Arif Ayarci

Lab 2

2/25/2022

**Linux Shell Pseudocode**

**Tokenizing Input:**

For internal commands, user inputs will be tokenized with function strcmp() that will compare two arguments to check if they match. The first argument will be argv[#] and the second will be the input that needs to be tokenized. If the arguments match it will return 0, if not they will not return 0.

For external commands, user inputs will be obtained with function getline() to read user input line by line instead of strcmp() as these commands will be more complex. The strtok() function will be used to tokenize blank spaces.

Library needed: #include <string.h>

if(strcmp(argv[1] == “cd”) == 0){

}

If(strcmp(argv[1] == “clr”) == 0){

}

//and so on.

**Built in Commands:**

**cd or cd <directory>**

The cd command will only take 0 or 1 arguments. cd by itself will print the current working directory. cd with a directory argument after will change the current default directory to the argument inputted after the cd command.

The cd command will make use of the chdir() and getcwd() with printf() functions to operate. stat will be used to determine if a path is a directory. If the path is not a directory or there are too many arguments an error message will be printed.

Library needed: #include <unistd.h>, #include <sys/types.h>

If(argc == 2){

char path[1024];

printf(“%s\n”,getcwd(path, sizeof(path)));

}

If(argc == 3){

chdir(argv[2]);

}

**clr**

The clr command will only take this argument. clr will clear the contents of the bash.

The clr command will make use of the printf() function, printing “\e[1;1H\e[2J” as its argument. Printing this string will clear the entire bash.

Library needed: #include <stdlib.h>

printf(“\e[1;1H\e[2J”);

**dir or dir <directory>**

The dir command will only take 0 or 1 arguments. dir by itself will print the contents of the current working directory. dir with a directory argument after will print the contents of the directory inputed after the dir command.

The dir command will make use of the opendir(), readdir(), and closedir() functions to operate. The dir will also make use of directory stream DIR and struct dirent.

Library needed: #include <dirent.h>

DIR \*d;

struct dirent \*dir;

opendir(“.”); //or opendir(“argv[2]”); for directory input

while(dir = readdir(d)!= NULL){

printf(“%s\n”, dir -> d\_name);

}

closedir(d);

**path**

The path command will take 0 or more arguments, more arguments being as many paths the user wants to input as possible. The arguments in this command will be added to the search path of the shell. 0 arguments will set path variable to “”. Initial shell path should contain “/bin”.

This command will make use of the setenv() function.

Library needed: #include <stdlib.h>

int i;

for(i = 2; i < argv[argc-1]; i++){

setenv(“PATH”, argv[i], 1);

}

**environ**

The environ command will print the environmental variables. The command will make use of char \*\*envp argument in main to operate.

Libraries needed: #include <stdio.h>

int main(int argc, char \*\*argv, char \*\*envp){

int i;

for(i = 0; envp[i]; i++){

printf(“%s\n”, envp[i]);

}

}

**echo <comment>**

The echo command will take one argument and only one argument being echo <comment>. This command will simply display the comment to the bash followed by a new line. echo command will ignore blank spaces.

The command will simply make use of the printf() function and print the comment input in argv[2] in the next line. strtok() function will be used to tokenize blank spaces.

Library needed: #include <stdio.h>

If(argc == 2){

printf(“\n%s”, argv[2]);

}

**help**

The help command will print the contents of a readme file. The command will make use of the readme\_doc file in the current directory that will be printed to the output.

The command will make use of a FILE stream with fopen() and fclose() to open and close readme\_doc. getline() and printf() functions will be implemented in a while loop to print the contents of the opened file.

Libraries needed: #include <stdio.h>

FILE \*n;

char \*txt;

size\_t size;

n = fopen(“readme\_doc”, “r”);

while(getline(&txt, &size, n)!= -1){

printf(“%s”, txt);

}

fclose(n);

printf(“\n”);

**pause**

The pause command will pause the operation of the program until enter is pressed. Enter in this program will be similar to /n as enter and /n both create a next line.

The pause command will make use of the getchar() function and wait until the user give a /n input, or in this case, presses the enter button.

Library needed: #include <stdio.h>

//paused

getchar();

//resumed

**quit**

The quit command will simply exit out of the program.

The quit command will make use of exit system call with 0 as a parameter.

Library needed: #include <stdlib.h>

exit(0);

**External Commands:**

External command will be executed with fork() and exec(). fork() will return -1 if it fails, else, it will return 0 which is the duplicated child process. waitpid() function will be implemented to suspend the current process and wait for the child’s status. exec() function will be implemented after the fork() for an external command.

int pid = fork();

if(pid==-1){

printf(“error”);

exit(1);

else if(pid==0){

execvp(path, argv[#]);

}

else{

waitpid();

}

**I/O Redirection and Background Execution:**

The I/O redirection symbols will include ‘<’, ‘>’, and ‘>>’. < will indicate redirection of input from the keyboard, > will indicate redirection of an output file for overwriting files if it exists, and >> will indicate redirection of an output file that overwrites a file and appends to an existing file.

The background execution symbol will be ‘&’. ‘&’ input will run the given process in the background. Multiple ‘&’ will run multiple programs in parallel to each other.

The I/O redirections will make use of functions dup() and dup2(). open() function will be used to make use of file descriptors to perform I/O redirection. dup() will be used to duplicate the file descriptor given in its argument. If dup() fails, it will return -1 and exit with an error message. If dup() succeeds, it will return the copy of the file descriptor. dup2() will take two arguments being the oldfd and newfd. newfd will be overwritten to have the same open file description as oldfd. dup2() function has the same return values as dup() function. close() function will be implemented at the end to close the file descriptors when they are not needed anymore.

& will make use of fork() function to duplicate a shell to run programs in the background. & will make use of wait() and waitpid() functions to wait for all processes to complete.

Libraries needed: #include <unistd.h>, #include <fcntl.h>

int fd;

fd = open(argv[#], O\_TRUNC);

if(fd == -1){

printf(“error\n”);

exit(1);

}

dup2(fd, 1)

close(fd);

**Piping:**

Piping will be used to communicate between two processes. The pipe will be implemented so that the output of one process will be the input for the other process.

Piping will make use of the pipe() command taking int fds[2] as an argument, fds[2] being two file descriptors. The first file descriptor, fd[0], is opened for reading with read() function and the second, fd[1], is opened for writing with write() function. The pipe() function will return -1 if it fails and 0 if it succeeds. All pipes make use of the fork() function as two processes are needed in a pipe. When one process on one side of the pipe is being read/write the other side must be closed with close() function.

Libraries needed: #include <unistd.h>

int fds[2];

if (pipe(fd) == -1){

printf(“error”);

exit(1);

}

int pid = fork();

if(pid == 0){

close(fd[1]);

read(fd[0], argv[#], sizeof(argv[#]));

close(fd[0]);

}

else{

close(fd[0]);

write(fd[1], argv[#], sizeof(argv[#]));

close(fd[1]);

}

**Testing Plan:**

**Built in Commands:**

**cd** //prints current directory.

**cd /home** //changes directory to /home.

**clr** //clear bash.

**dir** //prints contents of current directory.

**dir /home** //prints contents of /home.

**path /usr/bin** //adds “/usr/bin” to shell path.

**environ** //prints environmental variables.

**echo hello** hello

**help** //prints help.txt contents.

**pause** //pauses screen until enter input.

**quit** //exits program

**External Commands:**

helloworld.txt & //to run helloworld.txt in the background.

ls > helloworld.txt //external command with redirection.

ls -la > helloworld.txt //external command with flag and redirection.

ls | gerp helloworld.txt //external command with pipe.

**Error Handling:**

If an error of any type occurs from the arguments or functions returning -1, only one error message will be printed. The program will then exit with system call with parameter 1.

char error\_message[30] = "An error has occurred\n";

if(n == -1){

write(STDERR\_FILENO, error\_message, strlen(error\_message));

exit(1);

}