EE 3233 Systems Programing for Engineers (Fall 2020)

**Project 1:** **UNIX shell**

## Objectives

* To develop defensive programming skills
* To gain intimate familiarity with Linux shell functionality
* To learn about the process life cycle and how they are handled

## Overview

In this project you will implement a shell or command line interpreter. It will operate in a basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shell you will implement will be similar to, but much simpler than the one you use every day in Linux. You can find out which shell you are running by typing “**echo $SHELL”** at a prompt. You may then wish to look at the man pages for sh or the shell you are running (more likely bash) to learn more about all of the functionality that can be present. For this project, you do not need to implement much functionality; but you will need to be able to handle running multiple commands simultaneously.

Your shell can be run in two ways: interactive and batch. In interactive mode, you will display a prompt (any string of your choosing) and the user of the shell will type in a command at the prompt. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the list of commands that should be executed. In batch mode, you should not display a prompt. In batch mode you should echo each line you read from the batch file back to the user before executing it; this will help you when you debug your shells (and us when we test your programs). In both interactive and batch mode, your shell stops accepting new commands when it sees the quit command on a line or reaches the end of the input stream (i.e., the end of the batch file or the user types 'Ctrl-D'). The shell should then exit after all running processes have terminated.

Each line (of the batch file or typed at the prompt) may contain multiple commands separated with the ; character. Each of the commands separated by a “;” should be run simultaneously, or concurrently. (Note that this is different behavior than standard Linux shells which run these commands one at a time, in order.) The shell should not print the next prompt or take more input until all of these commands have finished executing (the wait() and/or waitpid() system calls may be useful here). For example, the following lines are all valid and have reasonable commands specified:

shell >

shell> ls

shell> /bin/ls

shell> ls -l

shell> ls -l ; cat file

shell> ls -l ; cat file ; grep foo file2

For example, on the last line, the commands ls -l , cat file and grep foo file2 should all be running at the same time; as a result, you may see that their output is intermixed.

To exit the shell, the user can type quit. This should just exit the shell and be done with it (the exit() system call will be useful here). Note that quit is a built-in shell command; it is not to be executed like other programs the user types in. If the "quit" command is on the same line with other commands, you should ensure that the other commands execute (and finish) before you exit your shell.

These are all valid examples for quitting the shell:

shell > quit

shell > quit ; cat file

shell > cat file ; quit

This project is not as hard as it may seem at first reading (or perhaps it doesn't seem that hard at all, which is good!); in fact, the code you write will be much smaller than this specification. Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. Your finished programs will probably be under 200 lines, including comments. If you find that you are writing a lot of code, it probably means that you are doing something wrong and should take a break from hacking and instead think about what you are trying to do.

## Program Specifications

Your C program must be invoked exactly as follows:

shell [batchFile]

The command line arguments to your shell are to be interpreted as follows.

batchFile: an optional argument (often indicated by square brackets as above). If present, your shell will read each line of the batchFile for commands to be executed. If not present, your shell will run in interactive mode by printing a prompt to the user at stdout and reading the command from stdin.

For example, if you run your program as:

shell /home/user/student/ batchfile

then your program will read commands from /home/user/student/batchfile until it sees the quit command.

Defensive programming is an important concept in operating systems: an OS can't simply fail when it encounters an error; it must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation.

You should consider the following situations as errors; in each case, your shell should print a message (to stderr) and exit gracefully:

* An incorrect number of command line arguments to your shell program.
* The batch file does not exist or cannot be opened.

For the following situation, you should print a message to the user (stderr) and continue processing:

* A command does not exist or cannot be executed.

Optionally, to make coding your shell easier, you may print an error message and continue processing in the following situation:

* A very long command line (for this project, over 512 characters including the '\n').

Your shell should also be able to handle the following scenarios, which are not errors (i.e., your shell should not print an error message):

* An empty command line.
* Extra white spaces within a command line.
* Batch file ends without quit command or user types 'Ctrl-D' as command in interactive mode.

In no case, should any input or any command line format cause your shell program to crash or to exit prematurely. You should think carefully about how you want to handle oddly formatted command lines (e.g., lines with no commands between a ;). In these cases, you may choose to print a warning message and/or execute some subset of the commands. However, in all cases, your shell should continue to execute!

shell> ; cat file ; grep foo file2

shell > cat file ; ; grep foo file2

shell > cat file ; ls -l ;

shell > cat file ;;;; ls -l

shell > ;; ls -l

shell

> ;

## Hints

Your shell is basically a loop: it repeatedly prints a prompt (if in interactive mode), parses the input, executes the command specified on that line of input, and waits for the command to finish, if it is in the foreground. This is repeated until the user types "quit" or ends their input.

You should structure your shell such that it creates a new process for each new command. There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows easy concurrency; that is, multiple commands can be started and allowed to execute simultaneously (i.e., in parallel style).

To simplify things for you in this first assignment, we will suggest a few library routines you may want to use to make your coding easier. (Do not expect this detailed of advice for future assignments!) You are free to use these routines if you want or to disregard our suggestions.

To find information on library routines to use, look at the manual pages (using the Unix command **man**). You will also find man pages useful for seeing which header files you should include.

## Parsing

For reading lines of input, you may want to look at **fgets().**To open a file and get a handle with type **FILE \***, look into **fopen().**Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()**is useful for displaying the problem.) You may find the **strtok()**routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or ...).

## Executing Commands

Look into fork(), execvp(), and wait/waitpid().

The fork() system call creates a new process. After this point, two processes will be executing within your code. You will be able to differentiate the child from the parent by looking at the return value of fork; the child sees a 0, the parent sees the pid of the child.

You will note that there are a variety of commands in the exec family; **for this project, you must use**execvp(). Remember that if execvp() is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, char \*argv[] matches those that the program sees in its function prototype:

int main(int argc, char \*argv[]);

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

foo 205 535

then argv[0] = "foo", argv[1] = "205" and argv[2] = "535". **Important:**the list of arguments must be terminated with a NULL pointer; that is, argv[3] = NULL. We strongly recommend that you carefully check that you are constructing this array correctly!

The wait()/waitpid() system calls allow the parent process to wait for its children. Read the man pages for more details.

## Grading

For this project, you need to hand in four distinct items:

* Your source code (no object files or executables, please!)
* A Makefile for compiling your source code
* A README file with some basic documentation about your code

Hand in your source code. You can zip everything together in a file labeled your\_name\_project1.zip.

When you submit all of your source code, make sure everything is in proper relative directories for compilation. Do **not**submit any .o files.

To ensure that we compile your C correctly for the demo, you will need to create a simple **makefile;**this way our scripts can just run make to compile your code with the right libraries and flags. If you don't know how to write a makefile, you might want to look at the man pages for make, or better yet, read this little [tutorial.](http://www.cs.wisc.edu/~remzi/Classes/537/Fall2005/tut.pdf) Otherwise, check out this very simple makefile:

#####################################################################

#

# shell.c is the name of your source code; you may change this.

# However, you must keep the name of the executable as "shell".

#

# Type "make" or "make shell" to compile your code

#

# Type "make clean" to remove the executable (and any object files)

#

#####################################################################

CC=gcc

CFLAGS=-Wall -g

shell: shell.c

$(CC) -o shell $(CFLAGS) shell.c

clean:

$(RM) shell

Finally, we would like to see a file called README describing your code. This file should contain the following four sections:

* Your name and login information
* Design overview: A few paragraphs describing the overall structure of your code and any important structures.
* Complete specification: Describe how you handled any ambiguities in the specification. For example, for this project, explain how your shell will handle lines that have no commands between semi-colons.
* Known bugs or problems: A list of any features that you did not implement or that you know are not working correctly

Due to the simplicity of this project, the documentation for this project is fairly minimal.

The majority of your grade for this assignment will depend upon how well your implementation works and a smaller portion will be given for documentation and style. We will run your program on different test cases, some of which will exercise your programs ability to correctly execute commands and some of which will test your programs ability to catch error conditions. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites, so that you will not be unpleasantly surprised when we run our tests.

While you can develop your code on any system that you want, make sure that your code runs correctly on a machine that runs the Linux operating system. Specifically, since libraries and environments sometimes vary in small and large ways across systems, you should verify your code on the linux VM I distributed. This will be the environment in which the projects will be tested.

## Final Note

The shell you are building is very simplistic. It doesn't have a PATH variable, it doesn't support changing directories, there is no shell history of previous commands you have run, the user can't customize the prompt, etc. Feel free to play around with adding such functionality; it will give you some more insight into how things really work. However, please don't let your fun ruin your code. Please turn in a copy of the shell that implements only the functionality described in this specification section for the project.