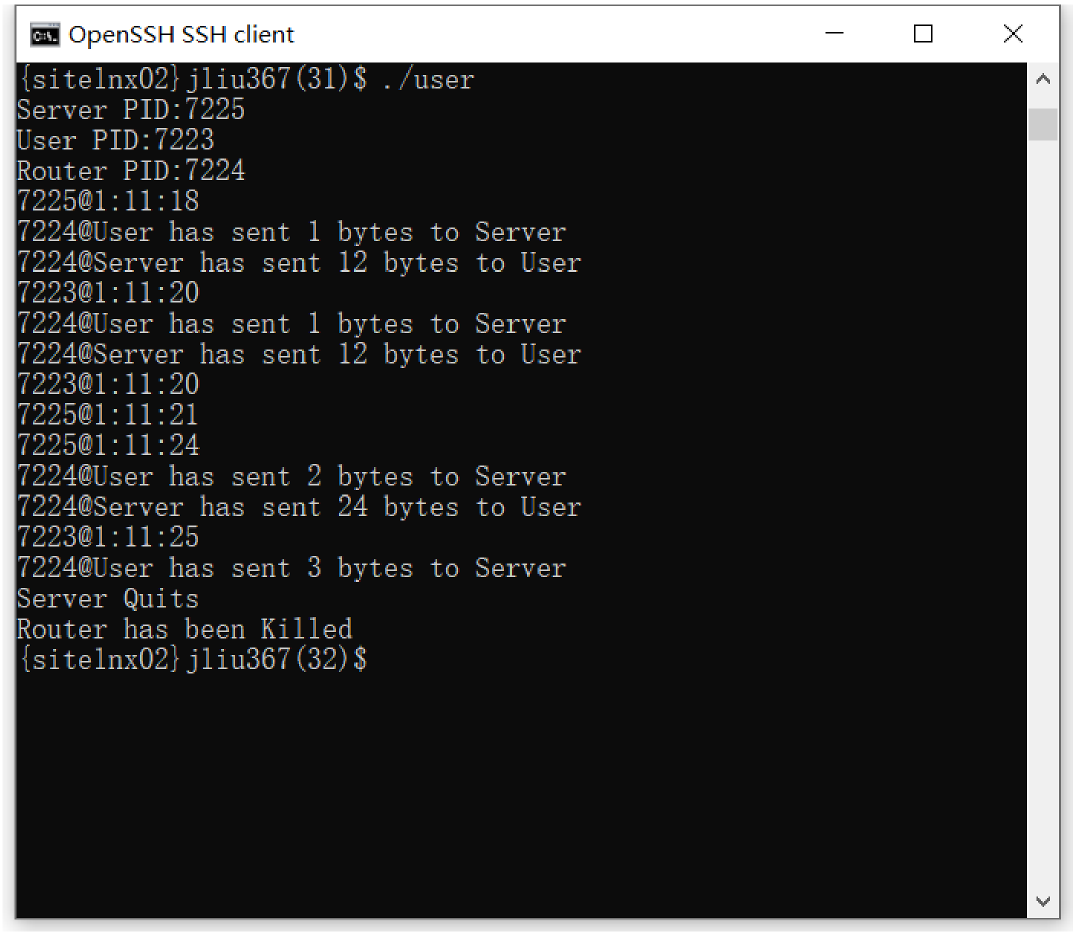


Their scheme should be implemented as following figure. First *User* creates the *Router* and *Server* process. And *Server* immediately creates a thread to print time every few seconds. Then *User* sends 't' command to *Router* and *Router* forwards the command to *Server*. Once *Serve* receives the 't' command, it should reply current time immediately. After *User* queries time for several times, it sends 'q' command to *Server*. And *Server* should identify this command and kill the thread and end itself. Then *User* will kill the *Router* process by using kill() and end up itself.



To complete the assignment:

1. Start with the provided files *user.c* and *router.c*. Complete the documentation in each file to indicate your name and student number. Take the time to document well your code. Consider taking the time to design your code before you start coding.
2. Utility functions has been given including:
    a. TimeInfo structure containing hour, minute, and second. It should be initialized by getTimeInfo
    b. closePipe
    c. getTimeInfo, which is used for get current TimeInfo
    d. printTimeInfo, which is to print the content of a TimeInfo
    e. printTime, which prints current time
    f. blockRead, which provides the read function using block mode and will not influent the read function in the system
3. The programs should be compiled using the command make.
4. To submit the assignment, upload the files *user.c* and *server.c*. Do not forget to click on the submit button after (and only after) you have uploaded the file.
5. See point 4 in “Background Information” for hints on how to observe processes/threads and pipes to help debug your programs.
6. When *user* is run, the following output should appear on your screen (Note that PIDs shall be specific to your execution).
   There are 10 functions you should implement in your code to get points, which is also mentioned in the comments. (10 marks each)
    a. Server process should create a thread to display current time for at least 2 times
    b. Server should respond the following command from Router
        -"t": send current time to the router
        -"q": stop waiting for the command, kill thread and exit.
    c. Server should end the process by itself instead of User process
    d. User queries time for at least 2 times by sending 't' command to Router
    e. Then User sends 'q' command to Router
    f. User waits for the Server ending itself
    g. User kills the Router process
    h. Router connects with User and Server via pipes
    i. Router receives any data from User and forwards the data to the Server
    j. Router receives any data from Server and forwards the data to the User



OpenSSH SSH client

```
{sitelnx02}jliu367(31)$ ./user
Server PID:7225
User PID:7223
Router PID:7224
7225@1:11:18
7224@User has sent 1 bytes to Server
7224@Server has sent 12 bytes to User
7223@1:11:20
7224@User has sent 1 bytes to Server
7224@Server has sent 12 bytes to User
7223@1:11:20
7225@1:11:21
7225@1:11:24
7224@User has sent 2 bytes to Server
7224@Server has sent 24 bytes to User
7223@1:11:25
7224@User has sent 3 bytes to Server
Server Quits
Router has been Killed
{sitelnx02}jliu367(32)$
```

Output Sample

Background information:

1. An open file descriptor is an integer, used as a handle to identify an open file. Such descriptors are used in library functions or system calls such as `read()` and `write()` to perform I/O operations.

2. In Unix/Linux, each process has by default three open file descriptors:
    a. Standard input (file descriptor 0, i.e. read(0,buf, 4) reads 4 characters from the standard input to the buffer buf). Typically, the standard input for a program launched from the command line is the keyboard input.
    b. Standard output (file descriptor 1).
    c. Standard error (file descriptor 2).
    d. When a command is run from the shell, the standard input, standard output and standard error are connected to the shell tty (terminal). So reading the standard input reads from the keyboard and writing to the standard output or standard error writes to the display.
    e. Note that many library functions used these file descriptors by default. For example *printf("String")* writes "String" to the standard output.
    f. From the shell it is possible to connect the standard output from one process to the standard output to another process using the pipe character "|". For example, the command "who | wc" connects the standard output from the who process to the standard input of the wc process such that any data written to the standard output by the who process is written (via a pipe) to the standard input of the wc process.

3. You will need the following C library functions:
    a. fork() – should be familiar from lectures
    b. pthread_create() – should be familiar from lectures
    c. pipe()
        - should be familiar from lectures
        - note that multiple process can be attached to each end of the pipes, which means that a pipe is maintained until no processes are connected at either end of the pipe
    d. execvp(const char * program, const char *args[]) (or execlp)
        - replaces the current process with the program from the file specified in the first argument
        - the second argument is a NULL terminated array of strings representing the command line arguments
        - by convention, args[0] is the file name of the file to be executed
    e. execlp(const char * program, const char *arg1, const char *arg2, … NULL)
        - replaces the current process with the program from the file specified in the first argument
        - the second argument subsequent arguments are strings representing the command line arguments.
        - by convention, arg1 is the file name of the file to be executed

f. dup2(int newfd, int oldfd) – duplicates the oldfd by the newfd and closes the oldfd. See http://mkssoftware.com/docs/man3/dup2.3.asp for more information. For example, the following program:

```
int main(int argc, char *argv[]) {
    int fd;
    printf("Hello, world!")
    fd = open("outFile.dat", "w");
    if (fd != -1) dup2(fd, 1);
         printf("Hello, world!");
       close(fd);
}
```

   will redirect the standard output of the program into the file outFile.dat, i.e. the first "Hello, world!" will go into the console, the second into the file "outFile.dat".

g. read(int fd, char *buff, int bufSize) – read from the file (or pipe) identified by the file descriptor fd bufSize characters into the buffer buff. Returns the number of bytes read, or -1 if error or 0 if the end of file has been reached (or the write end of the pipe has been closed and all data read).
h. write(int fd, char *buff, int bufSize) – write into the file/pipe bufSize characters from the buffer buff
i. close(int fd) – closes an open file descriptor
j. printf(): You may want to use the printf() function to format output. This function writes to the standard output (fd 1). But be careful since this function buffers output and does not write immediately to the standard output. To force and immediate write, used fflush(stdout). Alternatively, you may used sprintf(), to format the an output into a buffer and use write() to write to the standard output.
k. fprintf(): this is a version of *printf()* that provides the means to specify where output should be send. Use it to write to the standard error with *fprintf(stderr,"a message", arg, arg,…)*. This function is useful for debugging as it will write to the terminal in processes where the standard output has been redirected to a pipe.
l. getpid(): this function returns the PID of the current process. It is useful in creating messages printed on the screen to identify the source of the message.
m. Consult the manual pages (by typing 'man function_name', i.e. 'man fork') and/or web resources for more information.

4. Here is a hint at how you can observe the connection of processes to pipes. Insert long delays using the standard library function `sleep` (e.g. `sleep(300)`) to allow observation of processes, threads and pipes at different points in the execution of the programs.