CSCI 4061: Intro to Operating Systems
Project 2: Multi-Process Chat App

Posted Oct 22 – Due Nov 7, Midnight Groups of 3

1 Introduction

This lab focuses on three OS system concepts. First, the use of processes to decompose an application and to provide isolation (i.e. processors can fail independently without impact). Second, the use of interprocess communication (or IPC) to coordinate and communicate among the processes. Third, the use of "polling" to implement asynchrony in the absence of threads. To gain experience with these concepts, you will implement a simple "local" multi-party chat application using a multi-process architecture. Note that a real chat program would be client/server and span many machines. In contrast, the chat processes in our solution will all run on a single machine "locally". Your chat-service will have a central chat server, which handles all of the management of the chat, and waits for "users" to connect to the server, supporting both private peer-to-peer chatting and group chatting. The chat group will contain all users that are connected to the server and only one such group will be supported at a time.

In this architecture, isolation via processes is very important as users need to be completely isolated from each. Users may join and leave the chat, or their chat code may fail unpredictably, but the chat service should keep running until the chat server decides otherwise. To enable communication, you will use UNIX `pipes` as we will describe.

2 Description

Our chat service will be provided on single centralized server. When a user wishes to chat, they connect to the central server. The server process then creates an associated child process the user, through which users will be participating in the chat. **You will design such a multi-process chat application** in this assignment, given initial code. Your chat application will consist of one main parent process (called the SERVER process) and several child processes that each communicate with their associated user process. There is a unique child process for each connected user. The SERVER process will have zero or more child processes corresponding to each user currently connected to the chat server.

2.1 SERVER process

The SERVER process is the main parent process, which will run when the chat server program is started. It is responsible for forking child processes for each user. The SERVER process provides an interface for following administrative functions listed below.

1. *list* : List all of the users currently connected to the server. Print '<no users>' if there are no users currently.
2. *kick* <*username*> : Kick the specified user off the chat session.
3. *exit* : Terminate all user sessions and close the chat server as well.
4. <*any-other-text*> : Broadcast this text to all of the user processes with prefix, "*admin:*". Do nothing, if no users are connected to the chat. The users will print out the message. admin: <any-other-text>

2.2 USER process

The user process is another program that you will write to provide an interface to the server in the chat. When a user starts a USER process, this will connect to the SERVER and interact with SERVER given a *pipe*. **We will provide the code for the initial communication between the SERVER and USER processes.** The USER process must display the name of the user as part of the prompt. The USER process will have some commands of its own. All the user commands are listed below.

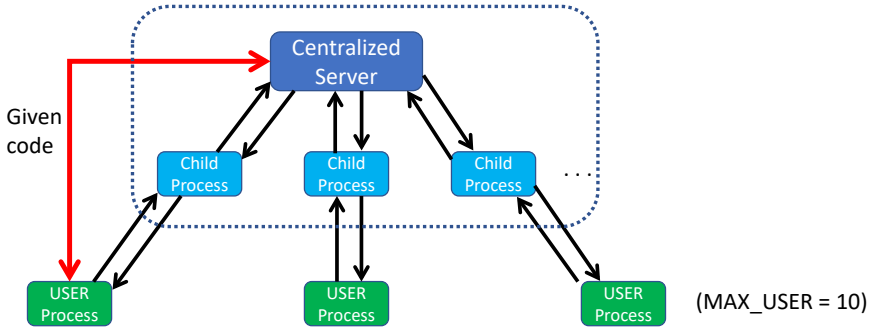

Figure 1: IPC among various SERVER and USERs processes.

1. *list* : Same as in the SERVER process. Output should be printed in the user's process terminal window.
2. *exit* : Disconnect this user. Terminate the USER process and remove them from the chat session.
3. *p2p* <*username*><*message*> : Send a personal message (<message>) to the user specified in <username>. Print error if the user is invalid.
4. <*any-other-text*> : Same as in the SERVER process but without a prefix. Broadcast this text to all user processes.

3 Forms of IPC

You will implement `pipe` communication (between processes) for this multi-process chat application. For each pair of communicating processes, you will have a read pipe and a write pipe as **in the knock-knock example in class**.

Fig. 1 demonstrates the IPC among the different processes. There are different kinds of messages exchanged among the various processes, corresponding to various commands described in the previous section. The centralized SERVER process waits for user connections. When a USER process connects to the SERVER, the SERVER creates two pipes to communicate with the user and another two pipes to communicate with the associated child process. The SERVER, then creates this child process that will communicate with the user.

The role of child processes is to forward messages between SERVER and USERS processes. USER processes do not communicate with the SERVER process directly but only to the associated child process. An exception is the initial connection from the USERS to the SERVER. We will provide code for this initial communication between a USER and the SERVER.

For each of the user commands entered from a user, a message will be sent to the corresponding child process which then sends it to the SERVER. The SERVER will process the command and send the result to the corresponding child process via pipes. Finally, the result will be sent to USER via pipes written by the associated child process. The user then outputs the result to the terminal. For instance, if the *p2p* command is used, the SERVER will parse the command string to figure out the destination user and its child process. Then, it sends the message to that user using the appropriate child process pipe.

4 Program Flow:

This section describes the flow of each of the involved processes.

4.1 SERVER Process:

When you invoke your server, the `main()` function of your program starts executing. Upon its invocation, the SERVER process performs the following tasks:

1. *Waiting for connections from users* :
   (a) The SERVER will call `setup_connection`(char * `server_id`). You will need to pass your own `server_id` to set up the SERVER and your USER will use it to connect to the SERVER.
   (b) The SERVER then *polls* (via non-blocking `get_connection`(char * user_id, int `pipe_child_writing_to_user[2]`, int `pipe_child_reading_from_user[2]`)) to check for a new user connection. The `get_connection`() returns -1 if there is no new user connection. In the `get_connection` function, two pipe arrays are internally created. You need to pass the two *pipe* arrays as parameters.
   (c) Once the SERVER gets a new connection, the SERVER needs to create two *pipes* for *bidirectional* communication with a child process and it creates a child process.
   (d) In a child process, the child process will have four *pipes* in total, i.e., two *pipes* for *bidirectional* communication with a user and two *pipes* for *bidirectional* communication with the SERVER.
2. *Processing server commands* :
   (a) The SERVER will get input for administrator commands. To this end, the SERVER *polls* (via non-blocking *read*) from *stdin (0)*.
   (b) You will need to use `fcntl` function to make *stdin* NON_BLOCKING:
   `fcntl`(0, F_SETFL, fcntl(0, F_GETFL) | O_NONBLOCK);
   (c) The non-blocking read will read data from *stdin*. If no data is available, the *read* system call will return immediately with return value of -1 and *errno* set to EAGAIN. In that case, simply continue the *polling*.
   (d) If *read* returns with some data, read the data into a buffer and handle the command. There are four types of commands that the SERVER process may *read* from *stdin*.
      i. *\list* : When this command is received, the SERVER creates a string with the names of all the active users and prints them. Print '<no users>' if there are no users currently.
      ii. *\kick <username>* : This command will be used to terminate a particular user's session. The SERVER should terminate the session for this user by `kill`ing its child process. This user should be removed from the user list as well. Care should be taken to clean-up the user's pipes and any zombie processes as well.
      iii. *\exit* : The SERVER should cleanup *all* of the users, terminate all their processes, and cleanup their pipes and wait for all child processes to terminate. Each child process should cleanup the pipes for a user and exit.
      iv. *<any-other-text>* : Any other text entered should be sent to all the active users' pipes with the prefix, "Notice:". All users' processes should print out this text as is.
3. *Processing user commands* :
   (a) The SERVER then *polls* (via non-blocking *read*) on the set of open *pipe* file descriptors created when forking the child process, in a loop. There might be no child process at this stage if there is no users. The termination condition for the loop is when the server's exit command is used on the SERVER process.
   (b) The non-blocking read will read data from the pipe. If no data is available, the *read* system call will return immediately with return value of -1 and *errno* set to EAGAIN. In that case, simply continue with the *polling*. **To reduce CPU consumption in the loop, you will use `usleep` between reads. Put this in your loops ASAP.**

(c) If *read* returns with some data, read the data into a buffer. There are four types of messages that the SERVER process may *read* from the child processes pipe.

i. *\list* : Same as above. The list of active users will be generated and sent to the requester via the child process' pipe. The user process should print this list. Print '<no users>' if there are no users currently.

ii. *\p2p <username><message>* : This command is used by a user to send messages to a particular user. When this command is received, the SERVER should search for the specified user in its user list, extract the message from the command string and send it to the addressed user through a pipe write. An error should be printed in the appropriate window if the *username* does not exist.

iii. *\exit* : When this command is used on a user's process, the SERVER should clean-up that user, remove it from the user list, terminate the child process associated to the user, and cleanup the corresponding pipes.

iv. *<any-other-text>* : Any other text entered should be sent to all the active users. All the users' processes should print out this text as is (prefixed by the sending user).

4.2 Child Process:

1. As explained, the role of the child process is to forward messages between SERVER and USERS processes. Child process will *poll* (via non-blocking *read*) on the *pipe* for an associated user and the SERVER.
2. If *read* returns with some data, read the data into a buffer and forward it to SERVER or USERS through the correct *pipes*.

4.3 User Process:

1. The user process starts by calling `connect_to_server`(char `server_id`, char * `user_id`, int `pipe_user_reading_from_server`[2], int `pipe_user_writing_to_server`[2]). This function will return -1, if the connection failed. You need to pass the `server_id` to connect to the SERVER. You will need to pass the user id (on the command line) to be used for the chat when the user process is started. Lastly, you need to pass two *pipe* arrays as parameters to obtain *pipes* that will be used for communicating with SERVER.
2. For user commands, a user process *polls* (via non-blocking *read*) from *stdin (0)* as done in the SERVER process commands.
3. If *read* returns with some data, the process will read the data into a buffer. Then, it will send data to the child process through `pipe_user_writing_to_server`.
4. The commands that a user can send, are *list*, *p2p*, *exit*, and <any-other-text>, as explain in previous section.
5. The user process then *polls* (via non-blocking *read*) on *pipe_from_server* to read data from an associated child process. Again, to reduce CPU consumption in the loop, you will use `usleep` between reads.
6. If *read* returns with some data from the child process, it will be printed out.

5 Error Handling:

You are expected to check the return value of all system calls that you use in your program to check for error conditions. If your program encounters an error (for example, if an invalid user name is supplied in the *\p2p* command), a useful error message should be printed to the screen. If any error prevents your program from functioning normally, then it should exit after printing the error message. (The use of the perror() function for printing error messages is encouraged.) Upon executing the *\exit* command on the SERVER, the main SERVER process and its child processes (if exist) must exit properly, cleaning-up all of the users, waiting for all child processes and freeing up any used resources.

6 Extra Credit:

Handle crash failures, e.g. a control-c can be hit in a USER or SERVER window. If this happens, handle it. A crashed USER should be cleaned up and not disrupt the SERVER. A crashed SERVER should allow every USER to terminate automaticaly. Think about how you can detect failure via pipes. Be careful to take care of zombie or/and orphaned processes. Add another command: *seg* – create a segmentation fault in the user process by any means (e.g. char *n = NULL; *n = 1;). The result of this should be that the SERVER cleans up that user and otherwise the chat should run smoothly. **ALL** of these must be implemented for extra-credit.

7 Implementation Notes:

Some useful items.

1. Remember that the SERVER and USER processes must be running on the same machine.
2. We will be providing most of the nitty-gritty C/parsing type code.
3. To kill a process, use the `kill` system call.
4. When you create/get pipes remember to close the ends that the process does not need.
5. Don't forget to remove a new line (\n) when you read inputs from *stdin*.
6. Hint: The read will return 0 if the pipe is closed (broken). You will need to check this return value to detect if the user or server processes are failed (terminated). **You must close pipes appropriately (not used for a process) to detect if the pipe is valid or not.**
7. If the the user connects with a name already used, the connection will be closed.
8. **Remember to sleep via usleep in all polling loops!**
9. This lab can be done with static memory allocation – you do not need to allocate any dynamic memory. For this reason, `memset` is a handy call to zero out a buffer for repeated use (something you may want to do).
10. You may need to build up strings for message, use (`sprintf`) for this.
11. To manipulate strings, functions like `strcpy`, `strncpy`, `strtok`, `strlen` may be handy. But, remember to **ALLOCATE** the memory for any strings you are creating.

8 Suggestions

Some suggestions regarding implementation.

1. Dividing the work among team members:
   The project could be divided into segments that may be done by different members. One way is to divide it into the following:
   (a) Writing the server program (creating child processes).
   (b) Writing the command handlers in the SERVER program.
   (c) Writing the user program.
2. Recommended steps to get going:
   (a) Start with writing a SERVER and USER, which prints a prompt, reads any input provided and fills up a string with that input.

(b) Write a basic SERVER that waits connections from USERS. Try to get the message passing working between the SERVER and child processes.

(c) Write a basic USER that connects to the SERVER and gets two pipe file descriptors to communicate with the server.

(d) Get the message passing working between the USER and child process.

(e) Implement all commands, putting all of the above pieces together.

9 Grading Criteria

5% README file. Be explicit about your assumptions for the implementation.

20% Documentation with code, Coding and Style. (Indentations, readability of code, use of defined constants rather than numbers, modularity, non-usage of global variables etc.)

75% Test cases

1. Correctness: Your submitted program does the following tasks correctly:
   (a) Starts the server and get inputs.
   (b) Demonstrates correct usage of pipes by transferring different kinds of messages as their intended purpose - broadcast as well as peer-to-peer.
   (c) all exiting is done cleanly.
2. Error handling:
   (a) Handling invalid user name specification in the commands.
   (b) Exiting the SERVER should close all the other user sessions.
   (c) Error code returned by various system/wrapper-library calls.
   (d) There should be no ”Broken-Pipe” error when your program executes. Also, appropriate cleanup must be done whenever any of the child-processes (SERVER, child, and USER processes) terminates. For eg., closing the pipe ends.

10 Documentation

You must include a README file which describes your program. It needs to contain the following:

1. The purpose of your program
2. A brief description of who did what on the lab
3. How to compile the program
4. How to use the program from the shell (syntax)
5. What exactly your program does
6. Any explicit assumptions you have made
7. Your strategies for error handling

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion.

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability. At the top of your README file and main C source file please include the following comment:

```
/* CSci4061 F2018 Assignment 2
* section:  one_digit_number
* date:  mm/dd/yy
* name:  full_name1, full_name2 (for partner)
* id:  d_for_first_name, id_for_second_name */
```

11 Deliverables:

1. Files containing your code
2. A README file (readme and c code should indicate this is assignment 2).

Note: You will need to provide makefile to be used by us to compile your program with the standard make utility. All files should be submitted on the class canvas site. This is your official submission that we will grade. Please note that future submissions under the same homework title OVERWRITE previous submissions; we can only grade the most recent submission. Communicate effectively, work together, share the load, and submit ONE solution

All files should be submitted using the SUBMIT utility. You can find a link to it on the class website. This is your official submission that we will grade. We will only grade the most recent and on-time submission.