Lab 2 - The learning Shell

Learning Outcomes:

After completing the exercises in this lab you should be able:

- To analyse and understand the implementation of a simple UNIX/Posix shell program in the C programming language
- To extend an partial implementation to include basic functionality you would expect in a functional shell program

Organisation

Please attempt this lab individually as you will need this to be completed in order to complete your first term assignment project.

Prerequisites and Assumptions

This lab is designed for the Arch Linux environment you studied in Lab 1. It is assumed that, before you begin, that you are generally familiar with the Linux target environment and programming in the C language. It is strongly recommended that you complete the lab 1 exercises if you have not already done so and that you revise your understanding of C programming concepts if necessary

1. Obtaining the source code for lsh(1)

Boot the Arch Linux environment from Lab 1 and log in as user **arch**.

You can obtain the source code for this lab by executing the following commands in the **`/home/arch`** directory.

```
arch@kingfisher:~$ sudo pacman -Sy git-core

arch@kingfisher:~$ git clone git://github.com/ditbjg/2012-dt228-
os2.git
```

Change into the **`lab2`** directory and list the files. The output should look something like the following.

```
arch@kingfisher:~$ cd 2012-dt228-os2/lab2

arch@kingfisher:~$ ls -l

total 76
-rw-r--r-- 1 arch users 31790 Sep 30 00:02 LICENSE
-rw-r--r-- 1 arch users   412 Sep 30 00:02 Makefile
-rw-r--r-- 1 arch users  1606 Sep 30 00:02 fork.c
-rw-r--r-- 1 arch users  4530 Sep 30 00:02 lsh.c
-rw-r--r-- 1 arch users  4205 Sep 30 00:02 symtab.c
-rw-r--r-- 1 arch users  1235 Sep 30 00:02 symtab.h
drwxr-xr-x 2 arch users  4096 Sep 30 00:02 tests
-rw-r--r-- 1 arch users  4826 Sep 30 00:02 tokenise.c
-rw-r--r-- 1 arch users  1161 Sep 30 00:02 tokenise.h
```

The main source file for the learning shell is **lsh.c**. This is the file that you will need to modify to complete this lab.

2. Building and testing your code

To build the **lsh** binary, run **make** as follows:

```
arch@kingfisher:~$ make
```

To test your shell, run **make** as follows:

```
arch@kingfisher:~$ make tests
```

You can run the shell manually and interact with it as a user as follows:

```
arch@kingfisher:~$ ./lsh

lsh: Copyright (C) 2012 Brian Gillespie
This program comes with ABSOLUTELY NO WARRANTY; This is free
software, and you are welcome to redistribute it under certain
conditions; Type "copyright" or "license" for more information.
lsh>>
```

3. Functional requirements

The code is peppered with **TODO** comments to indicate what needs to be done and generally where. Specifically you should implement the following changes to the code you have been given:

a. Implement external command execution using the **fork()** system call. You may like to consult the week 2 lecture notes but, essentially, you will now run the command process as a child process of your shell. In other words, fork(2) before exec(2), There is sample code provided in fork.c, demonstrating how the fork(2) system call works. You can build and run this test program as follows (from within the **lab2** directory):

```
arch@kingfisher:~$ make fork

arch@kingfisher:~$ ./fork
```

b. Implement command argument passing. The starter code you have been give does not pass user arguments onto commands that is executes. You should add this facility. To assist in this, you have been given a tokeniser library to parse and split a command line input into its constituent tokens. Examples of how to use the tokeniser API are provided in the tokeniser.c file at the bottom of that file. To run the tokeniser examples as a standalone program, use the following:

```
arch@kingfisher:~$ make tokeniser

arch@kingfisher:~$ ./tokeniser
```

c. Implement shell environment variables and populate the PS1 and PATH variables with default values. Allow the setting of arbitrary variables from within lsh, according to standard Bourne shell syntax such as, for example:

```
lsh>> PS1='% '

%
```

Again, you should plan to use the **tokeniser** here to allow you to handle arguments such as quoted strings such in the above example. To assist in the implementation of lsh

environment variables, you have also been given a **symtab** library implement a symbol table. Examples of how to use the symtab API are provided in the symtab.c file at the bottom of that file. To run the symtab examples as a standalone program, use the following:

```
arch@kingfisher:~$ make symtab
```

```
arch@kingfisher:~$ ./symtab
```

Pre-populate your PS1 and PATH variables when your lsh is started, as follows:

```
PS1='lsh>> '
```

```
PATH='/bin:/usr/bin:/usr/local/bin'
```

d. Implement a PATH environment lookup for executing commands in lsh that do not contain a '/' character in their name, i.e. are neither absolute or relative paths. Your code should look in each directory listed in the PATH variable for the specified command an try to execute the first one found.

4. How will I know when I am done?

When all the tests are passing after running the following command, you have completed the lab.

```
arch@kingfisher:~$ make tests
```