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 $\boldsymbol{userid}$ logs in, thread created, socket $\boldsymbol{snum}$ with port $\boldsymbol{port}$# is used”. When a user logs out, UM also prints a message: “User $\boldsymbol{userid}$ logs out, socket $\boldsymbol{snum}$ 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 $\boldsymbol{userid}$ 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 $\boldsymbol{userid}$ socket $\boldsymbol{snum}$ with port number $\boldsymbol{port}$# has been created" should be printed. Before the user process terminates, a message "User $\boldsymbol{userid}$ socket $\boldsymbol{snum}$ has been closed, process terminating" should be printed.

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 $\boldsymbol{T}$ 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 $\boldsymbol{T}$ 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# $\boldsymbol{T}$

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 kills all Shsh processes and closes 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 closes 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 $\boldsymbol{x}$ 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.