### CS333: Operating Systems Lab

# Lab 3: Building a Shell

## Goal

In this assignment, we will understand processes and signals by building a custom shell-based client for the file server of Lab 2.

## Before you start

- Read about and understand the structure of a typical shell: read an input string from the user, `fork` a child, `exec` a command executable, `wait` for the child, and repeat the process. Read about all variants of the `wait` and `exec` system calls - there are several of them, and you need to pick different variants under different circumstances.
- Understand the basic structure of any shell code, e.g., the xv6 shell code.
- Read the complete lab, especially the tips and guidelines towards the end of the lab carefully, to avoid spending too much time on some tricky parts of the lab.

## Helper files

Before you begin solving the lab, you must write some helper files, as instructed below.

- Modify the server from Lab 2 to write a program called `server-slow.c`. This program will simply sleep for 1 second after every write into the socket. This exercise may not make much sense to you now, but it will be very useful for your debugging the lab. For many tasks in this lab, you will need your file download to last a very long time, so that you can perform some interesting tests (like signal handling) during the download. With your slow server, you can make downloads of even 2MB files to last a long time, so that you can easily debug.
- Next, write a simple standalone client for the file server. Write a program called `get-one-file.c` that takes four arguments: the file name to fetch, the server IP address, server port number, and a variable to indicate whether the client should display the downloaded file contents on stdout or not. Your client can be a simple single threaded client that just sends one request to fetch one file and exits. (You should have already written something like this during your initial phases of building the client in Lab 2.) You may test it with the server of Lab 2 (or your slow server above). Below are shown two invocations of the client program to get a file from a server running on the local host at port 5000. You can see the use of the two options: `display` and `nodisplay`.

```
$./get-one-file files/foo100.txt 127.0.0.1 5000 nodisplay
$./get-one-file files/foo100.txt 127.0.0.1 5000 display
... < downloaded file contents displayed to standard output> ...
```

- Next, write a signal handler to handle the SIGINT (Ctrl+C) signal in your client. Normally, when you type Ctrl+C, the default signal handler provided by the OS will cause the `get-one-file` process to terminate without printing anything. You must now write `get-one-file-sig.c` that has a signal handler to handle SIGINT. This signal handler should print out how many bytes have been downloaded before terminating the process, as shown below. (Note how running with a slow server is useful to debug this client, as you will have ample time to hit Ctrl+C and observe the effect.)

```
$./get-one-file-sig files/foo100.txt 127.0.0.1 5000 nodisplay
Received SIGINT; downloaded 52037 bytes so far.
```

## A shell-based client for the file server

We now start the most interesting part of this lab. You will write a program called `client-shell.c`. When run, `client-shell` should ask the user for input commands, and execute them. Most commands will require the shell to `fork` one or more child processes, and `exec` some executable in the spawned processes. Use the string "Hello>" as your command prompt.

### Commands

Below are the commands you must implement in your shell, and a description of what the shell must do for those commands.

- `cd` *`directoryname`* must cause the shell process to change its working directory. This command should take one and only one argument; an incorrect number of arguments (e.g., commands such as `cd`, `cd dir1 dir2` etc.) should print an error in the shell. `cd` should also return an error if the change directory command cannot be executed (e.g., because the directory does not exist). *For this, and all commands below, incorrect number of arguments or incorrect command format should print an error in the shell. After such errors are printed by the shell, the shell should not crash. It must simply move on and prompt the user for the next command.*

- `server` *`server-IP server-port`* should store the server IP and port details for future file downloads. There is no need for the shell to connect and check that the server is up and running. An incorrect number of arguments must print an error.

- `getfl` *`filename`* should download the file from the remote server and print its contents to the standard output. Note that this, and all future file download commands below, should have had a server IP address and port specified by the `server` command apriori; otherwise an error must be printed. The shell should prompt the user for the next command only after the file download completes.

  Specifying an incorrect number of arguments (e.g., no filename or more than one filenames) should cause the shell to print an error and prompt for the next command. For this, and all file download

commands below, any errors that may be generated during the file download itself (e.g., socket connection to the server failed) should also be printed out to the shell.

- `getfl` *`filename`* `>` *`outputfile`*, is similar to `getfl` above, except that it should download the file and store it in the specified filename, without printing anything to standard output. Usage of output redirection using $>$ with any other string format (e.g., specifying no or more than one filenames after $>$) should print an error.

- `getfl` *`filename`* | *`command`*, is similar to `getfl` above, except it should cause the downloaded file contents to be piped to some built-in Linux command (like `grep`), and print whatever output comes out of the Linux command. The original file contents must not be printed to the shell. You may assume that *command* is a simple Linux command without any complications like pipes and redirections. Any errors generated during the execution of this *command* should be printed in the shell. Usage of pipes with any other string format should print an error.

- `getsq` *`file1 file2 file3`* `...` should download the multiple files sequentially, one after the other. The next download should start only after the previous one finishes. The file contents should not be printed to the shell, and can be discarded after download. The shell should ask for the next command only after all downloads complete. At least one file name must be provided to `getsq`; otherwise the shell must print an error.

- `getpl` *`file1 file2 file3`* `...` should download the multiple files in parallel, simultaneously. The next file download can, and should, start without waiting for the previous one to finish. The file contents should not be printed to the shell, and can be discarded after download. The shell should ask for the next command only after all downloads complete. At least one file name must be provided to `getpl`; otherwise the shell must print an error.

- `getbg` *`filename`* should download the file in the background. Note that all of the earlier commands download the file (or files) in the foreground, and the shell does not ask for the next command until all downloads finish. However, when you run this command to download a file in the background, the shell should start the download, and come back to prompt the user for the next command. When the download completes in the background at a future time, the shell should print a message that the background process completed. This command should be given one and only one filename; any other format must print an error.

- All simple built-in commands of Linux (like `ls`, `cat`, `echo`) etc. should be supported. There is no need to support I/O redirection or pipes or any such features in the simple Linux commands. Any errors returned during the execution of these commands must be displayed in the shell.

- `exit` should cause the shell to terminate, along with all child processes it has spawned.

- For any other command that doesn't match what is specified above, the shell should print an error and move on to prompt the user for the next command.

**NOTE:** All the file download related commands above MUST execute the `get-one-file-sig` executable you wrote earlier in a new child process, with suitable arguments. You must NOT reimplement the file download functionality (opening a socket etc.) as part of this shell code. You may assume that the `get-one-file-sig` executable is in the same directory as your shell (if it makes your life simpler).

### Signal handling and reaping child processes

The handling of the SIGINT (Ctrl+C) signal by the shell should be as follows. While the shell is running a command in the foreground (e.g., any of `getfl`, `getsq`, `getpl`, or any Linux built-in command, typing Ctrl+C should cause all the running foreground child processes to terminate, but not the shell itself. For the `getsq` command, the shell must terminate the current ongoing download, not start any other downloads, and return for user input. For the `getsq` command, all ongoing parallel downloads must terminate. Hitting Ctrl+C should not terminate any running background processes started by `getbg`. Typing the `exit` command should cause all background child processes to receive the SIGINT signal and terminate; the shell itself must terminate after cleaning up all state, dynamic memory etc. Further, all file downloads (foreground or background) that are terminated by SIGINT must print out how many bytes they have downloaded so far before terminating (much like you do in `get-one-file-sig`).

The shell must also carefully reap all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input. For commands that create background child processes, the shell must periodically check and reap any terminated background processes, while running other commands. When the shell reaps a terminated background process at a future time, it must print a message to let the user know that a background process has finished. By carefully reaping all children (foreground and background), the shell must ensure that it does not leave behind any zombies or orphans when it exits.

## Tips and guidelines

- You are given a sample code `make-tokens.c` that takes a string of input, and "tokenizes" it (i.e., separates it into space-separated commands). You may find it useful to split the user's input string into individual commands.
- You may assume that the input command has no more than 1024 characters, and no more than 64 "tokens". Further, you may assume that each token is no longer than 64 characters.
- You will find the `server-slow` program useful for debugging. You may want to test your file download commands with `server-slow`, instead of the regular server of Lab 2. A slower server will make downloads last longer, so that you have enough time to hit Ctrl+C during the execution of your command, and observe its effect on your child processes. We will use a slow server in our tests as well.
- You may want to build a simple shell that runs simple Linux built-in commands first, before you go on to tackle the more complicated cases.
- You will find the `dup` system call and its variants useful in implementing I/O redirection and pipes. When you `dup` a file descriptor, be careful to close all unused, extra copies of the file descriptor. Also recall that child processes will inherit copies of the parent file descriptors, so be careful and close extra copies of inherited file descriptors also. Otherwise, your I/O redirection and pipe implementations will not work correctly.
- You must catch the SIGINT signal in your shell and handle it correctly, so that your shell does not termiante on a Ctrl+C, but only on receiving the exit command.
- When you send a SIGINT signal to a process, all child processes will automatically receive the signal. However, this will not work for us, because we want to send SIGINT selectively only

to the foreground processes (recall that the background processes must terminate when the shell finally exits, not on a Ctrl+C). One way to overcome this problem is to place your child processes in a separate process group (check out the `setpgid` system call), so that they do not receive the SIGINT sent to the parent automatically. The parent shell process must then manually send the SIGINT (using the `kill` system call) to those child processes that it wants to terminate.

- You will find the `chdir` system call useful to implement the `cd` command.
- You may assume that no more than 64 files are requested at a time in the `getsq` and `getpl` commands. Further, you may assume that no more than 64 background processes or 64 foreground processes will exist at any time.
- Carefully handle all error cases listed above for each command. For example, an incorrect command string should always print an error.

## Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- Your shell code `client-shell.c`. Comment your code so that it is readable.
- Your helper code `server-slow.c`, `get-one-file.c`, `get-one-file-sig.c`.
- An optional makefile to build your code.
- `testcases.txt` showing results for some sample runs of your code.

Evaluation of this lab will be as follows.

- We will test this lab primarily by running your shell-based client with our server. We will test that you have correctly implemented all commands listed above, including handling of error cases listed with each command. We will test that your code implements signal handling correctly, by running with a slow server and hitting Ctrl+C on several download commands (foreground, background etc.). We will log download requests at server, so we will make sure downloads happening in sequence or in parallel as specified. We will also monitor status of child processes and ensure they are being cleaned up correctly.
- Finally, we will also read through your code to check for clarity and correctness.