M1522.000800 System Programming
Fall 2019

System Programming Shell Lab Report

Jinje Han
2018-18786

1. Introduction

The purpose of this lab is simple. Write your own shell code. The name of the shell to create is called tsh. Various test inputs are provided and can be easily entered to check that the required functions are properly implemented.

2. Important Concepts

There are a few functions you need to know to program the features of the shell. ① fork() divides current process into child process and parent process, returning twice. The child processs returns 0, and the parent process returns the pid of that child process. ② execve() takes an executable object file, arguments, and environment variables and executes the file. This function doesn't return and immediately exits. ③ waitpid() waits for the termination of a process and them resumes current proces. Depending on the argument, you can decide which process to wait for. If -1 is entered as the first argument, it waits for the termination of any child process. ④ setpgid() is a function that puts a process into a process group. setpgid(0, 0) creates a process group with the pid of the process that executed this command, and puts the current process into that group. ⑤ sigprocmask() is a function that can change which signal is blocked in the current process. If you put a set of signals as argument, you can add, remove, or replace them.

One commandline is divided into a built-in command and others. Built-in commands contain certain keywords, so if a command is included in it, run it immediately. Otherwise, the first word of the commandline is the path of the program to be executed and additional words are arguments.

Various functions were implemented to focus only on shell implementation. Below are descriptions of the functions I needed to implement and how I did it. I have omitted references to messages that should be printed.

3. eval()

This function takes a single commandline and executes the corresponding command. First of all, we need to parse the commandline and break it into words, which is already implemented in parseline(). In addition, parseline() returns whether the current input operation should be executed in the foreground or in the background.

Next, put the parsed command array into builtin_cmd(). If the return value is 1, this commandline is a built-on command, meaning that the command has already been executed in that function. If not, now we need to execute the command. If you just use execve() function without forking, the current tsh program will terminated after execution, so you have to create a child process with fork() and execute a in it. The parent process receives SIGCHLD signal generated when the child process terminates, and the appropriate processing is performed by the sigchld_handler(). However, when the child process terminates before any of the parent process's commands are executed, this process can be twisted(ex: It can be skipped before it is placed in the jobs[] array). Therefore, block SIGCHLD using sigprocmask() before fork(), and execute the commands that must be executed before the end of the child process in the parent process. The child process doesn't need this, so you can just free the blocking. Also, tsh's process group needs to adjust the group ID of the child

process so that tsh is one process in its process group. You can separate the process group by using setpgid(0, 0) in the child process.

The only difference between the foreground and the background process is whether the main program(tsh) waits for the process to finish. So we put the fork() return value(=pid of the child process) in waitfg() only to wait in the foreground.

I thought that the first word of the parsed commandline(=the location of the executable file) should be substracted from the the second argument of execve(). However, it seemed that the the function automatically does it for me. Also, I didn't need to set environment variables, so I used execvp() instead of execve().

If an invalid command is encountered, an error message should be displayed and the process should be terminated. If terminated with return() function, the fork() process will return from execvp(), resulting with two tsh programs. Therefore, the invalud command must terminate with exit().

If file redirection is required, the file name is entered in the file[] array that is passed as the third argument to parseline(). In this case, all output to stdout should be directed to this file via dup2() function. The problem is that in this case, the file descriptor of stdout is closed. So we need to copy stdout to dup() and then return all stdout back to dup2() after all the processing is done.

4. builtin_cmd(), do_bgfg()

builtin_cmd () is a function that handles when a built-in command is entered in the commandline. This function takes an array of parsed commandline, matches the first word with built-in commands one by one, and executes the command if it exists returning 1 or just returns 0. In this way, eval() executes builtin_cmd(), and if this function returned 1, then the command processing would be terminated.

If you type 'quit', this is the command to exit the shell, so execute exit(0) to terminate the process itself. If you enter 'jobs', this is a command that lists all the current background jobs, so you can run listjobs(), which is already implemented to do the same. This function also lists the foreground, but if you run this jobs command, it is guaranteed that there is no foreground process already, so running it is fine. 'fg' and 'bg' commandes are passed to do_bgfg() for execution.

do_bgfg is a function that handles only bg and fg of built-in commands. First, both bg and fg takes the pid or jid of a process as an argument. If an Argument starts with '%', it is a jid, so you can get the corresponding job_t struct and pid using getjobjid (). If the argument just starts with a number, it is a pid, so you can get the corresponding job_struct through getjobpid(). Now that we know the pid of the job we want to process, we only need to process it depending on whether the command is fg or bg. fg is the process of bringing a stopped or running process to the foreground. First of all, this process may have stopped, so kill sends SIGCONT to resume process execution. Next, bringing the process to the foreground means waiting for the process to finish (running in the background for the process, but running in the foreground, but no difference). You can wait for the termination by putting pid in waitpid(). bg keeps a stopped background process running in the background. In this case, just send SIGCONT. In addition, both of these commands need to change state in the jobs [] array.

5. waitfg()

This function takes a pid and waits for the process corresponding to that pid to terminate in the foreground. The pause() function can be used to resume execution as soon as the process terminates, but for a simple and intuitive implementation, the method checks whether or not it terminates every second. Receive the pid of the process currently running in the foreground through fgpid() in an infinite loop of the while() statement containing sleep (1). Exit the loop with a break statement if this received pid through fgpid() is different from the pid received as the argument. That's the way. If no process is running in the foreground, fgpid() returns 0, so you can escape the loop as well.

6. Signal Handlers

sigint_handler () handles tasks that change the state of the jobs [] array when the child process terminates and becomes a zombie process or stops. The zombie child pid can be found by adjusting the arguments to the waitpid () function. waitpid () includes the WNOHANG option because there is a zombie but should not wait for the report to terminate, and the WUNTRACED option because the stopped child must also receive it. First of all, when child is finished, just execute deletejob (). If the child receives SIGSTOP or SIGTSTP and stops, it has to handle different things accordingly. This can be seen by processing the status supplied as the second argument of waitpid (). WIFSTOPPED (status) is 1 for SIGSTOP and WIFSIGNALED (status) is 1 for SIGTSTP. This is not handled by other handlers, because status must also be printed.

In sigint_handler (), you need to write an action that should be done when the user presses ctrl + c. Since you can pass SIGINT to a job in the foreground, you can get the pid of the process currently in the foreground through fgpid (), and then pass SIGINT to kill () on the group. sigstp_handler () plays a similar role, with the only difference being that you should pass SIGSTP to the foreground task when the user presses ctrl + z.

7. Conclusion

I thought the shell would work something incredibly complex, but it was simpler than I thought. I think it was because there are many functions that are implemented in advance. A lot of hints were given in handout and it helped me a lot. After implementing as a hint and finding why I should do like that also became a lot of study.