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

**Posted Feb 24 – Due Mar 10, Midnight** Groups of 3

# 1 Introduction

This lab focuses on two 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. 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". In addition, YOU will be acting as both the *chat/server administrator* as well as the "chatting users" using a command-line interface, as we will describe. Your chat-service will have a central chat server, which handles all of the management of the chat, and enables "users" to "log in" 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 provide a command-line interface (CLI)-based interface to the server admnistrator, as well as to the users. There will be a separate process for each user currently logged in to the server. "Logging in/Connecting" to the server is emulated by a simple interactive command executed at the server. Associated to each user process will be a separate terminal window, through which they will be participating in the chat. **You will design such a multi-process chat application** in this assignment, given some initial code. Your chat application will consist of one main parent process (called the SERVER process). This server process will have one child process (called the SHELL process), which will act as the interface for the *chat/server administrator*. In addition, the SERVER process will have zero or more children corresponding to each user currently "logged in" to the chat server. These processes will be running as separate `xterm` windows, each window running its own instance of the SHELL process as the interface to the user.

## 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 the SHELL process. This SHELL process will be the *chat/server administrator*'s interface to the server. Using this shell, the *chat/server administrator* can issue commands to perform various functions listed below.

1. *add* <*username*> : Add the given user to the chat session. A new `xterm` window should be opened for the new user. (`xterm` is the standard terminal emulator for the X Window System. We will use `xterm` in this assignment as is available through a CLI command on most of the CSELab machines.) This new window should invoke a SHELL process of its own that will act as the user's interface for chatting. We will provide the `xterm` command. <*username*> can be any string.

2. *list* : List all the users currently logged in to the server. Print '<no users>' if there are no users currently.

3. *kick* <*username*> : Kick the specified user off the chat session. The server should take care of exiting this user's SHELL process and closing their `xterm` window.

4. *exit* : Terminate all user sessions and close the chat server as well.

5. <*any-other-text*> : Broadcast this text to all the user windows. Do nothing, if no users are logged in.

### 2.2 SHELL process

The SHELL process is another program that you will write to provide an interface to both the *chat/server administrator* and all users in the chat. This will provide a running prompt and interactive interface both both chatting users and the *chat/server administrator*. The same SHELL program is to be used for both the *chat/server administrator*'s interface and the users' interface. The SERVER process will start its SHELL as a direct child process as specified in the previous subsection. But for each new user, the SERVER process will first fork an `xterm` window as a child process. This `xterm` window will be assigned to the added user. The window in turn will start the same SHELL program inside it, which will act as the user's interface. Both of these actions (launching the `xterm` window and running the shell program in it) can be done by *exec*ing a single command that will be provided to you with the code.

The user shell should display the name of the user as part of the prompt. The user shell will have some commands of its own. All the user commands are listed below.

1. *list* : Same as in SERVER process. Output should be printed in the user's window.

2. *exit* : Log off this user. Essentially close this chat window and remove this user from the chat session.

3. *p2p* <*username*><*message*> : Send a personal message (<message>) to the user specified in <username>. Print error if invalid user.

4. *seg* : Create a segfault 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.

5. <*any-other-text*> : Same as in SERVER process. Broadcast this text to all the user windows.

## 3 Forms of IPC

You will implement parent-child `pipe` communication 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.

### 3.1 Inter-Process Communication Infrastructure:

Fig. 2 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.

For each of the server commands described earlier, when the command is entered in the server shell, a message will be sent to the server process. The server will process the command and generate its output and send it out on the required pipes. So for example, if the server shell has issued the *list* command, the server process will form a list of all users and send it back to the server shell. If *add* has been issued, the server will add the user in its user list, start the user `xterm` window and its shell process and send a text message back to the server shell indicating that a user was added.

For each of the user commands, when the command is entered in the user shell, a message will be sent to the server process. Again the server will process the command and send the result out on the corresponding pipes. For instance, if the *p2p* command is used, the server will parse the command string to figure out the destination user and send out the message to that user using the appropriate pipe.

## 4 Program Flow:

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

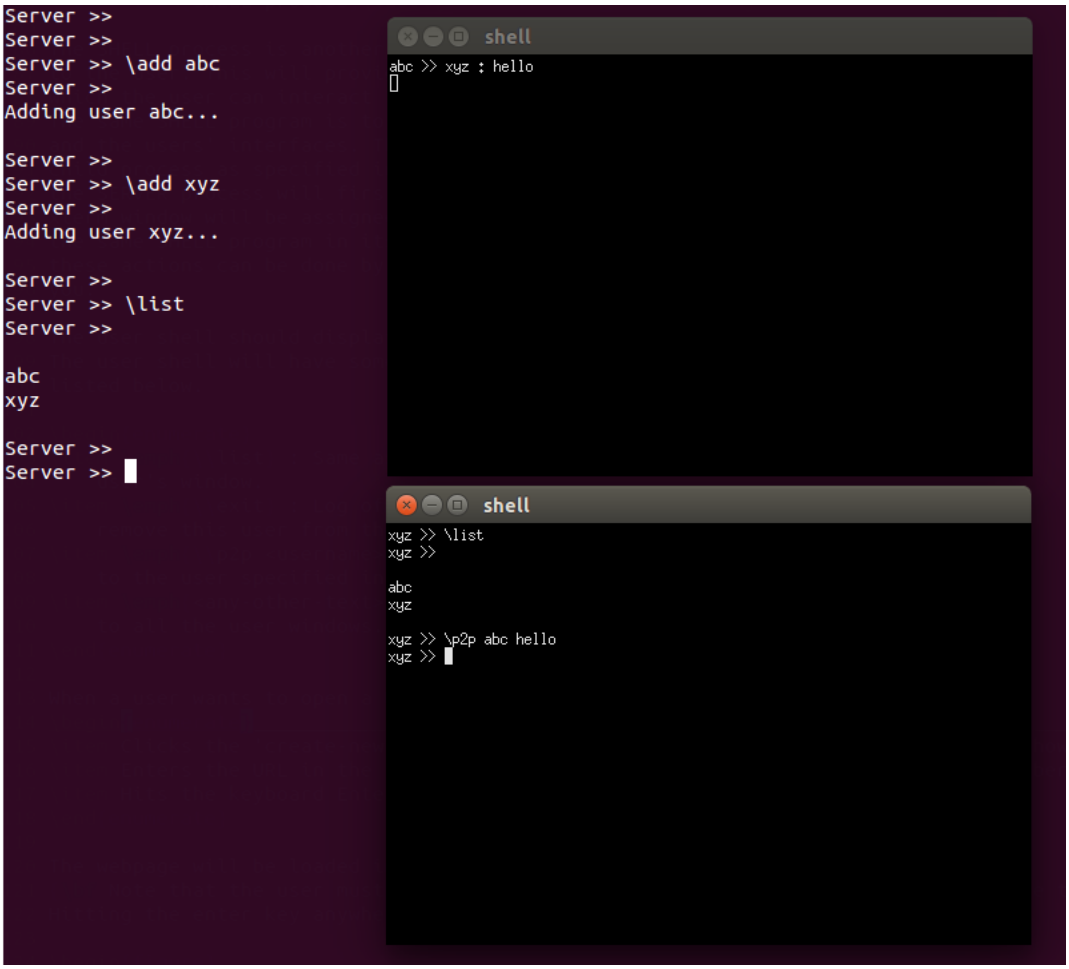


Figure 1: A sample of what the entire app might look like

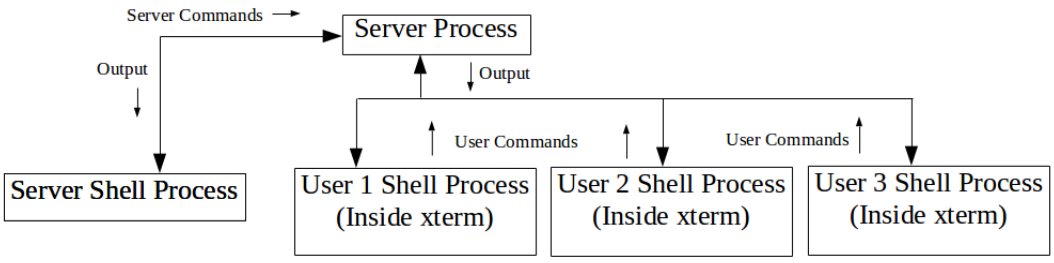


Figure 2: IPC among various browser process

## 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. *fork* the SHELL program: This program will act as the interface to the server.

   (a) Before this, the SERVER process creates two *pipes* for bi-directional communication with the SHELL process.

   (b) Second, it *forks* the new child process, the SHELL process. The pipes that the server had just opened are passed to the server SHELL process as arguments so that the server SHELL process knows which pipes it needs to read and write on for communicating with the server. Also the name of the server is passed to the SHELL process so that it can be displayed as part of the SHELL prompt. See Section 4.2 for a description of the SHELL program.

2. Waits for requests from child processes:

   (a) The SERVER process then *polls* (via non-blocking *read*) on the set of open *pipe* file descriptors created when forking the child process (see step b above), in a loop. At this stage, there is at least one child process (the SHELL process), which the SERVER polls for messages. The termination condition for the loop is when the server's exit command is used on the server SHELL 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* (see step 1(b)). To reduce CPU consumption in the loop, you will use `usleep` between reads.

   (c) If *read* returns with some data, read the data into a buffer. There are six types of messages that the SERVER process may *read* from the SHELL's pipe.

      i. *add* <*username*> : Upon receiving this message, the SERVER process first creates two *pipes* for bi-directional communication with a SHELL process which will be created for this new user. The SERVER then forks an `xterm` window with the SHELL program running inside the window (the command that you need to *exec* for this will be provided). The pipes that the server just opened are to be passed to the SHELL process as arguments so that the SHELL knows which pipes it needs to read and write on for communicating with the server. Also, the name of the user is passed so that the SHELL can display that name as part of its prompt. *The \add command can be used only on the SERVER SHELL.*

      ii. *list* : When this command is received, the server creates a string with the names of all the active users and sends it back to the requester's SHELL process. The SHELL process should print this list out. Print '<no users>' if there are no users currently. *This command can be used on both the SERVER's as well as any USER SHELL. The output should be printed in the window of the requester.*

      iii. *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 SHELL program and then the `xterm` window. 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. *The \kick command can be used only on the SERVER SHELL.*

      iv. *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 if the *username* does not exist. *The \p2p command can be used only on a USER SHELL.*

      v. *exit* : *This command can be used on both the SERVER as well as any USER SHELL.* When this command is used on a user's shell, the server should clean-up that user, remove it from the user list, terminate all its child processes, close its `xterm` window and cleanup the corresponding pipes. When

the command is used on the server's shell, the server should cleanup *all* the users, terminate all their processes, close all `xterm` windows and cleanup their pipes. After this the server should terminate the server SHELL as well, cleanup the pipes and exit, thereby marking the successful completion of the program.

vi. *<any-other-text>* : Any other text entered, either on the server's shell that is not among the server commands or on the user shell that is not part of the user commands, falls in this category. This text should be sent to all the active users' pipes. All the users' SHELL processes should print out this text as is.

### 4.2 SHELL Process:

1. The SHELL process is created by the SERVER as described in Section 4.1. There can be two types of SHELL processes:
   (a) The server SHELL created for the *chat/server administrator*.
   (b) The user SHELL created for each active user.
2. Upon its creation, the SHELL process invokes its own *main()* function. The SHELL process performs the following tasks: *(Note: The specification for the SHELL process is common for both, the server's shell as well as the users' shells.)*
   (a) *fork* a child process: This child process waits for messages from the server. It polls the set of pipes passed as arguments to the SHELL program, in a loop. Whenever a new message comes in, the child process will print it out as is on STDOUT.
   (b) Starts the user interface prompt in the parent : The SHELL process first sends back the pid of the child created above in part (a), back to the server through the pipes. This is to enable the server to terminate the child of the SHELL process during cleanup. The parent SHELL process also starts a loop, which presents a prompt to the shell's user. The prompt should display the name of the user. Also, whenever some text is entered at the prompt, the SHELL program should send it to the SERVER program through a pipe write, *except in one case stated below*. The text can be any of the shell commands described earlier.
   *seg* : Create a segfault in the user process by any means. This is the only command that will not be sent to the server. It will need to be parsed by the SHELL program. When this command is seen, the SHELL program should generate a segfault. The cleaning up of the user should be taken care of by the SERVER automatically by detecting that the user shell has died. *Note: This command will only be used on the USER SHELLS.*

## 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.) You should also take care of zombie or/and orphaned processes (e.g. if a user exits either through the *exit* or *seg* commands, or by clicking the cross button of the `xterm` window, make sure that the SHELL and `xterm` processes for that user and all of their child processes are removed). Upon executing the *exit* command on the SERVER shell, the main SERVER process and its child shell must exit properly, cleaning-up all of the users, waiting for all child processes and freeing up any used resources.

## 6 Implementation Notes:

Some useful items.

1. We will be providing most of the nitty-gritty C/parsing type code.
2. To kill a process, use the `kill` system call
3. When you create pipes remember to close the ends that the process does not need.
4. To detect a failed process, the pipe read will return 0 AND a waitpid with WNOHANG can be used.
5. On a windows laptop, you may need to run a X window server, `xming` is a free one. Linux normally has one already installed.
6. If you `ssh` to a Linux CSE machine you may need to enable X-11 forwarding in your `ssh` client.
7. For closing an XTERM window, you may again use the `kill` system call.
8. You may need to convert from integers to strings (`sprintf`) or from strings to integers (`atoi`).
9. To create strings, functions like `strcpy`, `strncpy`, `strtok`, `sprintf` are all handy. But, remember to **ALLOCATE** the memory for any strings you are creating.

# 7 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 SHELL program.
   (b) Starting the XTERM program with a running SHELL.
   (c) Writing the command handlers in the SERVER program.
2. Recommended steps to get going:
   (a) Start with writing a basic SHELL, which prints a prompt, reads any input provided and fills up a string with that input.
   (b) Write a basic SERVER that forks a SHELL program. Try to get the message passing working between the SERVER and the SHELL.
   (c) Setup the code to start a single user XTERM window from the SERVER program. Get the SHELL program running inside the XTERM.
   (d) Get the message passing working between the SERVER shell and the USER shell.
   (e) Now go ahead and implement all the commands, putting all of the above pieces together.

# 8 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 (50%): Your submitted program does the following tasks correctly:
   (a) Starts the `xterm` window and shell prompt correctly for new users. (5%)
   (b) Demonstrates correct usage of pipes by transferring different kinds of messages as their intended purpose - broadcast as well as peer-to-peer.(30%)
   (c) Terminates the users correctly via *exit* command as well as directly closing the `xterm` window, as well as a segfault. Credits will only be granted for this aspect of correctness if after exiting the main server, no zombie OR orphaned processes remain in the system. (15%)
2. Error handling(25%):
   (a) Handling invalid user name specification in the SHELL commands.(5%)
   (b) Exiting the SERVER shell should close all the other user sessions. (12%)
   (c) Error code returned by various system/wrapper-library calls. (3%)
   (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 SHELL/USER SHELL/`xterm` processes) terminates. For eg., closing the pipe ends.(5%)

# 9 Documentation

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

1. The purpose of your program
2. How to compile the program
3. How to use the program from the shell (syntax)
4. What exactly your program does
5. Any explicit assumptions you have made
6. 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 S2016 Assignment 2
* section:  one_digit_number
* section:  one_digit_number
* date:  mm/dd/yy
* name:  full_name1, full_name2 (for partner)
* id:  d_for_first_name, id_for_second_name */
```

# 10 Deliverables:

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

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.