Project 2: UNIX Shell

CS 4350\_251

Group 6:

Roger Escobedo - aevistax@gmail.com

Sarah Forbis - sef46@txstate.edu

Denise Gan - denise.gan89@gmail.com

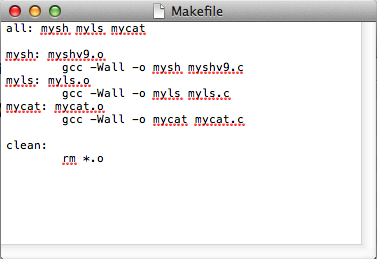
Kemal Guddeta - kemal.hussien@gmail.com

**Section l: Introduction**

1. **Roger Escobedo:** Wrote the code for the myls command, as well as the code for commands that use piping and creating a subshell. Also wrote part of the report that asks to describe the implementation of the myls command, piping, and subshells.
2. **Sarah Forbis:** Wrote the code for the mycat command and the section of the report asking to describe the implementation of the mycat command. Also provided screenshots of the results when executing commands that dealt with mycat.
3. **Denise Gan:** Wrote code for the mycp command, and the section of the report that asks to describe the implementation of the mycp command. Also provided screenshot of output when executing mycp command that will generate error and debugging screenshot.
4. **Kemal Guddeta:** Wrote code for the mycd and mypwd command, as well as the code for implementing the mysh. Also wrote the section of the report that asks to describe how the shell was implemented.
5. **Group Coordination:** When starting the project, we all first met as a group to decide who would write which command and parts of the report. There were some meetings after as a means for the group to check on each other’s code and generally see how things were going. After each meeting an email was sent out to the entire group as a recap of what happened during the meeting. That way if someone was not able to come to one of the meetings, they would still be kept up to date on what was happening within the group and project. Between these meetings, we mainly kept in contact through email, and would upload code to a Google Drive folder so other group members would stay updated on each other’s code. The report was also shared on Google Drive so that it was easier to keep all the needed material in one place.

**Section ll: Task I**

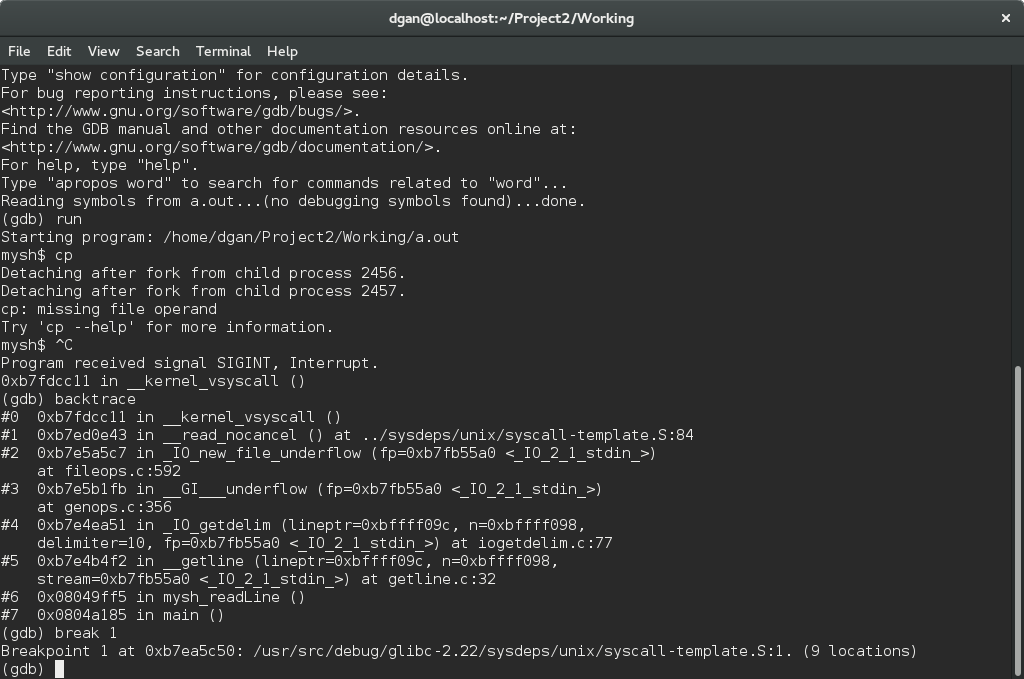
1. **Makefile for the shell program:**



1. **Makefile for one of the required commands:**



1. **Screenshot of debugging code:**



**Section III: Task II**

1. **Implementation of the mycat command**

The functionality of the mycat commands mirrors the cat commands in the regular UNIX system. For example, the command “mycat foo.txt” performs the same functionality as the command “cat foo.txt ” and prints the contents of the file “foo.txt”. The mycat commands also follow the same syntax as the cat command, with the command followed by either the name of a file or a redirection symbol - ‘<’ or ‘>’ - followed by the name of a file. The mycat command also supports having both redirection symbols and two file names within the same command, such as the command “mycat < foo.txt > foo2.txt”. However, the mycat command does not support the functionality of when just output redirection is entered, such as “mycat > foo.txt”, or just entering the name of the command without any redirection or file names.

When implementing the mycat command, one of the issues that needed to be considered was whether or not a file existed. Consider the case where the contents of one file are being redirected to another file, the system needs to check to see if the file whose contents are being redirected already exists. For example, consider the following command:

mycat foo.txt > foo2.txt

The system must check to see if the file foo.txt exists before attempting to copy its contents to foo2.txt. The system does not need to check to see if foo2.txt exists because if it already exists, its current contents will be overwritten. Otherwise, the system creates a file with the name foo2.txt.

Another issue that had to be taken into consideration was if the appropriate amount of arguments was not typed into the command line. For example, if “mycat >” or “mycat <” was entered into the command line, the system needs to print out an error statement to the user saying the redirection commands require an argument to be executed. This was handled by having a switch statement that checked each character read in from the command line, and if it encountered one of the redirection symbols, the variable flag (outRedirectFlag and inRedirectFlag) would be set to true. Then, after going through the command, an if block checks these flags to see if one or both of the redirection symbols were entered into the command line. If either of the flags were found, it then goes to another if statement to check if a file was entered into the command line as well, by checking the size of the variable fileName1 and fileName2. If both of them still have a length of zero, an error message is fired saying an argument is required for redirection.

The system also had to recognize when a command it is not built to execute and display the proper error message. For example, the commands “mycat” and “mycat > foo.txt” were not implemented into the mysh system. If either of these commands, or others not recognized by the system, an error message must be displayed saying the command entered is unknown to the user. The prior is handled in the same way as when the system checks for if arguments were provided when necessary. Before the system can read through each character of the entered command, the system checks to see if the appropriate number of arguments has been entered. If one or less arguments have been entered, it displays an error message saying the command requires an argument. The latter is handled by checking for if only the ‘>’ redirection symbol was entered and a fileName was given. If this is the case, an error message is displayed saying the command cannot be executed.



1. **Implementation of the mycp command**

The mycp command should be like that of the cp command in Unix as pertains to copying text files and directories. The syntax of the command is as prescribed in the Project 2 assignment description:

mycp a\_file b\_file

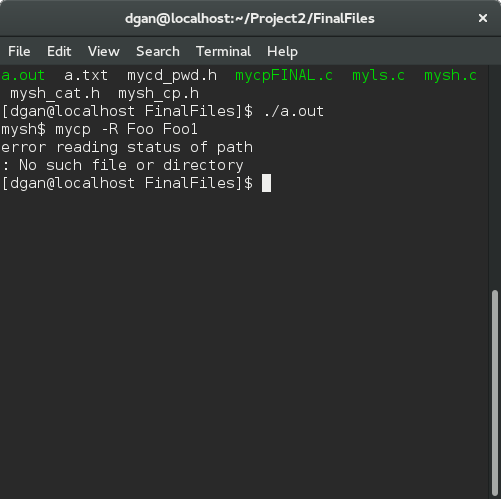
mycp ­-R a\_dir b\_dir

For example, “mycp a.txt b.txt” will create the “b.txt” file and copy the contents of “a.txt” to this new file in the current working directory. The first thing that happens is a check of user input. Did the user correctly enter a command as prescribed by the syntax? If not, the program displays the proper usage and exits. If the user did enter the correct number of arguments, then one of two things may happen: it is determined whether the user wishes to attempt a copy of two files or two directories. The latter scenario is addressed in the second half of this section. So if the correct number of argument is given (in this case four, but argv[3] should be NULL), the function copyFiles() is called and the two paths are given as arguments. The source file is opened for reading, file permissions saved, new target file created, file permissions of the new file matched to the old, read and write from source to target, and finally the source and target are closed. If at any point an error occurs in these actions, the function returns 0 for failure.

In another scenario, a user may have a directory “Foo” and wish to copy all of its contents (files and sub-directories) to a new directory “Foo1”. To be clear, “Foo1” does not yet exist. This task would be accomplished by executing the command “mycp -R Foo Foo1”. As with the previous example, the program checks whether the correct number of arguments was given (in this case four) and whether the option matches “-R”. Then, it is determined whether “Foo1” exists. If not, it is created. The contents of “Foo” is copied recursively to “Foo1”. Upon completion, newly-created “Foo1” contains a sub-directory “Foo” that is a copy of the original “Foo” directory. The void function copyDir() is called and the two paths are given as arguments. Dependent upon whether a regular file or directory is identified, copyFile() or copyDir() is called recursively. If the current file is either the current or parent directory, copyDir() immediately returns.

Among the multiple issues that had to be handled to implement this command, two will be discussed: file permissions and open and close errors. These are easily visible and easy to follow in the copyFile() function. The open() of the source, and close() operations of the source and target files is important. Before copyFIle() returns, these files are closed. Should an error occur while closing, the function returns a 0 for failure. Furthermore, consideration needs to be given to file permissions. It is vital that the file permissions of the source file be matched in the new target file. This is accomplished using stat() and fchmod(). If errors occur while doing so, the program exits and the “a\_file” is not copied to “b\_file”.

To demonstrate an error message that may occur when executing “mycp” in bash, a directory that does not exist is attempted to be copied. The command to be executed is “mycp -R Foo Foo1”. The error output is shown below:



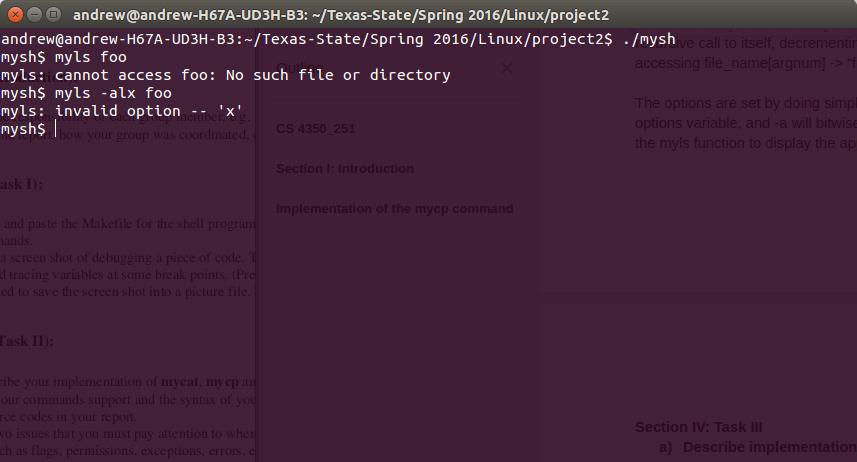
1. **Implementation of the myls command**

Myls is implemented as an external program. The program uses getopt to parse out the options given in the command line. As of right now, the program accepts two different arguments. The -a and -l option. If an invalid option is given, the program will immediately exit and return a usage error that’s already predefined by the library.

The arguments are then passed into the myls() function. The function checks whether or not the current argument is a directory. If it is a directory, then the function will display all of the contents inside of the directory. If the argument is not a directory, the argument is then checked if it is a valid file. If the file is not valid, then there is an output stating that the current argument cannot be accessed.

Because myls is able to take multiple files and multiple directories as arguments, it needs to be able to display the necessary information for all given arguments myls(int options, int argnum, char\*\* file\_name). So $(myls file1 directory1) for example will display file\_name[argnum] -> “directory1” and it’s appropriate information, depending on the options, and then make a recursive call to itself, decrementing the argnum pointer that is passed to the parameters, and accessing file\_name[argnum] -> “file1”.

The options are set by doing simple bitwise operations. -l for example will biwise | (or) 1 with the options variable, and -a will bitwise | (or) 2 with options. The options are are masked with 0 in the myls function to display the appropriate information.



**Section IV: Task III**

1. **Describe implementation of the shell program.**

Implementation of our own own shell program requires understanding of what shell does in its lifetime. Basically it passes through three steps. The first step is initialization, where a typical shell would read and execute its configuration files. The next step is interpretation and execution of commands it reads from standard inputs and finally it executes any shutdown commands, frees up any memory, and terminates. Since our shell is so simple we do not apply any configuration files and shutdown commands rather we use a do while looping in the main function to terminate a program. So during its looping in the main our shell does three basic functions; reading the command from standard input, separating the command strings into program and array of arguments, and finally executing the parsed command.

To read string of commands from the standard input a function **mysh\_readLine ()** was used. This function uses **getline ()** c library function which is very easy to use. After reading a string of command we need to spare the string of commands into program and arguments. To sparse the command string we used two functions based on the presence of special characters in the command to figure out if there is piping and redirection are required or not. If there is no redirection and or piping we employed a simple function that parse the input command string into lists of arguments. **strtok ()** library functionisused in this case by considering simplewhite space as a delimiter in split**\_stringLine ()** functiontoreturn pointers to within the strings of command and placing ‘\0’ byte at the end of each token. We then store each pointer in an array of character pointers. The process repeats until no token is returned by strtok, which we null-terminate the list of tokens.

For our shell process get started on the shell **mysh\_launch ()** function was implemented which starts by forking a child process, where the child process returns its process id to the parent process. Since we want to run the commands given by users who run ***mysh*** we need to use an exec system call so that it lets the child process not to return from the exec call unless there is an error. Out of available exec system calls we used **execvp()** as it takes an array of string arguments. It is also preferred because it does not require the full path of a program argument instead the operating system search the program in the path. In this function if there is an error in forking it catches the error in the second condition shown in function and returns by displaying an error message. The last condition shows a case where the fork executes successfully and the parent process has to wait for the command to finish running. So we used waitpid() for the process’s state to change.

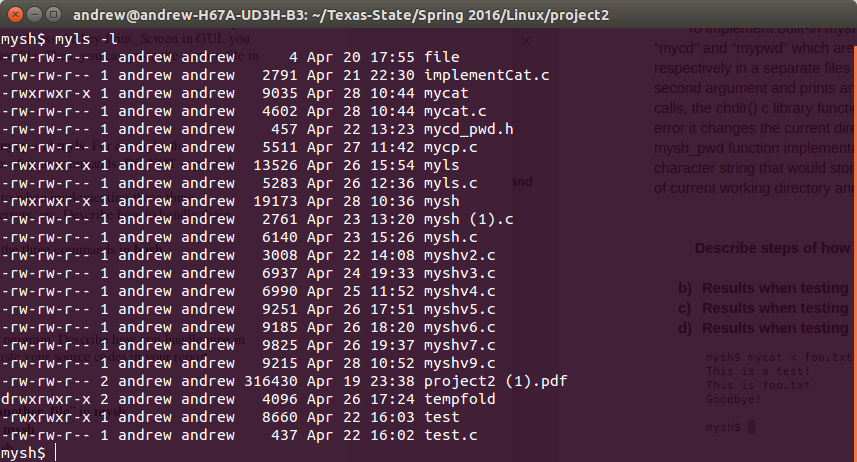
To make a built in command to our shell program we first created an array of built-in command names followed by an array of their corresponding functions which is an array of function pointers that takes array of strings and returns an integers. Finally, mysh\_execute() function was implemented which checks if the command equals each built-in and if so it runs it. If it does not match it calls a mysh\_launch() function to launch the process. It also checks If the user by chance hits the key without supplying arguments to return.

To implement built-in mysh commands we first created an implementation function for the “mycd” and “mypwd” which are comparable shell built-in command to “cd” and “pwd” respectively in a separate files “mycd\_pwd.h”. The mysh\_cd () function first checks existence of second argument and prints an error message if it doesn’t exist. If there is a second argument it calls, the chdir() c library function which checks for existence an error and return. If there is no error it changes the current directory to the users supplied directory and returns. In the case of mysh\_pwd function implementation we followed a simple procedure by first declaring a variable character string that would store character of string of paths we get by calling getcwd() function of current working directory and then using printf() function prints it on terminal.

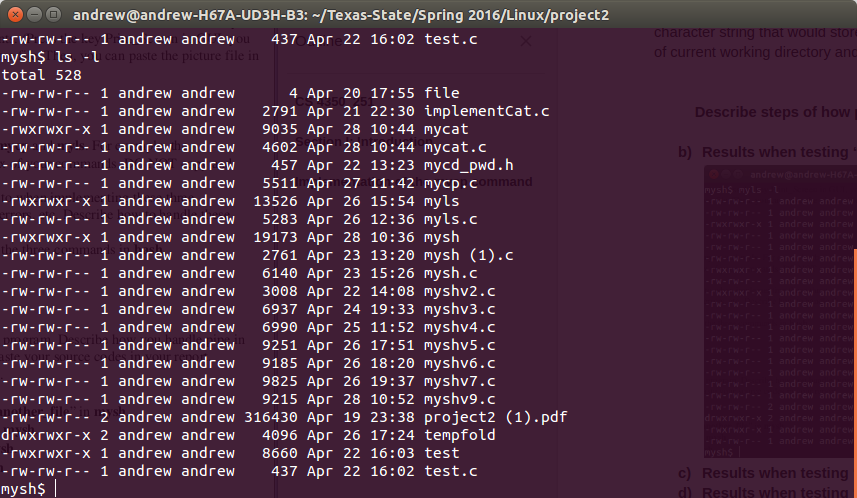
**Describe steps of how piping was handled in the shell.**

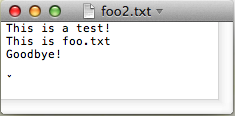
Piping was handled within the parse\_programs function in myshv9.c. If the programs being executed are empty, an error message is displayed saying their was an allocation error and exits the function. The function then checks for if ‘|’ or ‘$(“ is present, and performs the corresponding operations for each case. Within the code for these cases, the token variable is assigned to the program variable for handling the piping. If successful, the variable program is returned to be executed by the shell. Commands that used piping, such as myls | mycat, the code for those commands were not built into the cmd\_built and func\_built arrays. This was done so that both parameters of argc and argv could still be passed to the files. When they were built into the shell code, only the arguments for argv could be passed.

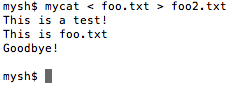
1. **Results when testing “myls -l” in mysh:**



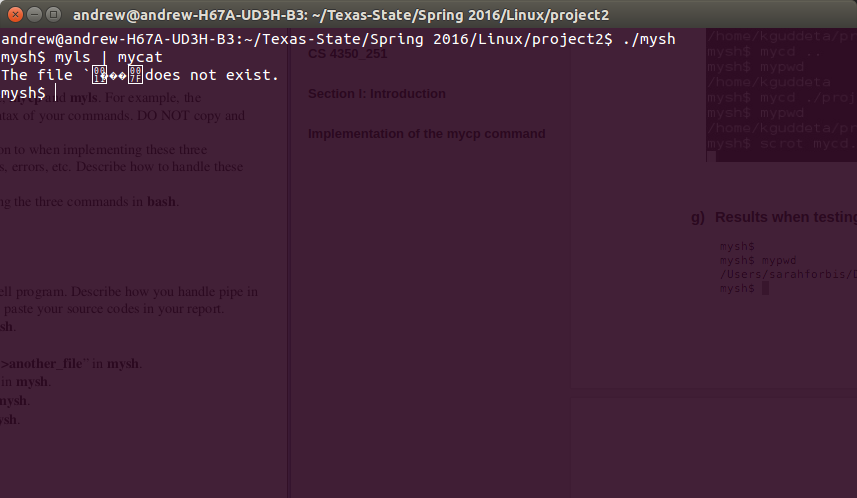
1. **Results when testing “ls -l” in mysh:**

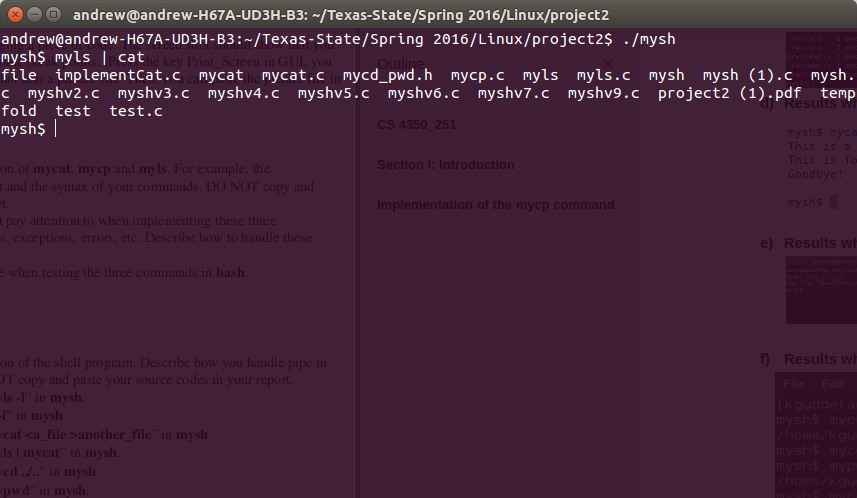


1. **Results when testing “mycat < a\_file > another\_file” in mysh:**

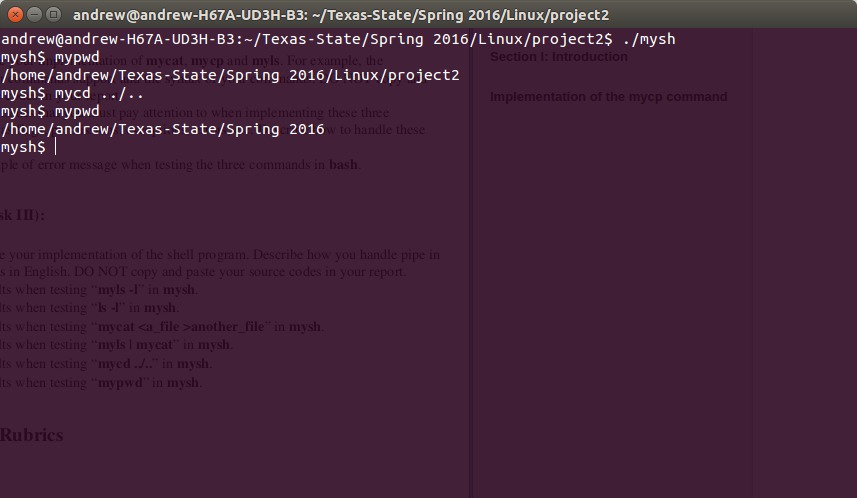


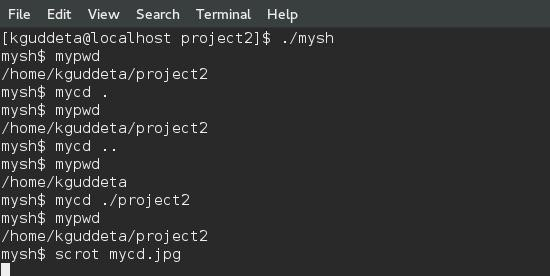
1. **Results when testing “myls | mycat” in mysh:**





1. **Results when testing “mycd ../..” in mysh:**





1. **Results when testing “mypwd” in mysh:**

