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. You may need to invoke different variants of exec like `execvp` or `execlp`, depending on the format of the arguments.
- 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 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.

Warm-up exercises

1. Write a program that forks a child process using the fork system call. The child should print "I am child", followed by its PID, and exit. The parent, after forking, must print "I am parent", followed by its PID. 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? Can you ensure that the parent always prints after the child by a suitable use of the wait system call?

2. Write a program that uses the exec system call to run the "ls -l" command in the program. Running your program should produce the same output as that produced by the "ls -l" command.

   You must write three versions of your program using the following three different variants of the exec command: `execl` (takes arguments as a list), `execlp` (takes arguments as a list, and searches system PATH for the executable), and `execvp` (takes arguments as a vector, and searches system PATH for the executable).

   Hint: Example code to invoke these different variants of exec is given below.

```
execl("/bin/ls", "ls", "-l", (char *)NULL);

execlp("ls", "ls", "-l", (char *)NULL);

char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
```

3. Write a program `runcmd.c` that executes a simple Linux command with a single argument, for example, `sleep 10`. Your program should take two command-line arguments: the name of the command and the argument to the command. 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. You can choose a suitable variant of exec.

   Hint: In a main function defined as `int main(int argc, char *argv[])`, the variable `argc` contains the number of arguments, and you can access the `char *` string command-line arguments using the variables `argv[0]`, `argv[1]`, and so on.

   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
```

Part A: A simple shell

We will now build a simple shell to run 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. 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 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 are given starter code `my_shell.c` which reads in user input and "tokenizes" the string for you. The tokenization function returns a 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 in a vector, e.g., `execvp`, you should be able to pass the entire `tokens` array easily. You must add code to `my_shell.c` to fork a child process, execute the commands found in these "tokens" in the child, and reap the child.

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. *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 the next command. There are two types of errors to handle here. First, if the command itself does not exist, then the exec system call will fail. In this case, your shell must print an error message and move on to the next command. The second case is if the command exists but the arguments are incorrect. Since you cannot check all possible arguments of all commands, you can simply call exec and let the error message be printed on screen by the executable itself.

To test this lab, run a few common Linux commands in your shell, and check that the output matches what you would get on the 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. Please verify this property using the `ps` command, in parallel in another terminal, when testing your shell code.

You can now continue adding more features to the shell, as described below.

- Change the command prompt of your shell as follows. Your shell must show the current working directory (cwd) in the command prompt, e.g., if the current directory of the shell is `/home/labuser`, then the command prompt should be `/home/labuser $`

  You may assume that the current working directory is no more than 256 characters in size. You can use the function `getcwd(char *buf, size_t size)`, where cwd will be stored and returned in the `buf` array. Read `man getcwd` for more information.

- 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.

- Modify your shell code so that the child process spawned to run a command returns a suitable exit code in case of errors. Specifically, in case of an incorrect command where `exec` fails, the child should exit with exit code 1. The parent shell process must read this exit code of the child and suitably display it. For example, if the child exited with an exit code 1, the shell should print `EXITSTATUS: 1`. Note that if you use `wait(&ws)` to reap a child, where `ws` is an integer, then `WEXITSTATUS(ws)` gives the exit code of the child process. Read `man wait` for more information.

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. Hint: you may use a variant of the wait system call, e.g., `waitpid(-1, NULL, WNOHANG)` to reap background processes without blocking. Read up more on how to invoke wait without blocking.

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 command 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 foreground process, without terminating itself. To do this, you must understand how to write custom signal handlers to "catch" signals and override the default signal handling mechanism, using interfaces such as `signal()` or `sigaction()`.

You must also understand the concept of process groups in order to solve this part correctly. 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, say background processes, to receive the signal, then you must place these children in a separate process group. You can use the `setpgid` system call to change the process group of a process. 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. Note that you must use valid PIDs or PGIDs when invoking this system call.

To solve this part correctly, you must carefully place the background processes in a separate process group, so that they do not receive the Ctrl+C signal and terminate. You can do this either by placing each background process in its own process group, say, using `setpgid(0,0)`, or by placing all background processes in a common process group that is not its parent shell's group. 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 Ctrl+C.

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

Finally, 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 in parallel mode, the shell must terminate all foreground commands and return to the command prompt. 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. (You may also optionally try the variant of terminating only the current command in the sequence and moving on to the next command, instead of cancelling all subsequent commands in the sequence.)