Lecture Notes on Operating Systems

Lab: Building a Shell

In this lab, you will build a simple shell to execute user commands, much like the `bash` shell in Linux. This lab will deepen your understanding of various concepts of process management in Linux.

Before you begin

- Familiarize yourself with the various process related system calls in Linux: `fork`, `exec`, `exit` and `wait`.The "man pages" in Linux are a good source of learning. You can access the man pages from the Linux terminal by typing `man fork`, `man 2 fork` and so on. You can also find several helpful links online (e.g., `http://manpages.ubuntu.com/manpages/trusty/man2/fork.2.html`).
- Write simple programs using these system calls, as a warmup to solving this lab. For example, you can write a program that forks a child process. The child process should print some message and exit. The parent should wait to reap the child, print a message after reaping the child, and then exit. Write many such simple programs until you feel confident with the basic usage of these system calls.
- It is important to understand the different variants of these system calls. In particular, there are many different variants of the `exec` and `wait` system calls; you need to understand these to use the correct variant in your code. For example, you may need to invoke wait differently depending on whether you need to block for a child to terminate or not.
- Familiarize yourself with simple built-in commands in Linux like `echo`, `cat`, `sleep`, `ls`, `ps`, `top`, `grep` and so on. To implement these commands in your shell, you must simply "exec" these built-in executables, and not implement the functionality yourself.
- Understand the `chdir` system call in Linux (see `man chdir`). This will be useful to implement the `cd` command in your shell.
- For part B, understand the concepts of foreground and background execution in Linux. Execute various commands on the Linux shell both in foreground and background, to understand the behavior of these modes of execution.
- For parts C and D, understand signals and signal handling in Linux. Understand how processes can send signals to one another using the `kill` system call. Read up on how to write custom signal handlers to "catch" signals and override the default signal handling mechanism, using interfaces such as `signal()` or `sigaction()`.

- For part D, understand the notion of processes and process groups. Every process belongs to a process group by default. When a parent forks a child, the child also belongs to the process group of the parent initially. When a signal like Ctrl+C is sent to a process, it is delivered to all processes in its process group, including all its children. If you do not want some subset of children receiving a signal, you may place these children in a separate process group, say, by using the `setpgid` system call. Lookup this system call in the man pages to learn more about how to use it, but here is a simple description. The `setpgid` call takes two arguments: the PID of the process and the process group ID to move to. If either of these arguments is set to 0, they are substituted by the PID of the process instead. That is, if a process calls `setpgid(0,0)`, it is placed in a separate process group from its parent, whose process group ID is equal to its PID. Understand such mechanisms to change the process group of a process.

- Read the problem statement fully, and build your shell incrementally, part by part. Test each part thoroughly before adding more code to your shell for the next part.

Part A: A simple shell

We will first build a simple shell to mostly run Linux built-in commands. A shell takes in user input, forks one or more child processes using the `fork` system call, calls `exec` from these children to execute user commands, and reaps the dead children using the `wait` system call. Your shell must execute *all* simple Linux commands like `ls`, `cat`, `echo` and `sleep`. These commands are readily available as executables on Linux, and your shell must simply invoke the existing executable. It is important to note that you must implement the shell functionality yourself, using the fork, exec, and wait system calls. You must not use library functions like `system` which implement shell commands by invoking the Linux shell—doing so defeats the purpose of this assignment!

Your simple shell must use the string "$ " as the command prompt. Your shell must run in one of two modes: interactive or batch. If no command-line argument is provided, your shell should interactively accept inputs from the user and execute them. If a batch file of commands is provided as command-line input to your program, then your shell must execute all commands in the batch file one after the other. You are provided a sample batch file `commands.txt` as an example; however note that you will need to handle all built-in Linux commands, not just those provided as examples in this file.

In this part, the shell must return for user input (or move on to the next command in the batch) only after the execution of the previous command completes. Further, in this part, a shell in interactive mode should continue execution indefinitely until the user hits Ctrl+C to terminate the shell. In batch mode, the shell must exit once it reaches the end of the batch file.

You can assume that the command to run and its arguments are separated by one or more spaces in the input, so that you can "tokenize" the input stream using spaces as the delimiters. For this part, you can assume that the Linux built-in commands are invoked with simple command-line arguments, and without any special modes of execution like background execution, I/O redirection, or pipes. You need not parse any other special characters in the input stream. *Please do not worry about corner cases or overly complicated command inputs for now; just focus on getting the basics right.*

Note that it is not easy to identify if the user has provided incorrect options to the Linux command (unless you can check all possible options of all commands), so you need not worry about checking the arguments to the command, or whether the command exists or not. Your job is to simple invoke `exec` on any command that the user gives as input. If the Linux command execution fails due to incorrect command or arguments, an error message must be printed on screen (by your code or by the executable)

and your shell must move on to the next command. That is, if the command itself does not exist, then the exec system call will fail, and you will need to print an error message on screen, and move on to the next command. On the other hand, if the command exists but the arguments are incorrect, your exec system call will succeed, but the executable will print an error message and terminate, without you having to print an error yourself. In either case, errors must be suitably notified to the user, and you must move on to the next command.

A skeleton code `my_shell.c` is provided to get you started. This program reads input (interactively or from a batch) and tokenizes the input for you. You must add code to this file to execute the commands found in the "tokens". 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 can compile and run this code in two ways as shown below.

- `./my_shell` will run the program in interactive mode.
- `./my_shell commands.txt` will run the program in batch mode.

Once you complete the execution of the built-in commands, proceed to implement support for the simple `cd` command in your shell using the `chdir` system call. The command `cd <directoryname>` must cause the shell process to change its working directory, and `cd ..` should take you to the parent directory. You need not support other variants of `cd` that are available in the various Linux shells. For example, just typing `cd` will take you to your home directory in some shells; you need not support such complex features. Note that you must NOT spawn a separate child process to execute the `chdir` system call, but must call `chdir` from your shell itself, because calling `chdir` from the child will change the current working directory of the child whereas we wish to change the working directory of the main parent shell itself. Any incorrect format of the `cd` command should result in your shell printing `Shell: Incorrect command` to the display and prompting for the next command.

Your shell must gracefully handle errors. An empty command (typing return) should simply cause the shell to display a prompt again without any error messages. For all incorrect commands or any other erroneous input, the shell itself should not crash. It must simply notify the error and move on to prompt the user for the next command.

For all commands, you must take care to terminate and carefully reap any child process the shell may have spawned. Please verify this property using the `ps` command during testing. When the forked child calls `exec` to execute a command, the child automatically terminates after the executable completes. However, if the `exec` system call did not succeed for some reason, the shell must ensure that the child is terminated suitably. When not running any command, there should only be the one main shell process running in your system, and no other children.

To test this lab, run a few common Linux built-in commands in your shell, and check that the output matches what you would get on a regular Linux shell. Further, check that your shell is correctly reaping dead children, so that there are no extra zombie processes left behind in the system.

Part B: Serial, parallel, and background execution

Now, we will build support for executing multiple commands at a time in your shell, as described below. Extend your shell program of part A to support the following modes of operation.

- If a command is followed by &, the command must be executed in the background. That is, the shell must start the execution of the command, and return to prompt the user for the next input,

without waiting for the previous command to complete. The output of the command can get printed to the shell as and when it appears.

- Multiple user commands separated by && should be executed one after the other in sequence in the foreground. The shell must move on to the next command in the sequence only after the previous one has completed (successfully, or with errors). The shell should return to the command prompt after all the commands in the sequence have finished execution.
- Multiple commands separated by &&& should be executed in parallel in the foreground. That is, the shell should start execution of all commands simultaneously, and return to command prompt after all commands have finished execution.
- A command not followed by any of the special characters mentioned above must simply execute in the foreground as before.

In all the cases above, you may assume that each of the individual commands are simple Linux built-in commands without pipes or redirections or any other special cases. Also assume that a single input line to the shell will only correspond to one of the modes (single foreground command, single background command, multiple commands in series or in parallel), and not a combination of modes. You may also assume that there are spaces on either side of the special tokens like &, &&, and &&&. You may assume that there are no more than 64 foreground or background commands executing at any given time. *Once again, please focus on getting the basic common cases to work correctly, before beginning to worry about corner cases.*

A helpful tip for testing: use multiple long running commands like `sleep` to test your series, parallel, or background implementations, as such commands will give you enough time to run `ps` in another window to check that the commands as executing as specified. You should be able to run one or more sleep commands in background/series/parallel modes to verify the correctness of these implementations.

Across all cases, carefully ensure that the shell reaps 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 the command that creates background child processes, the shell must periodically check and reap any terminated background processes while running other commands. For example, the shell may check for dead children and reap them every time it obtains a new user input from the terminal. A time delay is reaping background children (e.g., until the user types the next command) is completely acceptable. When the shell reaps a terminated background process at a future time, it must print a message `Shell: Background process finished` to let the user know that a background process has finished.

You must also test your implementation for the cases where background and foreground processes are running together, and ensure that all of them are executing as expected. In particular, care should be exercised in reaping children in the correct order in such cases, because a generic `wait` system call can return any dead child to you. So you must take care to identify which child you have reaped. For example, if you are waiting for a foreground process in a series to terminate and `wait` reaps a background process, you must not erroneously run the next command in the series, but you must wait for the foreground command to terminate as well. To avoid such confusions, you may choose to use the `waitpid` variant of this system call, to be sure that you are reaping the right child. Once again, use long running `sleep`-like commands, run `ps` in another window, and monitor the execution of your processes to test cases with multiple background and foreground processes. In particular, test that a background process finishing up in the middle of a series of commands will not cause your shell to incorrectly move forward to the next command in the series before the previous one finishes.

Part C: The exit command

Up until now, your shell executes in an infinite loop, and only the signal SIGINT (Ctrl+C) would have caused it to terminate. Now, you must implement the `exit` command that will cause the shell to terminate its infinite loop and exit. When the shell receives the `exit` command, it must terminate all background processes, say, by sending them a signal via the `kill` system call. Obviously, if the shell is receiving the comand to exit, it goes without saying that it will not have any active foreground processes running. Before exiting, the shell must also clean up any internal state (e.g., free dynamically allocated memory), and terminate in a clean manner.

Part D: Handling the Ctrl+C signal

Up until now, the Ctrl+C command would have caused your shell (and all its children) to terminate. Now, you will modify your shell so that the signal SIGINT does not terminate the shell itself, but only terminates the one or more foreground processes it is running. That is, when executing multiple commands in serial mode, the shell must terminate the current command, ignore all subsequent commands in the series, and return back to the command prompt. When in parallel mode, the shell must terminate all foreground commands and return to the command prompt. Note that the background processes should remain unaffected by the SIGINT, and must only terminate on the `exit` command. You will accomplish this functionality by writing custom signal handling code in the shell, that catches the Ctrl+C signal and relays it to the relevant processes, without terminating itself.

Hint: Recall that, by default, any signal like SIGINT will be delivered to the shell and all its children. To solve this part correctly, you must carefully place the various children of the shell in different process groups, say, using the `setpgid` system call. Note that `setpgid(0,0)` places a process in its own separate process group, that is different from the default process group of its parent. Your shell must do some such manipulation on the process group of its children to ensure that only a subset of the children receive the Ctrl+C signal. For example, placing all the background children in a separate process group will ensure that they do not get killed by the Ctrl+C signal immediately.

Once again, use long running commands like `sleep` to test your implementation of Ctrl+C. You may start multiple long running background processes, start series or parallel foreground processes, hit Ctrl+C, and check that only the foreground process(es) are terminated and none of the background processes are terminated.

Submission instructions

- You must submit the shell code `my_shell.c` or `my_shell.cpp`.