CS 4348/5348 Operating Systems -- Projects 2&3

The goal of Projects 2&3 is to let you get familiar with some process related system calls in Unix (or Linux) systems, including how to create processes and how to communicate between processes. The projects have to be programmed in C or C++ and executed under Unix (or Linux and alike). Each project is divided into multiple phases to help you build a working program. You only need to submit your final program for each project. If you cannot get your final program to work, you can submit a working program for an intermediate phase. Non-working code will get 0 point.

1 Project 2 Phase 1: Standalone Shell-Shell

The first project in many OS courses is to build a simple shell, which is the user interface to the operating system. Instead, we will build a shell on shell, which takes user commands and pass to shell for processing. We call this program Shsh and the executable name should be "**shsh.exe**".

Also, in Unix, everything in execution is a process, including those OS programs. After the system boots up, the first process is created, which forks many other processes to offer operating system and other system services. When you login, a process executing the shell program is created. In the shell process, when you execute a program, a new process is created. In Unix, this process creation operation is also offered as a system call to the users. We will use "fork" system call to create processes and use other system calls to enable communication between the processes we have created.

In the Shsh program, you need to process 3 types of commands, "**cmd**", "**pipe**" and "**exit**". We call these the **shshcmd**. You first print the command prompt "**cmd>** " and then read the command from stdin and process it.

The cmd command starts with the keyword "cmd" and followed by any shell command. You need to retrieve the shell command (let's call it **shcmd**) following cmd and use the system call "**system**()" to execute it. For example, if Shsh reads the following shell command

```
cmd> cmd cd testdir; ./a.out
cmd>
```

it should execute system("cd testdir; ./a.out"). Note that the "system()" system call will result in the creation of a new subprocess executing the shell program. Note that you have to distinguish which commands will only cause effects internal to the process and which ones will result in side effects external to the process. For example, if you execute cd in process *p*, which changes *p*'s cwd, but the parent Shsh process will not get the effect of cd. On the other hand, if a subprocess creates a file, which is an external effect, the parent process of course will be able to see the file. Thus, we need to assume that the commands issued to Shsh are independent, as though being issued in different sessions.

The pipe command starts from keyword "pipe", followed by a sequence of shell commands (shcmd) separated by ";". An example pipe command is given below. Your Shsh should **fork** one subprocess for each shell command in the pipe command and pass the output from one process to the next using the pipes created by the "**pipe**()" system call. You need to create and use the pipes carefully to achieve correct data flow. In order to force the "system()" system call to use the proper pipes as input and output, you need to map stdin and stdout to the pipes by using **dup2**() system call for the corresponding subprocess.

```
cmd> pipe cat proj2.txt; sed -e "s/fork/create-process/g"; cat > out
cmd>
```

The exit command simply asks Shsh to terminate.

When Shsh starts, it should obtain the process ids of its own and its parent's using the **getpid**() and **getppid**() system calls, respectively. Then, it prints "**Shsh process *pid* has been created by process *ppid***". When it terminates, it prints "**Shsh process *pid* terminates**". (Italic font indicates that the to-be-printed item is a variable.) When receiving a shshcmd, Shsh should print out "**Shsh processing *shshcmd***", where *shshcmd* is the entire shshcmd command Shsh has received. For each pipe command, Shsh should print out the entire configuration in multiple lines. Each line should be "**Shsh forked *pid* for *shcmd*, with in-pipe *pipedesc* and out-pipe *pipedesc***", where the two *pipedesc*s are different and they are the file descriptors of the input and output pipes for the corresponding process.

2 Project 2 Phase 2: User Manager

For this phase, you will implement a user manager (UM) to manage multiple simulated users using the system. The executable for UM should be "**UM.exe**". Each user has a ***userid***, which is of type **integer**. Since UM reads its commands from one window, we need to attach *userid* for every command. The five UM commands are "***userid* login**", "***userid* logout**", "***userid shshcmd***", "***userid* pre**", and "**0**". The userid value starts from 1 and 0 in the userid field means to terminate the UM. UM prints out command prompt "**ucmd**> " and reads and handles user commands.

In Unix systems, a shell process is created upon each user login. Similarly, when a user logs in to UM, UM uses **fork**() to create a Shsh process (U4) for the user. UM also creates pipes to communicate with each user's Shsh (U1). For each "*userid shshcmd*" command, UM simply passes the shshcmd (without userid) to the Shsh process of the corresponding user via one pipe and waits for its response from the other pipe (U6*). Note that you need to change your Shsh program to run as functions within the UM. If you use exec() to run the Shsh process, the pipe and current directory settings cannot be passed. (You can use named pipe if you use exec(). But we will not explore it in this project.)

To avoid garbling the output of different users, UM will create one file for each user to store the outputs of the user (U2). For a user with userid = *uid*, its output should be written to "***uid*.out**" **by UM**.

A shshcmd may require multiple rounds of user-Shsh interactions (e.g., shshcmd = "cmd ./a.out" and a.out needs to read input data). Also, one shshcmd may produce multiple output objects. For the first case, the user would know what Shsh wants and can send the subsequent input data. To safely pass the inspection of the UM, we use the "**pre**" command to indicate that the message is following the previous shshcmd. In the second case, UM should read the pipe from Shsh continuously till it receives the next Shsh command prompt "cmd> " (U6**).

Note that each user of the UM should have a consistent view of her/his own Shsh environment. For example, if user $u1$ creates a file "xyz.c", "xyz.c" should not appear in user $u2$'s environment. To achieve this, we let each user have a home folder and assume that each user will always work within her/his own home folder and its subfolders. When UM creates a Shsh process for a user with userid = *uid*, it also creates a subdirectory "***uid*.dir**" as her/his home folder (U3).

A user's Shsh process will have the same current working directory (cwd) as the cwd of the UM at the Shsh process creation time. Thus, before handling any shshcmd, the Shsh process needs to change its cwd to the home folder of the user using the **chdir**() system call (S2). Thus, upon creating a user's Shsh process, the UM should send user's **userid** (integer) and the **home folder path** of the user (string) to the user's Shsh process (U5) via the corresponding pipe and the Shsh process should read the information from the pipe (S1). Just like Unix login, the user starts from her/his home folder. But unlike Unix, the user will not work beyond her/his home folder.

After all the initial steps, Shsh should read commands from its input pipe, executes them like before, and writes the response to its output pipe. To ensure that the output of the "cmd" commands and the last shcmd in a pipe command) are delivered to the output pipe of the Shsh process, we need to map stdout of the Shsh process to the output pipe (dup2). Similarly, we need to map stdin of the Shsh to the input pipe in order for a

command that requires inputs from stdin (like running ./a.out) to properly get its inputs. These have to be done before any "system()" system call is invoked (S3).

When getting the logout command, UM translates it to "exit" and sends it to the corresponding Shsh process for termination. You need to close the corresponding pipes accordingly. Also, your Shsh printouts should now be "**Shsh process (*userid*, *pid*) has been created by process *ppid***" and "**Shsh process (*userid*, *pid*) terminates**". The printouts defined in the previous phase regarding each shshcmd received and processes and pipes created for each pipe command stay the same. Note that the printouts mentioned above are system information and should be printed out by Shsh, only appear on the monitor, not to be sent to UM and printed in the user file. All other outputs (related to the execution of shshcmd, besides those within the pipe command) should be delivered to UM and printed to the user file by UM.

We assume correct user behaviors. Each user (same *userid*) will not make multiple logins before logging out. The commands issued by the users are correct and conforming to the required format. No commands with non-explicit interactions (e.g., text editors) will be issued.

3 Project 3 Phase 1: User-UM Communication via Sockets

In Project 2, we have some "not very good" designs. The most important one is that multiple Shsh processes are supposed to work concurrently for processing the commands of multiple users. In Project 2, the UM will only take one command at a time, i.e., only one Shsh process will be working at a time. This is modified in Project 3.

In this project, we will let users send their commands via sockets to the UM. Thus, each user can input their commands from their own windows. Also, instead of writing the output for each user to a user file, we will pass the output to the user process to be displayed on the user's window. In this case, UM will be a dispatcher (a common server-client design, like web servers, file servers, etc.) for forwarding messages between users and their Shsh processes.

3.1 The UM Process

The UM should create a **server socket** to **listen** for the connection requests from the users. The server socket port number should be the same as the one used by the user's client socket. When starting the UM, we will provide the **port number** as a command line input. Note: If two students use the same port number on the same host, there will be conflict and may not be easy to detect. You need to watch out for this potential problem. We recommend that each student uses **<1..4>+ <last 4 digits of your student id>** to avoid most of the potential conflicts. Also, please check the return values of socket related calls to make sure that server socket initiation is successful. Also note that the UM needs to create its server socket before the users submitting their commands to avoid having failed connection attempts. You can achieve this sync manually by starting the user processes only after UM prints out a message "**UM server socket ready**".

When the server socket of the UM receives a connection request from a user process, it accepts the connection and the **accept**() system call would return a new socket. This returned socket shall be used for subsequent communications between UM and this specific user. The UM, after accepting the connection, will create a new Shsh process for the user (like before). The UM should also **create a new thread** to dedicatedly handle all the communications related to one user. You can use **pthread_create**() to create thread. Within each thread, the activities are the same as those of the original UM. With this design, we no longer need to attach the userid to each command (login still needs userid), but we will just leave it as an artifact. Upon user logout, close all the corresponding pipes, sockets, and files, and terminate the thread.

When UM creates a thread for a user, UM should print out a message: "**New user *userid* logs in, thread created, socket *snum* with port *port#* is used**". When a user logs out, UM also prints a message: "**User *userid* logs out, socket *snum* has been closed, thread is terminating**". The socket number (*snum*) is the file descriptor for the socket.

A Shsh process, upon creation or termination, will print the same message as before.

3.2 The User Process

As can be seen, we now need a **user** process to handle socket communication (with **user.exe** as the executable name). When starting a user process, some command line inputs should be provided and they are specified below. Besides the *userid*, it also needs to know the host name of the UM and the port number of the server socket of the UM.

user.exe *userid* UM-host-name UM-port-number

In Project 2, UM prompts and reads user commands. Now, in UM, you need to remove its user command prompt and reads commands from its server socket. Instead, the user process prompts "ucmd> " and reads user commands. There will be no login command from the user and the user process, upon starting, sends the login command to UM automatically. When it receives the logout command from the user, the process terminates. We also allow the UM to send the logout (termination) request to the user (same command format as the user logout command) to ensure graceful termination upon unusual situations.

The user does not need to attach *userid* to each command keyed in. The user process automatically adds it to the commands before sending them to UM. It also reads responses from UM (indirectly from Shsh) via socket and prints the outputs on user's window.

When the user process finishes setting up its socket (after connecting), a message "**User *userid* socket *snum* with port number *port#* has been created**" should be printed. Before the user process terminates, a message "**User *userid* socket *snum* has been closed, process terminating**" should be printed.

4 Project 3 Phase 2: Interrupt Handling

To let you get hands on experience with interrupt handling, we use interrupts to deliver Administrator (Admin) commands to UM. In Unix, we use **signal**() system call to associate an interrupt bit (of the interrupt vector) to a signal handler function and **kill**() system call to send a signal (an interrupt). The list of Admin commands, the interrupt bit (signal) each command uses, and the corresponding actions for each command are defined in the table below.

command	signal	system actions
terminate	SIGQUIT (^Y)	Gracefully terminates UM and other processes.
sleep	SIGINT (^C)	Sleep for *T* seconds.
infor	SIGRTMAX	Print UM information.
listuser	SIGRTMIN	Print the list of active users and their information.

After receiving the sleep command, the signal handler should first print a notification message "**UM receives the sleep signal, going to sleep for *T* seconds**" and then goes to sleep. Note that we only require the main thread (original) to sleep, not the threads for users. The **sleep time *T*** should be given as the last command line input for the UM. Together with the **port number of the server socket** (briefly, port#) discussed earlier, the UM starts by

UM.exe port# *T*

In UM, you need to maintain a list of users who are still active (who has logged in, not yet logged out). For each user, you need to maintain its **userid**, **pid of user's Shsh process**, the **socket number** of the socket for the UM thread to communicate with Shsh, and the **port number** of the socket. In case it is necessary to terminate the system (e.g., your system seems to run into an infinite loop), Admin can issue the terminate command to UM via SIGQUIT signal. If we only terminate UM, we will need to manually terminate all the user processes and users' Shsh processes. Thus, before terminating UM, UM **kill**s all Shsh processes and **close**s all the Shsh **sockets**. The user processes may be remote, thus, we use the **logout** command (sent from UM) to terminate all the user processes. After all users are informed of the termination, UM **close**s its **server**

socket. Finally, UM can terminate itself from the signal handler by designing your own program logic. Before actual termination, UM should print "**UM terminated on Admin request with *x* active users**", where x is the number of active users.

In 32-bit systems, Unix reserved two bits as user defined interrupt bits. To allow flexibility in the mix existences of 32-bit and 64-bit systems, new Linux systems define SIGRTMIN and SIGRTMAX to be the minimal and maximal bit indices of user defined interrupt bits. We use them for issuing Admin requests for UM information. When printing UM information, you need to print UM's pid, the port number of its server socket, and the number of active users. When printing the list of users, you need to print one user per line, including the **userid**, the **pid of** the user's **Shsh** process, the **socket number** used to interact with Shsh.

You need to write the Admin program which reads in the Admin commands from the administrator and delivers the corresponding signals to UM. The executable for Admin should be "**admin.exe**" and it should be started with the UM's pid as the command line input.

5 Project Submission

You need to submit your program **before** midnight (11:59pm) of the due date (check the web page for the due date). Use UTD elearning system for submission.

Your submission should include the following:

- ✓ All source code files making up your solutions to this assignment.
  - ✓ For Project 2, you have to have separate .c files for Shsh and UM. Best is to have "**shsh.c**" and "**UM.c**". In any case, specify your file names for Shsh and for UM in your *README* file.
  - ✓ For Project 3, you have to have separate .c files for Shsh, UM, user, and admin. Best is to have "**shsh.c**", "**UM.c**", "**user.c**", and "**admin.c**". In any case, specify your file names for each component in your *README* file.
- ✓ The *Makefile* that generates the executables from your source code files. Your Makefile should generate 3 executables "**UM.exe**", "**user.exe**", "**admin.exe**".
  - ✓ For Project 2, only submit the first executable "UM.exe".
  - ✓ If you only submit Phase 1 of Project 2, you should submit "**shsh.exe**".
- ✓ If you did not finish the project, you need to specify which phase you have completed in your *README* file.
- ✓ The *DesignDoc* file, which should contain the pseudo code for UM and Shsh (please do this before you start implementing them so that you can have a clear idea when proceeding to coding) and the description of the major features of your program that is not specified in this handout.