COMP213 Final Project

In this project, you will implement a command-line interpreter or 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 it has finished.

The shells you implement will be similar to, but much simpler than, the one you run every day in Unix. 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 `tcsh` or `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 shell (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:

```
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()` 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.

```
prompt> quit
prompt> quit ; cat file
prompt> 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 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 /u/j/v/batchfile
```

then your program will read commands from `/u/j/v/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:

1. An incorrect number of command line arguments to your shell program.
2. 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 a 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!

Example problematic input lines that your shell should handle gracefully:

```
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> ;
```