Assignment 1

Tiny Unix Shell

Important Notes

1. This is an **individual** project. Do your own work!
2. The project is based on the Linux operating system.
3. Make sure your program works on linprog.cs.fsu.edu because that is where it will be graded.

Overview

In this assignment, you will implement a command line interpreter (or a shell). The shell should operate in this 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 the child has finished. The shells you implement will be similar to, but much simpler than, the one you run every day in Unix. 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 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() functions may be useful here). For example, the following lines are all valid and have reasonable commands specified:

```
prompt>
prompt> ls
prompt>/bin/ls prompt> ls -l
prompt> ls -l ; cat file
prompt> 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() function will be useful here). Note that quit should be 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.

```
prompt> quit
prompt> quit ; cat  file
prompt> cat  file ; quit
```

This project is not as hard as it may seem at first reading; 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 coding and instead think about what you are trying to do.

Program Specifications

Your C program must be invoked exactly as follows:

```
tinysh [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 the interactive mode by printing a prompt to the user at stdout and reading the commands from stdin. For example, if you run your program as:

```
tinysh /home/usr/batchfile
```

then your program will read commands from /home/usr/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 (to 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!

```
prompt> ; cat file ;grep foo  file2
prompt> cat file ; ;  grep foo file2
prompt> cat  file ; ls -l ;
prompt> cat file ;;;;  ls -l
prompt> ;; ls -l  prompt> ;
```

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. To find information on these library routines, look at the manual pages (using the Unix command man or just Google it). 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 white-space 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.

Miscellaneous Hints

Remember to get the basic functionality of your shell working before worrying about all of the error conditions and end cases. For example, first focus on interactive mode, and get a single command running (probably first a command with no arguments, such as "ls"). Then, add in the functionality to work in batch mode (most of our test cases will use batch mode, so make sure this works!). Next, try working on multiple jobs separated with the ; character. Finally, make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands.

We strongly recommend that you check the return codes of *all* system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of junk at it and make sure the shell behaves well. Good code comes through testing – you must run all sorts of different tests to make sure things work as desired. Don't be gentle – other users/your grader certainly won't be. Break it now so we don't have to break it later.

Keep versions of your code. More advanced programmers will use a source control system such as GIT. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

Grading

For this project, you need to submit the following items in a zip file (name your file as os_proj1_fsuid.zip) to Canvas assignment link:

- Your source code in a single file, called shell.c. Your code will be compiled with the following command:

gcc -std=c99 -Wall -Wextra -Werror -o tinysh shell.c

Don't submit object files or executables. Make sure your code compiles. Code that does not compile gets zero points.

- A README file with some basic documentation about your code. In this file, include the following information:

1. Your name and FSUID
2. Design overview: A few paragraphs describing the overall structure of your code and any important structures.
3. 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 and other error/warning conditions.