Lab 03 – The UNIX Shell (Part I)

Objectives

1. Be able to clearly explain what a shell program is.
2. Learn the syntax of a UNIX command.
3. Learn the difference between absolute and relative pathnames.
4. Learn about the PATH environment variable.
5. Learn about shell metacharacters.
6. Learn about the fork() and exec() system calls.
7. Write your own shell in C.

Pre-Reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

What is a Shell?

Remember the following:

A shell is a *command interpreter*

When you type a command and press **<Enter>**, the shell interprets your command and executes it.

The General Syntax of a UNIX Command

When the shell program starts, a shell prompt appears on the screen. This is an indication that the shell is ready to accept keystrokes in the command line that directly follows the prompt.

The general syntax, or structure, of a single command (as opposed to a command line that may have multiple commands typed on the same line separated with input and output redirection characters) as it is typed on the command line is as follows:

General Command Line Syntax

```
$ command [[-]option(s)] [option argument(s)] [command argument(s)]
```

Where

- **$** is the shell prompt.
- Anything enclosed in **[ ]** is not always needed.
- **command** is the name of the valid UNIX command in lowercase letters.
- **[[-]option(s)]** is one or more modifiers that change the behaviour of **command**.

- **`[option argument(s)]`** is one or more modifiers that change the behaviour of **`[[-]option(s)]`**.
- **`[command argument(s)]`** is one or more objects that are affected by **`command`**.

Note the following four essentials:

- A space separates commands, options, option arguments, and command arguments, but no space is necessary between multiple option(s) or multiple option arguments.
- The order of multiple options or option arguments is irrelevant.
- A space character is optional between the option and the option argument.
- Always press the **`<Enter>`** key to submit the command for interpretation and execution.

The following are examples of commands typed on the UNIX command line. These examples illustrate some of the variations of the correct syntax for a single command that may have options and arguments.

```
$ ls
$ ls -l -a
$ ls -la
$ ls -la *.txt
```

If the command executes properly, then you are returned to the shell prompt; if it does not execute properly, then you get an error message displayed on the command line, and you are returned to the shell prompt. Note that the UNIX command line is **case-sensitive**.

Absolute and Relative Pathnames

The pathname for a file in the UNIX file system can be given in two ways:

- **Absolute pathname**: the full path of the file. The absolute pathname of the root directory is (/). The absolute pathname of any other file begins with (/) and ends with the filename, which means that the path *begins* at the root and *ends* at the file.

- **Relative pathname**: the path of the file relative to the current working directory (the directory the shell is currently in). A relative pathname begins with either a dot (.), which is interpreted as though it were the absolute pathname of the current working directory, or two dots (..), which is interpreted as though it were the absolute pathname of the parent directory of the current working directory.
  For example, if you are currently in the directory /usr, then the relative pathname of the executable file /bin/bash is ../../bin/bash.

The Search Path

The shell interprets your commands by assuming that the first word in a command line is the name of the command that you want to execute. To execute a command, the shell searches several directories in the file system, looking for a file that has the name of the command. If found, then shell executes the file.

In the Bourne shell program (/bin/sh), as well as the Korn and Bash shells, the names of the directories that the shell searches to find the file corresponding to a command are stored in the shell variable PATH. To view this variable, use the echo command.

```
echo $PATH
```

The following is a sample run of this command under the Bourne shell, although it would be the same in Korn and Bash.

```
$ echo $PATH
/usr/lib/libfm:/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin
:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

The directory names in the search path are separated by colons.

Shell Metacharacters

Most of the characters other than letters and digits have special meaning to the shell. These characters are called shell metacharacters and therefore cannot be used in shell commands as literal characters without specifying them in a certain way. Shell metacharacters include:

```
Newline Space Tab # " $ & ' ( ) * [ ] ^ ` { } | ; < > ? / \ . ~
```

Carefully read the instructions in the procedure to find out which metacharacters your shell must support in Part I and in Part II (next week's lab).

Execution of Shell Commands

When a command is entered into the shell, the shell process creates a new process to execute the command. Recall that a process is a program that is currently being executed. While the child process executes, the shell waits for it to finish. Once the child process terminates, the shell resumes execution.

A UNIX process can create another process by using the **fork** system call (which is a function call in C). This system call creates an exact main memory copy of the original process. Both processes continue executing, starting from line of code immediately after the **fork** function call. The **fork** function, when called in your C program, has a *different* return value for the parent and child processes, which tells each process which one it is. If the **fork** function tells a process it is the parent, then it must wait until the child process terminates.

The **exec** system call allows a child process to overwrite itself with the executable code for another command. Hence, the newly created child process ***becomes*** the command to be executed. Therefore, the **fork** and **exec** functions are used in tandem to execute a file.

Procedure

Write a basic shell program in C. This program should behave like a simplified version of the Bourne shell (/bin/sh). The shell program consists of a never-ending while loop with the following steps:

1. Print the prompt string to **stdout**.
2. Read in the user's command from **stdin**. The read command that is used must block (i.e. halt the program execution) until the user presses **<Enter>**, at which point the command line that was entered is saved to a character array and the process resumes execution.
3. Parse the command line into the **argv[]** string array.
4. If the command is **exit** or **quit**, then **break** the while loop so that the program can end. Otherwise, search for the executable file with the same name as the given command (the first word in the command line read from the keyboard, **argv[0]**).
5. Create a child process using the **fork** system call.
6. The parent process must wait (i.e. halt execution) until the child terminates. The child process must use the **execve** system call to run the executable file for the given command.
   Do **not** use any of the **execlp** or **execp** calls because these calls don't need for you to search the PATH environment variable. We want you to search PATH in this lab.
7. When the child process terminates, the parent resumes execution, and the while loop starts over again.

Hints:

- Take a look at the **man** page for the following C functions provided by the standard C library and system call library:
  - `getenv`
  - `access`
  - `fork`
  - `execve`
  - `wait`
  - `fprintf (for writing to streams other than stdout)`
  - `gets (for blocking read)`

- Use the **#define** directive to define constant values. For example, you should include the following constants in your program:

```
#define LINE_LEN      80
#define MAX_ARGS 64
#define MAX_ARG_LEN   64
#define MAX_PATHS     64
#define MAX_PATH_LEN  96
#define WHITESPACE    " .,\t&"
```

- Consider breaking up the steps within the while loop into functions (where possible). If your source code file becomes too large, messy, and complicated, feel free to divide the source code into several files, where each file contains a subset of the functions. Make sure to create the appropriate header files.

  Keep your code organized! You will be extending the functionality of your shell in the next lab, and you don't want to start off next week with a ton of messy code.

- The prompt string can be hard-coded into the program (rather than set as an environment variable), and it can be whatever you like. For example, implementing it can be as simple as:

```
promptString  = "mini-shell >";
void printPromt(){
      printf("%s", promptString );
}
```

- The shell does **not** need to support any fancy metacharacters. In fact, the *only* characters it must support are letters, numbers, and the basic metacharacters listed in the following table. Note: this table will be extended for Part II (next week's lab).

Metacharacter	Purpose	Example
/	To be used as the root directory and as a component separator in a pathname	**/usr/bin**
New Line	To end a command line	
Space	To separate elements on a command line	**ls /etc**
Tab	To separate elements on a command line	**ls /etc**
.	To represent the current working directory	**ls .**
..	To represent the parent directory of the current working directory	**ls ..**

Deliverables

1. All of the source and header files for your shell program.

2. A make file for building your program. The output executable file should be called **`minishell`**.