Lecture Notes on Operating Systems

Lab: Building a Shell

Part A: A simple shell

We will first build a simple shell, much like the `bash` shell of Linux. 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. Learn about the `fork` system call and all variants of the `wait` and `exec` system calls before you begin this lab.

Begin your code by writing a shell that executes simple Linux commands like `ls`, `cat`, `echo` and `sleep`. These commands are readily available as executables on Linux, and your shell must simply invoke them. Your simple shell must use the string "Hello>" as the command prompt. If the user command is one of the Linux built-in commands, you must `exec` the corresponding Linux executable, and return for user input after execution completes. The shell must continue execution in this manner in an infinite loop, until the user hits Ctrl+C to terminate the main shell process. Any errors returned during the execution of these commands must be displayed in the shell. For now, you may assume that the user input does not include any extra functionalities like background execution, pipes, or I/O redirection.

However, not all commands are built into Linux, and the shell must write code to implement some commands as well, e.g., the command `cd` *`directoryname`* and the command `history`. Once you complete the execution of the built-in commands, proceed to implement support for `cd` and `history` in your shell. The semantics of the commands must be similar to what you find in the `bash` shell. For example, `cd` *`directoryname`* must cause the shell process to change its working directory. You will find the `chdir` system call useful to implement the `cd` command.

Note that for all commands you implement in this lab, an 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.

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. Further, 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.

Part B: Serial, parallel, and background execution

Now, we will build support for executing multiple commands at once in your shell, as described below.

- 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). An error in one command should cause the shell to simply move on to the next command. The shell should return to the command prompt after all the commands 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.

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. 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.

In all the cases above, you may assume that each of the individual commands are simple Linux built-in commands without pipes or redirections. 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.

Part C: Signal handling and exit

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 will modify your shell so that the signal SIGINT does not terminate it. You will also implement the `exit` command that will cause the shell to terminate its infinite loop and exit. When the user hits Ctrl+C, the shell should terminate the current foreground processes (the current command in serial execution, or all the commands in a parallel execution), and return to the command prompt. The background processes should remain unaffected by the SIGINT. When the shell receives the `exit` command, it must terminate all background processes, clean up any internal state (e.g., free dynamically allocated memory), and finally terminate.

Read up on how to write custom signal handlers to "catch" signals and override the default signal handling mechanism in a process. You must catch the SIGINT signal in your shell and handle it correctly, so that your shell does not terminate on a Ctrl+C, but only on receiving the exit command. Further, 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.

Part D: I/O redirection and pipes

Now, add support for I/O redirection and pipes when executing simple built-in Linux commands. Follow the syntax and semantics of the Linux shell as closely as possible.

You will find the `dup` and `pipe` family of system calls 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. Without these precautions, your I/O redirection and pipe implementations will not work correctly.