Assignment 1 – A Simple Shell

Assignment Type: Advanced Assignment

Deadline: Friday, 28th of February 11.55 PM, 2020.

Introduction

The goal of the first assignment is to write our own, very simple, shell. The shell should print a command prompt and allow the user to enter a command. After entering this command, the shell should properly execute the command, this includes supplying the provided arguments to the program. The user can exit the shell using the exit command. The shell should also have a functioning cd command to change the current working directory and the shell should show the current working directory in the prompt.

A simple version of the ls command must be included. Using these simple system commands, bigger commands can be created where the output of one command is sent to another. This is for instance done using the pipe character (|).

Through redirection, using the > symbol, the output of a program can be redirected to a file that is stored on disk. As the final part of this assignment you will implement (simplified) support for handling pipes and redirection, so that your shell is capable of executing commands such as *cat Makefile | grep gcc*, *sort Makefile > sorted.txt* and *cat words.txt | sort > sortedwords.txt.*

Specification

The assignment is to program a shell program that conforms to the specification given in this section. The shell must be written in plain C. This is to prepare for the actual kernel coding in the next assignment, in which only plain C can be used. Make sure to use a C and not a C++ compiler to warn you for the use of C++ constructs, because points will be deducted when C++ constructs are used in the code. The shell should have the following features:

1. Print a command prompt that also displays the current working directory.
2. Allow the user to enter commands and execute these commands.
3. The entered command string must be tokenized into an array of strings by removing the space delimiters. Also delimiters consisting of more than one space must be handled correctly.
4. Implement *exit* (to *exit* the shell) and *cd* (to change the current working directory) as built-in commands. The *cd* should display errors if necessary.
5. Ability to find program to be launched in the file system using a hard-coded array of standard locations in case the name of the executable is not preceded by an absolute or relative path. You must implement this yourself, do not use *execlp* or *execvp*.

6. Display an appropriate error if a requested command cannot be found or is not executable.
7. Execute commands and correctly pass the provided arguments to this command.
8. Execute commands that contain a pipe character by starting two new processes that are interconnected with a pipe. Your shell should be able to handle data streams of arbitrary length.
    a. **Note 1**: your shell only has to be capable of running commands that contain a single pipe character. Check how many pipe characters a command contains and simply display an error if more than one pipe character is found.
    b. **Note 2**: to simplify implementation, you only have to deal with the case that the pipe character is separated by spaces, so the tokenizer you have to implement already creates a separate token for the pipe character. For instance, a command of the form cat *Makefile*|*wc -l* does not have to be handled correctly by your program.
9. Execute commands that contain a redirection character by redirecting the output of the command to the given filename. If the file already exists, it should be overwritten. The redirection character will only be specified once and only has to function for *stdout* (file descriptor 1). Example: *sort words.txt > sorted.txt*.
10. Execute commands that contain both a single pipe character and a single redirection character. Example: *cat words.txt | sort > sorted.txt*.
    a. **Note**: you may assume that the redirection character and associated filename will always be located at the end of the command and that the redirection character is separated with spaces. So, a redirection character may not occur before the pipe character.

Programming Language

You are required to complete this assignment using the C language. Make sure to compile your code using a C and not a C++ compiler (use *.c* as extension and not *.cc* or *.cpp*). This means that you cannot use C++ features such as classes, virtual methods and *cout* and *cin* for I/O. Some notes:

1. Instead of using *cin* and *cout* for I/O, use the *printf* and *scanf* functions.
2. To dynamically allocate memory, use the malloc and free functions instead of new and delete.
3. A man page exists about every function in the standard C library. For example, to learn more about *scanf* use *man scanf*. The manual pages about library functions are always in section 3: *man printf* will give you information about the shell command, but *man 3 printf* about the C library function. Similarly, system calls are in section 2. 2
4. Do not include iostream or set a namespace. Instead, include *<stdio.h>*, *<stdlib.h>*, *<string.h> and <unistd.h>*.
5. Ask the teaching assistant for help if you have problems!

Guide to library functions

You will have to use library functions to accomplish the various tasks. Some of these library functions are wrappers around actual system calls (POSIX API), which are traps to the operating system kernel (e.g. *fork* and *execv*).

Reading and parsing user input. There are several ways to obtain input from the user, we suggest to use *fgets*. The command string entered by the user will have to be split (or tokenized) into an argument vector using the space character as delimiter. You can do this tokenization manually, or use a provided string manipulation function such as *strsep*. Make sure to also properly deal with delimiters consisting of multiple spaces.

Executing a program. To start a new process and execute a program the *fork* and *execv* should be all you need. In the parent process you will need to use the wait system call. The first item in the argument vector indicates the program to run. If this item does not contain a slash, it is not a relative or absolute path and we must find out where this program is located in the file system. To do so, concatenate the name of the program to each of the paths in the hard-coded path array and use, for example, the stat call to test whether the file exists.

Creating a pipe. To create a pipe you have to use the pipe system call. You have to pass an array of two integers as an argument, which will be filled with the file descriptors of the input and output end of the pipe. To reconnect standard input and output to the pipe, you will also need the close and dup system calls. Also useful are the defines STDIN_FILENO and STDOUT_FILENO that represent the file descriptors for stdin and stdout respectively.

Skeleton

You can use the skeleton below as a starting point for your shell program. As the program grows, you may want to improve the structure of the code by moving certain parts into separate functions.

```
const char *mypath[] = {
       "./",
      "/usr/bin/",
      "/bin/",
       NULL
};

/* To be embedded in a suitable function ... */
while (...)
{
      /* Wait for input */
      printf ("prompt> ");
```

```
      fgets (...);

       /* Parse input */
      while (( ... = strsep (...)) != NULL)
      {
            ...
       }

      /* If necessary locate executable using mypath array */

      /* Launch executable */
       if (fork () == 0) {
           ...
           execv (...);
            ...
      } else {
            wait(…);
      }
}
```

Submissions:

This assignment is to be submitted on both Github and Slate.

Make sure you add the Teaching Assistant as collaborator on each assignment.

Username: MuhammadAbdullahAziz98

For any queries, consult the TA: Mr. M. Abdullah Aziz via email: l164085@lhr.nu.edu.pk