Project 1: Unix/Linux Shell

The University of British Columbia
Department of Electrical and Computer Engineering
EECE 315

Group A6

Alex Sismanis 56269103 – 100%
Amanbir Takhar 24442097 – 100%
Christopher Tan 49158108 – 100%
Hooman Shariati 21155098 – 100%

1. Introduction

The purpose of this project is to help students understand how processes work in an Operating System. For this project, students will code in C their own command interpreter known as a Shell in a Linux environment using the gcc compiler. By forking processes, students will understand how a kernel can remain active while producing cloning itself to carry out processes and be terminated instead. Furthermore, traversal through directories and work with environment variables helps the student learn the details of the complex commands involved in the coding of a shell, like chdir or execve. To write a working shell, our group divided the tasks and had each group member work individually. Working code was exchanged via email, and eventually compiled into a working shell.

2. Structure

The shell starts with initializing variables such as the command_t struct, after that the main code is set inside an infinite loop. Within the infinite loop, the program prompts the user for a unix command like a typical unix shell like bash would. The first argument will be read as the name of the command which is also copied to the first index of the argv member variable in the command struct. The argv member variable is an array of character pointers (i.e. strings) that take in any extra information needed to execute a command. The command->name member is compared to different scenarios for change directories, modifying and deleting environmental variables and background processors. This high-level description can be seen in the block diagram Figure 1

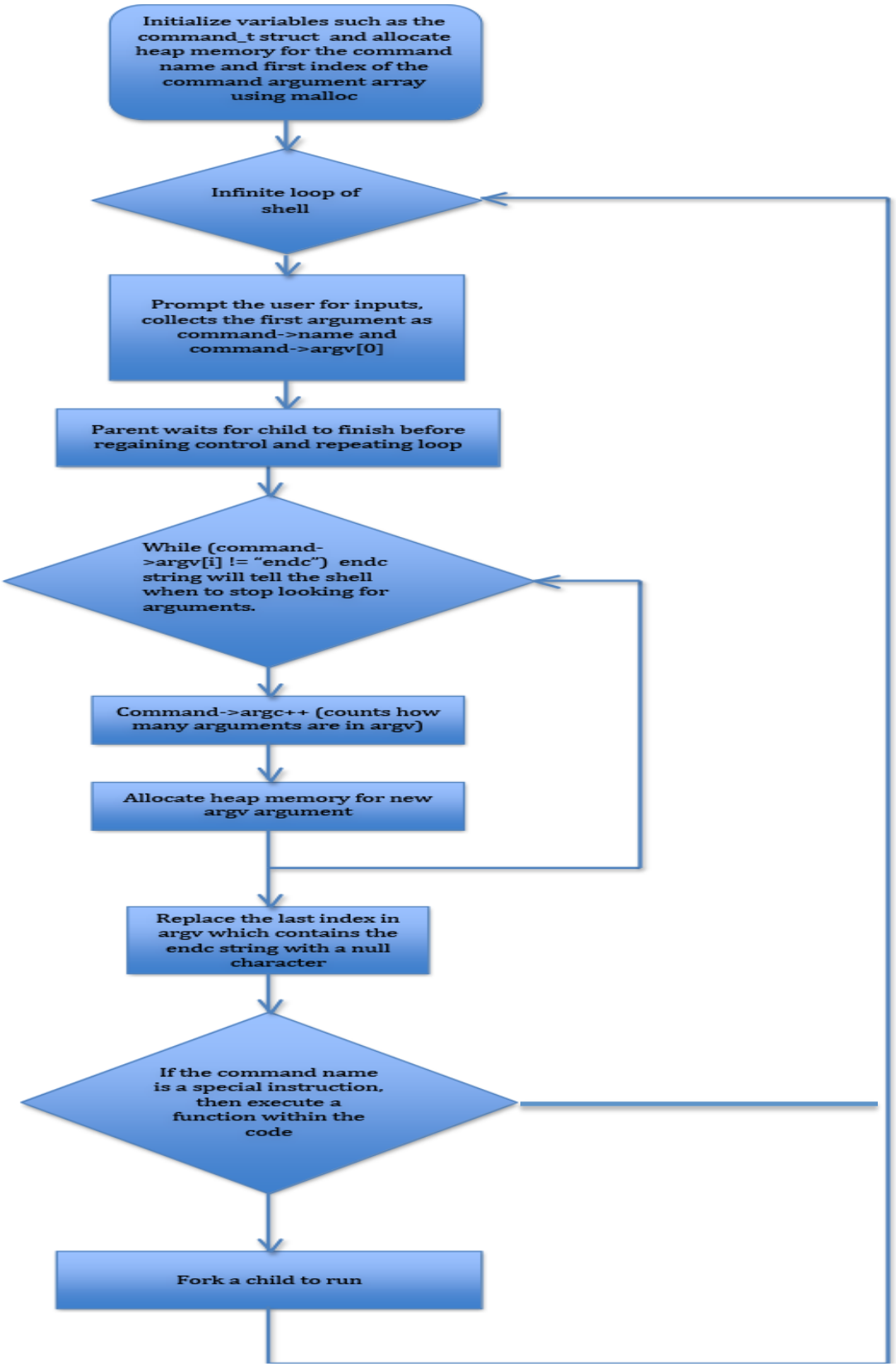

Figure 1: Block Diagram of Shell structure

3. Member Tasks

3.1 Alex Sismanis:

Understanding and writing the code to parse the PATH and then find the path of a given command. By parsing the path allowing the different directories of path to be easily interpreted and understood in order to check it and then later modify it. Check and modify the path of the system. Wrote the code for checking, modifying and deleting environmental variables and path.

3.2 Amanbir Takhar:

Learning how to use the chdir() function for the shell to change directory as there is no "cd" binary file in the bin. Hence, change directory has to be programmed using that function. Did some testing as well.

3.3 Christopher Tan:

Wrote the prompt, the command line parser and the function which forks a child and executes the child process which executes most of the shell processes. Integration of group member's code, some testing and help dedicate task for the group.

3.4 Hooman Shariati:

Figured out how to run background processes. Wrote the background() function to run processes in the background and print PIDs. Made sure the Shell can differentiate between each child in order for the parent to wait only for a specific child (the one running in the foreground). Implemented the go home function so when you issue the command cd, the shell automatically changes current directory to home directory. Did some debugging.

4. Solution Description

4.1 Command line parser

In order to arrive at a solution, much thought must be given to detail and execution part by part. First off, a suitable skeleton code was chosen and agreed upon by the group, which was taken from the provided project description. The basics were taken care of in the beginning, consisting of prompting the user as well as scanning the command line to understand what the user intends to do, achieved by elementary printf and scanf statements. Moving forward, we are required to parse the command line and build an argv array. To do so, the code takes the first argument as the name of the command as well as the first string in the command struct's string array command->argv. The loop which collects all the other arguments for the argv string array has to be terminated by the string endc. Once the arguments have been passed, the code proceeds to match the command->name and execute that function. If none of the conditions match, the shell will assume that the command->name is a default Unix binary command such as "ls" and "clear". If all fails, the shell will print out an error message.

4.2 Functions which parse the PATH

For parsing the path a function was created parsePath which was passed a pointer to an array with the maximum arguments as the size. It is initialized to null. Then the getenv function is used to get the PATH environment variable and store it into another char pointer called pathEnviroVar. This is then copied into myPath which is another char pointer. A pointer is then initialized to myPath and a for loop is started for each memory location that pointer points to until it points to an end

character ‘\0’. Length is set to the string length of pointer and then pointer is copied into a strng char with size of the max path length. The pointer is then set to the first instance of : in pointer by using the strchr function. If there is no more : at the current location of the pointer to the directory in the path it won’t set pointer to anything. This being the case a if loop is entered and the pointer is then set to the end of pointer and pointer is then copied into strng. At the position of dirs memory is allocated for strng to be copied. strng is then copied into dirs at the position and position is incremented and set to the global variable count which stores the amount of directories in PATH. This if is ended with a break and pathParse ends. If it is not at the last : in the directories in path it skips this whole if case and sets length2 to the location of pointer which is the first or next : in the PATH. A termination character is added to the end of length - length2 to signify the end of one directory in PATH. Memory is then allocated in dirs for the strng and it is copied into it to store the one directory. The position is then incremented and the for loop continues through all the directories in PATH. The next function is findPath. This is used to search the directories of path to find the executable file that has the same name as argv[0] which is the command the user wants to execute. First it checks if the filename is already in the root directory and returns it in the char pointer found if it is. Otherwise it iterates count times which is equal to the amount of directories in the PATH. The directory, dir[i], is then copied into the char name for each i. A / is added on the end so the argv[0] can be added to the end and it will be the path of a executable file. The access command is then used to check if it exists and we have access. If we do then found is allocated memory and name with the path of the executable is then copied into found and returned. If the executable can’t be found in all directories of path then NULL is returned. This is the basics of parsing the path and making it

readable. The next step is as simple as implementing these parsing functions. First, we wanted to be able to get the PATH environment variable and print it out. It is passed the variable that has the directories and a simple for loop is used to print each dir[i] with a \n after each one to print each directory on a new line.

The next function is to modify the PATH environment variable. The same concepts in pathParse are used and getenv gets the PATH and a spacer, :, is added then the new directory to be added is appended to the end of it. Setenv is then used to set the PATH. This same concept is used for createModEnviro which creates or modifies an environment variable. It is passed parameters for what the environment variable is and what it should be changed to and setenv is used to achieve this. For deleting an environment variable the name is passed as a parameter unsetenv is used to delete it.

4.3 The cd (change directory) function

Traversing directories was not a difficult task, it just required the understanding of the chdir function, which changes the current directory. The input is a path provided to the chdir function becomes the current working directory. One strategy was to write an external function to be called to traverse the directories, but that function gave our already partially working shell code many errors and warnings. Thus, the chdir function was simply included in the main function of our project. Where it is provided with a path, and if a -1 is returned, that means there is an error; an error message is printed to notify the user that the directory cannot be changed to the specified path. But if a 0 is returned, this means it is successful and the current working directory begins at the specified path.

Although, our group encountered difficulties with the tilde at the end of path names, we tried handling it as a system call, which was incorrect. The tilde turned out to be actually handled by the shell itself, so in order to make use of the tilde we needed to utilize the getenv() function using an environment variable “HOME” that can be fed as an argument to the original chdir() function.

The change directory was also upgraded to accept directories with spaces in their names as the path. This was done by detecting if the first character of command->argv[1] is a double quote, which basically tells the shell that this is a directory with spaces. When that is verified, the strings are concatenated together and the first and last characters (these being the double quotes) are removed. The string is then assigned back to command->argv[1].

In addition, we noticed that it is very cumbersome to write the full path name each time we want to change the working directory back to the home directory. As a result, we included a clause in the part of the code handling the cd command to make it change the directory to HOME when there are no parameters included (when commnad->argc == 1).

4.4 Running background processes

Once we could run processes successfully, we turned our attention to making them run in the background as well. It sounded like a trivial matter to do so just by making sure that the parent does not wait for the child. However, since we would end parsing only when the parser reached “endc”, including the ‘&’ would make it part of a parameter which would make the command unrecognizable. As a result, we had to include the & sign within the parser itself, as well as, designing a mechanism

to differentiate between the two. This, also, seemed to be a problem due to the fact that the execv system call requires the last element of the parameter array (argv) to be Null, which means we had to replace the “endc” with 0 which would make it impossible to differentiate between background and foreground processes. To rectify the problem, we decided to add the final zero to the parameter array right before the call to execv, separately, in each function. This proved to solve some of the previous bugs as well.

The last challenge was the fact that when we have multiple child processes, the parent waits only for the child that finishes first. This meant that as soon as we ran our first background processes, all other processes would run in the background (regardless of whether they ended with “endc” or ‘&’). To solve this issue, we had to change our wait() system call to waitpid() to make sure that the parent waits only for a particular child. Furthermore, we decided to print the PID of each running background process in order to give the user the ability to monitor their progress individually.

5. Test Cases

5.1 Command line parser and fork() function testing

The shell has been tested rigorously by inputting as many simple and common Unix commands such as “ls”, “ls -l”, “pwd”, “clear”, “env”, “echo” and “cd” just to name some. The shell has also been tested to run the executable file of the prior project (Project 0), which is in a different directory.

5.2 Executing programs and changing directories testing

Next test cases involve changing directories to the directory of where the Project 0 executable file is and the shell is to execute the file. In all cases, the shell has performed according to expectations as it was able to continuously run executable files in both the current directory and given directory as well as respond to usual commands like “clear” and “ls”. The cd function has also been tested on directories with spaces in the names.

5.3 Environmental variable testing

To test the functions “checkpath”, “setpath”, “deleteenv” and “createenv”, a few test dummy environmental variables were created and deleted successfully. The “checkpath” functions writes down the PATH in a neat format and the “setpath” function is able to modify it.

5.4 Background processes testing

Furthermore, background processes were tested using “HOME” and “PATH”, they both redirected the shell just as required. They were, also, tested using a sample program that we wrote (stored in both current and other directories which would make the system sleep for 5 seconds and then print a massage. We got the exact same result when we ran the program in the background both in Bash and our Shell.

6. Conclusion

This project helped students understand how processes work in an operating system, and mainly understand how a kernel works with the fork() function to collaborate on many tasks at once. Students implemented a self designed shell coded in c, that can run programs from different directories, modify the path of the system, traverse through directories, as well as run processes in the background. This project required more understanding of concepts and research than actual coding. Furthermore, system calls were used imminently by the kernel of the operating system, and are controlled by the shell that was developed over the duration of this project. The requirements for this project were met using a few key functions made available by the standard stdio.h and stdlib.h libraries, as well as some researched libraries like unistd.h and errno.h that gave us access to more useful functions such as chdir() and getenv(), just to name a few. All in all, rigorous research and debugging aided us in producing a working but not flawless shell, that is comparable in performance to the shell of a standard Linux box.

7. References

CCplusplus. (2012, January 05). Chdir example c. Retrieved from

http://www.ccplusplus.com/2012/01/chdir-example-c.html

Code Cogs. (n.d.). Getenv: Environment variable functions. Retrieved from

http://www.codecogs.com/reference/computing/c/stdlib.h/getenv.php

DIE. (2003). Chdir linux man page. Retrieved from

http://linux.die.net/man/3/chdir

Encrico Franchi. (2006, March 06). C simple shell. Retrieved from

http://rik0.altervista.org/snippets/csimpleshell.html

International Business Machines (IBM). (n.d.). Retreived from

http://publib.boulder.ibm.com/infocenter/zos/v1r12/index.jsp?topic=%2Fcom.ibm.zos.r12.bpxbd00%2Frtgtc.htm

Linux Gazette. (2005, February). Writing your own shell by Hiran Ramankutty. Retrieved from

http://linuxgazette.net/111/ramankutty.html

Stack Overflow. (2009, September 20). Printing current user. Retrieved from http://stackoverflow.com/questions/1451825/c-programming-printing-current-user

Stack Overflow. (2012, September 29). Writing own linux shell in c. Retrieved from http://stackoverflow.com/questions/12650092/writing-own-unix-shell-in-c-problems-with-path-and-execv