Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

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`, `man 2 wait` and so on. You can also find several helpful links online.
- 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 shell commands in Linux like `echo`, `cat`, `sleep`, `ls`, `ps`, `top`, `grep` and so on. To run these commands from your shell, you must simply "exec" these existing 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.
- 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.
- Understand the concept of signals and signal handling in Linux. Understand how processes can send signals to one another using the `kill` system call. Read up on the common signals (SIGINT, SIGTERM, SIGKILL, ..), and how to write custom signal handlers to "catch" signals and override the default signal handling mechanism, using interfaces such as `signal()` or `sigaction()`.
- 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.

Warm-up exercises

Below are some "warm-up" exercises you can do before you start implementing the shell.

1. Write a program that forks a child process using the fork system call. The child should print "I am child" and exit. The parent, after forking, must print "I am parent". The parent must also reap the dead child before exiting. Run this program a few times. What is the order in which the statements are printed? How can you ensure that the parent always prints after the child? You may refer to the man page for wait (`man 2 wait`) that has a very detailed example of a simple program using fork and wait.

2. Write a program that forks a child process using the fork system call. The child should print its PID (using the `getpid` system call) and exit. The parent should wait for the child to terminate, reap it, print the PID of the child that it has reaped, and then exit. Explore different variants of the wait system call (`wait`, `waitpid`) while you write this program.

3. Write a program that uses the exec system call to run the "ls" command in the program. First, run the command with no arguments. Then, change your program to provide some arguments to "ls", e.g., "ls -l". In both cases, running your program should produce the same output as that produced by the "ls" command. There are many variants of the exec system call, e.g., `execvp`, `execlp`, and so on. Read through the man pages to find something that suits your needs.

4. Write a program `runcmd.c` that executes a simple Linux command with a single argument, for example, `sleep 10`. Your program should take two commandline arguments: the name of the command and the argument to the command. You can access the strings given as commandline arguments using the variables `argv[0]`, `argv[1]`, and so on. The program must then fork a child process, and exec the command in the child process. The parent process must wait for the child to finish, reap it, and print a message that the command executed successfully. Your program must throw an error if it is given an incorrect number of arguments.

   A sample execution of the program is shown below.

```
$gcc runcmd.c -o runcmd
$./runcmd echo hello
hello
Command successfully completed
$ ./runcmd ls
Incorrect number of arguments
$
```

5. Write a program that uses the exec system call to run some command. Place a print statement just before and just after the exec statements, and observe which of these statements is printed. Understand the behavior of print statement placed after the exec system call statement in your program by giving different arguments to exec. When is this statement printed and when is it not?

6. Write a program where the fork system call is invoked N times, for different values of N. Let each newly spawned child print its PID before it finishes. Predict the number of child processes that will be spawned for each value of N, and verify your prediction by actually running the code. You must also ensure that all the newly created processes are reaped by using the correct number of wait system calls.

7. Write a program where a process forks a child. The child runs for a long time, say, by calling the sleep function. The parent must use the `kill` system call to terminate the child process, reap it, print a message, and exit. Understand how signals work before you write the program.

8. Write a program that runs an infinite while loop. The program should not terminate when it receives SIGINT (Ctrl+C) signal, but instead, it must print "I will run forever" and continue its execution. You must write a signal handler that handles SIGINT in order to achieve this. So, how do you end this program? When you are done playing with this program, you may need to terminate it using the SIGKILL signal.

Part A: A simple shell

We will first build a simple shell to run simple Linux commands. A shell takes in user input, forks a child process using the `fork` system call, calls `exec` from this child to execute the user command, reaps the dead child using the `wait` system call, and goes back to fetch the next user input. Your shell must execute *any* simple Linux command given as input, e.g., `ls`, `cat`, `echo` and `sleep`. These commands are readily available as executables on Linux, and your shell must simply invoke the corresponding executable, using the user input string as argument to the exec system call. 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 should interactively accept inputs from the user and execute them. In this part, the shell should continue execution indefinitely until the user hits Ctrl+C to terminate the shell. 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. You are given starter code `my_shell.c` which reads in user input and "tokenizes" the string for you. The function returns an null-terminated array of strings called `tokens`, where the first element of the array `tokens[0]` is the command to execute, and the rest of the elements in the array are the arguments. For variants of the exec command that need the name of the command along with a null-terminated list of arguments, you should be able to pass the entire `tokens` array easily. 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.

For this part, you can assume that the Linux 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.*

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. 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 simply invoke `exec` on any command that the user gives as input. If the Linux command execution fails due to incorrect arguments, an error message will be printed on screen by the executable. If the command itself does not exist, then the exec system call will fail. In both cases, errors must be suitably notified to the user, child process must be reaped, and the shell must move to the next command.

Once you complete the execution of simple commands, proceed to implement support for the `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 an error

message to the display, and prompting 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 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: Background execution

Now, we will extend the shell to support background execution of processes. Extend your shell program of part A in the following manner: if a Linux 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. A command not followed by & must simply execute in the foreground as before.

You can assume that the commands running in the background are simple Linux commands without pipes or redirections or any other special case handling like `cd`. You can assume that the user will enter only one foreground or background command at a time on the command prompt, and the command and & are separated by a space. You may assume that there are no more than 64 background commands executing at any given time. A helpful tip for testing: use long running commands like `sleep` to test your foreground and background implementations, as such commands will give you enough time to run `ps` in another window to check that the commands are executing as specified.

Across both background and foreground execution, ensure that the shell reaps all its children that have terminated. Unlike in the case of foreground execution, the background processes can be reaped with a time delay. For example, the shell may check for dead children periodically, say, when it obtains a new user input from the terminal. When the shell reaps a terminated background process, it must print a message `Shell: Background process finished` to let the user know that a background process has finished.

You must test your implementation for the cases where background and foreground processes are running together, and ensure that dead children are being reaped correctly in such cases. Recall that a generic `wait` system call can reap and return any dead child. So if you are waiting for a foreground process to terminate and invoke `wait`, it may reap and return a terminated background process. In that case, you must not erroneously return to the command prompt for the next command, 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 correct foreground child. Once again, use long running commands like `sleep`, run `ps` in another window, and monitor the execution of your processes, to thoroughly test your shell with a combination of background and foreground processes. In particular, test that a background process finishing up in the middle of a foreground command execution will not cause your shell to incorrectly return to the command prompt before the foreground command 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 process 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 foreground process it is running. 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.

Note 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. For example, `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 the foreground child receives the Ctrl+C signal, and the background children in a separate process group 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, then start a foreground command, hit Ctrl+C, and check that only the foreground process is terminated and none of the background processes are terminated.

Part E: Serial and parallel foreground execution

Now, we will extend the shell to support the execution of multiple commands in the foreground, as described below.

- Multiple user commands separated by && should be executed one after the other serially 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) and the corresponding terminated child reaped by the parent. 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 and all terminated children reaped correctly.

Like in the previous parts of the assignment, you may assume that the commands entered for serial or parallel execution are simple Linux commands, and the user enters only one type of command (serial

or parallel) at a time on the command prompt. You may also assume that there are spaces on either side of the special tokens && and &&&. You may assume that there are no more than 64 foreground commands given at a time. Once again, use multiple long running commands like `sleep` to test your series and parallel implementations, as such commands will give you enough time to run `ps` in another window to check that the commands are executing as specified.

The handling of the Ctrl+C signal should terminate all foreground processes running in serial or parallel. 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.

Submission instructions

- You must submit the shell code `my_shell.c` or `my_shell.cpp`.
- Place this file and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.