**CS232 Operating Systems Lab**

**Winter 2021**

**Lab 2**

**A Simple Shell**

**DUE DATE: January 27 2020**

**Lab 1 due date (revised): 18/1/2021**

|  |
| --- |
| **Building a shell** |

**LINUX shells:**   
The OS command interpreter is the program that people interact with in order to launch and control programs. On LINUX systems, the command interpreter is often called **shell**: a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. bash, sh, ksh, csh, tcsh, ... are all examples of LINUX shells. You use a shell like this every time you log into a Linux machine and bring up a terminal. It might be useful to look at the manual pages of these shells, for example, type "man bash" to find out about the default Linux shell “bash”.

The most rudimentary shell is structured as the following loop:

1. Print out a prompt (e.g., "CS232/456Shell$ ");
2. Read a line from the user;
3. Parse the line into the program name and an array of parameters;
4. Use the fork() system call to spawn a new child process;
   * The child process then uses the exec() system call (or one of its variants) to launch the specified program;
   * The parent process (the shell) uses the wait() system call (or one of its variants) to wait for the child to terminate;
5. Once the child (the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most commands people type on the shell prompt are the names of other LINUX programs (such as ps or cat), shells also recognize some special commands (called internal commands) that are not program names. For example, the exit command terminates the shell, and the cd command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

**Requirements in detail:**   
Your job is to implement a very primitive shell that knows how to launch new programs in the foreground and the background. It should also recognize a few internal commands. More specifically, it should support the following features.

* It should recognize the internal commands: exit, cd, pwd, source, and echo. exit should use the exit() system call to terminate the shell. cd uses the chdir() system call to change to a new directory. pwd prints the current directory. Store the current directory string in a variable. Change it when cd is executed. The first time, use the library function “getcwd” from your C program. Source reads commands one line at a time from the named file and executes them. Echo echoes the string that follows onto the screen.
* If the command line does not indicate any internal commands, it should be in the following form:  
       <program name> <arg1> <arg2> .... <argN>  
  Your shell should invoke the program, passing it the list of arguments in the command line. The shell must wait until the started program completes unless the user runs it in the background .
* Implement I/O redirection

$cat <myfile

Will output the contents of myfile to the screen (same as cat myfile)

Cat myfile >outfile will output myfile contents to the file outfile

Cat <myfile > outfile will do the same thing as above.

* Implement the ; feature:

$ sleep 2; echo “done”

The shell will fork a process which will sleep for 2 seconds, When it returns, the shell will execute the “echo builtin command, echoing “done”.

$ sleep 2; cat myfile

The shell will fork a process which will sleep for 2 seconds, When it returns, the shell will fork another process which will run cat with the argument “myfile”.

* You will have to implement Pipes. See below for details of what you have to do.
* Catch the control C (^C) interrupt: take no action, except for showing the prompt again in the next line (check what happens in bash). Children that are forked and running when ^C is pressed will get terminated.

To allow users to pass arguments you need to parse the input line into words separated by whitespace (spaces and '\t' tab characters). You might try to use strtok\_r() for parsing (check the manual page of strtok\_r() and Google it for examples of using it). In case you wonder, strtok\_r() is a user-level utility, not a system call. This means this function is fulfilled without the help of the operating system kernel. To make the parsing easy for you, you can assume the '&' token (when used) is separated from the last argument with one or more spaces or '\t' tab characters.

The shell runs programs using two core system calls: fork() and execvp(). Read the manual pages to see how to use them. In short, fork() creates an exact copy of the currently running process, and is used by the shell to spawn a new process. The execvp() call is used to overload the currently running program with a new program, which is how the shell turns a forked process into the program it wants to run. In addition, the shell must wait until the previously started program completes unless the user runs it in the background (with &). This is done with the wait() system call or one of its variants (such as waitpid()). All these system calls can fail due to unforeseen reasons (see their manual pages for details). You should check their return status and report errors if they occur.

No input the user gives should cause the shell to exit (except when the user types exit or Ctrl+D). This means your shell should handle errors gracefully, no matter where they occur. Even if an error occurs in the middle of a long pipeline, it should be reported accurately and your shell should recover gracefully. In addition, your shell should not generate leaking open file descriptors. **Hint:** you can monitor the current open file descriptors of the shell process through the /proc file system.

**Pipes:**   
Your shell needs to support pipes. Pipes allow the stdins and stdouts of a list of programs to be concatenated in a chain. More specifically, the first program's stdout is directed to the stdin of the second program; the second program's stdout is directed to the stdin of the third program; and so on so forth. Multiple piped programs in a command line are separated with the token "|". Allow at most three such pipes. A command line will therefore have the following form:  
     <program1> <arglist1> | <program2> <arglist2> | <program3> <arglist3>   
Try an example like this: pick a text file with more than 10 lines (assume it is called textfile) and then type  
     cat textfile | sed p| tail -n 10  
in a regular shell. Pause a bit to think what it really does. Note that multiple processes need to be launched for piped commands and all of them should be waited on in a foreground execution. The pipe() and dup2() system calls will be useful.

Testing your programme:

1) Create the file myfile in your main directory from you program by doing the following:

cat > myfile

The pen is mightier than the sword

The rain in Spain

Falls mainly in the plain

^d

2) The following script in file lab2script must then run without errors:

ls

mkdir lab2temp

echo “hello lab2”

ls

cd mkdir2temp

cp ../myfile myfile

cat myfile

cat < myfile

sleep 2 ; cat myfile

sleep 2 ; echo “middle” ; cat < myfile

cat myfile | sed p | tail –n 4

sleep 10

^C

echo “Done!”

The script is to be kep in the lab2script. It is to be executed from your shell by

$ source lab2temp

3) in file lab2script\_error put the following and run it using “source” AFTER running la2script in your main directory.

ls

mkdir mkdir2temp

cd mkdir2temp

chmod 555 myfile

cat > myfile

4) Your executable file is to be named lab2shell