Project 0: Implement a Basic Shell

Due April 19, 2017 : 11:59:59pm

Objective

Understand basic Unix system calls and use them for a basic shell implementation.

Specification (Weight 15)

This basic shell is a command line interpreter that accepts input from the user and executes the commands. Similar to the well-known shells such as *bash* or *tcsh*, this shell can execute commands, redirect the standard input or standard output of commands to files, pipe the output of commands to other commands.

When the shell is ready to accept commands, a user can type commands. Commands are alphanumeric tokens (e.g., ls, ps, cat) that represent programs that should be executed. This shell should search for these programs in the directories determined by the *PATH* environment variable. Commands can have arguments. Thus, tokens that follow a command (separated by white space) are treated as the arguments to this command (e.g., `cat x`).

In addition to execute commands with their arguments, your shell supports the standard Unix I/O redirection meta-characters with '<' and '>', command pipeline with '|' and a simple signal handling. More details are here.

In case of errors (e.g., invalid input, command not found, ...) your shell should display an error and wait for the next input.

To exit the shell, the user may type "exit" or Ctrl-Z twice (pressing the Z button while holding control).

Your shell is supposed to collect the exit codes of all processes that it spawns.

Simplification :

You may assume there is a white space separating all tokens. There are at most 3 commands connected by pipe '|'.

You may assume that the maximum length of individual tokens never exceeds 32 characters, and that the maximum length of an input line never exceeds 512 characters.

How to Test

You may test your program to run a sequence of commands using "shell < testfile" and an example of this test file is.
ls > temp
cat < inputfile | sort | wc > temp
cat < temp > temp1
rm temp
ls | sort | uniq
exit

What to Submit (To be updated)

Your shell implementation should use the `fork()` system call and the `execvp()` system call (or one of its variants) to execute commands, and `dup2()` for I/O pipes and redirection. It should also use `waitpid()` or `wait()` to wait for a program to complete execution. You might also find the documentation for signals useful to be able to collect the status of processes that exit when running in the background.

Use the instructions on this page to submit your code. If you are working with partners, be sure to form a group on submit.cs and submit your code as part of that group! You will be turning in the following files for project 0:

- shell.c

Hints and Suggestions:

1. A quick guide on C UNIX/Linux processes and I/O redirection
2. Download sample code for a simple shell program from here which includes test files. The submit.cs auto test program will use some additional test files. To get the full credit, make sure your code can execute these test command files correctly. A simple shell such as this needs a command-line parser to figure out what the user is trying to do. To read a line from the user, you may use `fgets(3)`.
3. If a valid command has been entered, the shell should `fork` to create a new (child) process, and the child process should `exec` the command.
4. Before calling `exec` to begin execution, the child process may have to close stdin (file descriptor 0) or stdout (file descriptor 1), open the corresponding file or pipe (with open for files, and pipe for pipes), and use `dup2(2)` to make it the appropriate file descriptor. After calling `dup2`, close the old file descriptor.
5. The main challenge of calling `execvp` is to build the argument list correctly. If you use `execvp`, remember that the first argument in the array is the name of the command itself, and the last argument must be a null pointer.
6. The easiest way to redirect input and output is to follow these steps in order: (a) open (or create) the input or output file (or pipe). (b) close the corresponding standard file descriptor (stdin or stdout). (c) use dup2 to make file descriptor 0 or 1 correspond to your newly opened file. (d) close the newly opened file (without closing the standard file descriptor).
7. When executing a command line that requires a pipe, the pipe must be created before forking the child processes. Also, if there are multiple pipes, the command(s) in the middle may have both input and output redirected to pipes. Finally, be sure the pipe is closed in the parent process, so that termination of the process writing to the pipe will automatically close the pipe and send an EOF (end of file) to the process reading the pipe.
8. Any pipe or file opened in the parent process may be closed as soon as the child is forked -- this will not affect the open file descriptor in the child.