**Project 2 Report**

**By:**

**Michael Weikman**

**Matt Wallace**

**Trevor Mendenhall**

**Daniel Brooks**

**Section I**

In an effort to increase communication and ease collaboration we used a git repository for our project. Things did come together much smoother than our last project, despite the end of semester time crunch.

Michael Weikman wrote the myls utility and detailed it in section 3. Trevor Mendenhall wrote the mycp utility and detailed it in section 3. Daniel Brooks wrote the mycat utility and detailed it in section 3. Matthew Wallace wrote the shell itself and detailed it in section 4. Integration testing of the utilities was also performed by Matthew, he also wrote sections 1 and 2 of the report. Michael Weikman handled combining the separate parts of the report into a consistently formatted report for submission.

**Section II**

mysh makefile:

CC = gcc

LDFLAGS = -lreadline

CFLAGS = -pedantic -g -std=c99

DEPS = strings.h shell.h

OBJ = mysh.o

EXEC = mysh

.PHONY: clean

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS) $(LDFLAGS)

../$(EXEC): $(OBJ)

$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

clean:

rm -f \*.o

myls makefile:

CC = gcc

LDFLAGS = -lm

CFLAGS = -Wall -pedantic -g -std=c99

DEPS = myls.h

OBJ = myls.o print.o util.o

EXEC = myls

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CLFAGS) $(LDFLAGS)

../$(EXEC): $(OBJ)

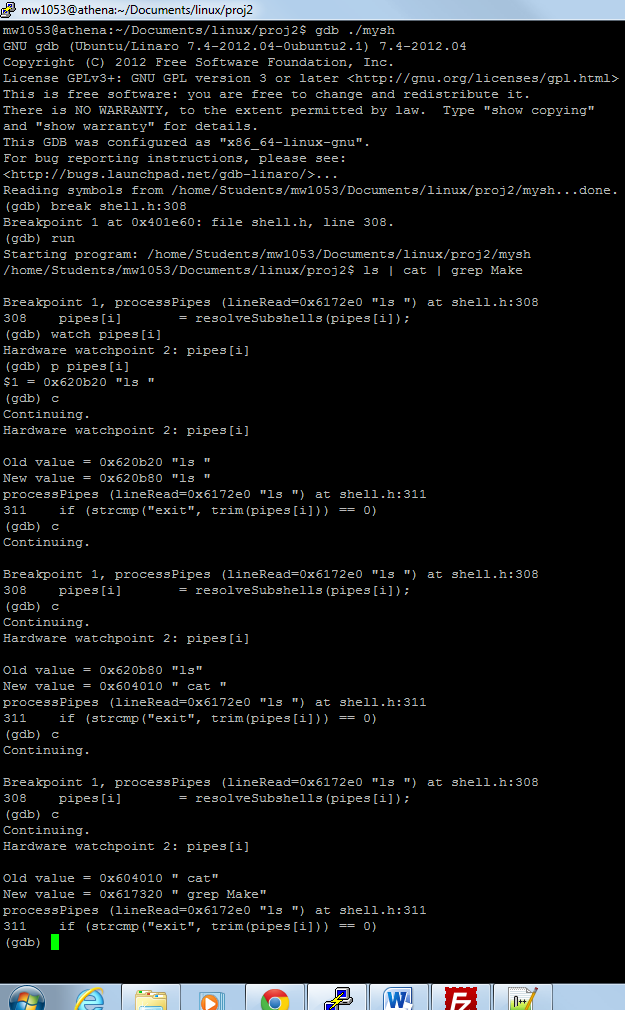
$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

.PHONY: clean

clean:

rm -f \*.o

Tracing as the shell deconstructs a chain of pipes:



**Section III:**

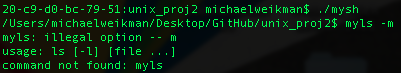
**Myls**

The myls function accepts one option ‘-l’. When executing myls without option it will list the files/folders of the current directory in row column format similar to ls. The files/folders are listed in alphabetical order in a cyclic orientation based on position in the array structure. I used FTSENT functions to parse provided or current (if not provided) directory information and store the content information in the returned structure. My program is dependent on two .c files print.c and util.c and one header file myls.h. The two .c files contain functions related to the printing and formatting of the list of files while the header files contains the relevant prototypes and extern variables that all files are dependent on. FTSENT structures contain all information required to perform the operations of this command, including file name, file name length, stat information, and error information. I created a primary struct in the header file FINFO to hold length (character space) of particular attributes of the file to format the long format listing to reflect that of the original ls function. The util.c contains functions needed to perform actions needed to gather length of numbers (character space) and translating st\_mode to readable permission format (i.e \_r\_w\_x).

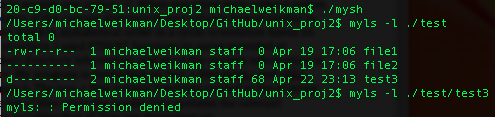
Essentially, how the command works with respect to the code, is the command is executed with or without an option, the option of which determines the printing format. If no option is provided the default print is row column format of just file names and if the option –l is provided it prints single column format with file attribute information similar to the normal ls command. If a single file is provided as a parameter to the command it will print just that file information based on the option selected. If a directory is provided it will traverse the directory tree and print files in that directory. If not directory or file is provided it defaults to the current directory and prints all files and folders in the current directory in the format selected by the provided or non-provided option. In the case of directories if the permissions of the directory don’t allow traversal or read access of the directory the command outputs error message explaining permission related error. Parent directory ( . . ) is ignored to prevent traversing parent directories. Hidden files are also ignored due to the project not requiring the displaying of them.

Two issues that need to be handled when handled when implementing this function are permissions of directory folders that may be traversed for listing and the use of an option that is not of the list of options available by the command. To handle the permissions issues, I kept track of return errors from functions used to traverse directory trees and used functions in the err.h library to print the errors and corresponding error message. For the option handling, I created a usage function that is called when any option other than –l is used. The usage function prints out to stderr the reason for the error and the usable option information for the myls command.

Invalid option error



No traversal permission for directory



**Mycat**

The Linux cat command is a very simple piece of functionality that is designed to do one thing very well - that is, displaying the contents of a file to the terminal, and concatenating multiple files in the case of multiple arguments. Additionally, cat may operate with zero arguments, in which case it will take its input from the keyboard. cat may also create new files, append to existing files, and provide input to subsequent commands by using the '>', '>>', and '|' operators, respectively. The core of our implementation involves iterating through each argument passed to the function. For each argument, a number of error checks are performed, and the file is then opened and read character by character, printing each character to the standard output stream until the end of file condition is reached. The file stream is then closed, and the program iterates to the next file to be read, or terminates if there are none remaining. In the case that no arguments are passed to the function, the program allocates a 256 byte character buffer and reads from standard input line by line using the getline() method. This continues until it is interrupted by a signal from the terminal.

The most basic error occurs when a specified file is not found in the provided location. In this case, the fopen() function will return a NULL value to our file pointer, which signals the process to print an error message and continue to the next file in sequence. Another potential issue that our program must check for is the type of file provided by the user. Specifically, if the file is a directory, the cat command will not attempt to open a stream or read from it. While a directory is indeed a file containing a list of contents and their inode numbers, it is not intended to be read from in most cases. Our program uses the stat functionality in order to read in a structure of file features, allowing us to confirm if the file is a directory. If this is the case, the program will not terminate, but rather display an error message to the terminal and iterate to the next argument.

Error Message for cat of non-existent file

Macintosh HD:Users:michaelweikman:Desktop:Screen Shot 2013-04-23 at 6.07.02 PM.png

**Mycp**

Functionality

This program emulates the actions of the cp shell command. The program’s file copy functionality expects two arguments in the form './mycp.exe a\_file b\_file' it will then copy the contents of the text file "a\_file" to the file "b\_file". The program will create b\_file in the current directory, if it has not yet been created. The program will prompt user for permission if b\_file is to be overwritten.

The programs directory copy functionality expects arguments and option in the form './mycp.exe -R a\_dir b\_dir`. The function will then traverse the directory structure recursively copying all sub folders/directories. The program expects rwx permission on the two directories.

Expected Conditions

The user shall not pass command arguments in form other than (mycp.exe a\_file b\_file and mycp.exe -R a\_dir b\_dir ). a\_file and a\_dir must exist before the function is called. a\_file is expected to have “r” permission while a\_dir is expected to have “wx”. b\_file does not have to previously exist, however if it does it must have “w” permission enabled. Simillarly b\_dir does not have to previously exist, however if it does it must have “rx” permissions.

Implementation

The copy file functionality opens both files and copies their contents line by line. The b\_file will be created if not already. The function will fail if a\_file doesn’t have read permission. The amount of data to be read in per line is a limitation, see Limitation section for further detail.

The cpdir function uses a while loop read in files contained in a\_dir and then create them in the b\_dir. The contents of the files are then copied over one character at a time. If the object read from a\_dir is its self a directory the cpdir function is called recursively with the new directory pointer passed to it. The opendir() and readdir() functions are used to work with the directory structures.

Limitations

The fgets() function used to copy files line by line is only allotted 1100 character spaces to allocate the information. This is larger than the max line size in my system, however if moved across systems user should be aware of this limitation. The cpdir() function saves the current path of the program and the relative path of the a\_dir directory’s tree structure. The total amount of space allocated for this is 256 characters. My current system has a max path length of 246 characters. If the program is moved across systems user should be aware of this limitation.

Error message for trying to copy from a file with no read permissions (file to file2)

Macintosh HD:Users:michaelweikman:Desktop:Screen Shot 2013-04-23 at 6.12.32 PM.png

**Section IV**

The main function of the shell is an infinite loop that grabs a command from the user using GNU readline and then passing that command to the processPipes function which handles all the execution from that point on. Once processPipes returns the memory allocated by readline is freed and the loop runs again. If a command is passed when mysh is invoked (e.g. `./mysh ls | cat`) the command passed as an argument is processed and then the loop terminates allowing the shell to exit.

The processPipes function splits the command on the ‘|’ char creating an array of commands. This array of sub commands is then lopped over to get the final output. If at any time the current command is `exit` then the shell immediately terminates. Processing of each sub command entails resolving subshells to get the actual command to be executed using the resolveSubshells function and then executing and capturing the output using the exec function. For each sub command after the first the output from the previous command is passed to exec to be used as the input for the next, allowing the chaining of pipes. If exec indicates an error then proper output is generated and the loop breaks so that no further commands in the chain of pipes are executed. Once the last command in the chain is executed the final output is printed to the terminal unless there was a redirection of output on that item.

The resolveSubshells function searches for (‘$(’, ‘)’) pairs and extracts the enclosed command. The preceding substring and remainder substring are saved for later use. The extracted command is execed and the output is captured. The preceding substring, the output from the subshell, and the remainder are glued back together to get the new string. This process is looped until no more subshells are detected in the command. This allows for resolution of multiple subshells but will not work for nested subshells. The resulting command after the loop has exited is returned as the new command.

The exec function first parses any redirections out of the command using the getRedirections function. Once that is done the command is broken on spaces so that we have an array of cstrings. The function is forked.

* In the child:

If previous input was provided or there is an input redirect then a pipe is created. First the previous input (if any) is written to the write end of the pipe then if there is an input redirect the input file is opened and data is read from it and written to the pipe and the read end is duped to stdin. If there is only an input redirect then the file is opened and the fd for that file is duped over stdin. If there is any error redirect then that file is opened and duped over stderr. A second pipe (created before the fork) has its write end duped over stdout so that the parent can see the results. The command is then execed using execvp.

* In the parent:

The parent waits for the child to complete and checks the return status. If it is non-zero then the proper errors are produced. The output from the child is read from the pipe into a buffer. If there was an output redirection then the file is opened and the output is written to the file. The result is still made available to the caller so that the output can be reused if there was a pipe as well as the redirection.

The resolveSubshells function loops while searching for any redirection delimiters. Each loop only the first delimiter is processed and the loop exits when no more redirection delimiters exist in the string. The name of the file redirected to is saved and the delimiter and the file name are removed from the command, overwritten by the remainder that came after it. If any redirection is specified multiple times it will be overwritten and only the last one matters. Once the loop exits the 3 redirections (NULL is used if none was specified) are made available to the caller for use.

**`myls –l`:**

/home/Students/mw1053/Documents/linux/proj2$ myls -l

total 304

-rw------- 1 mw1053 student 336 Apr 21 22:53 Makefile

-rwx------ 1 mw1053 student 12323 Apr 23 14:05 mycat

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:05 mycat\_src

-rwx------ 1 mw1053 student 18231 Apr 23 14:05 mycp

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:05 mycp\_src

-rwx------ 1 mw1053 student 14167 Apr 23 14:05 myls

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:05 myls\_src

-rwx------ 1 mw1053 student 25954 Apr 23 14:29 mysh

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:40 mysh\_src

-rw------- 1 mw1053 student 169 Apr 10 13:03 README.md

drwx------ 2 mw1053 student 4096 Apr 23 14:18 test

/home/Students/mw1053/Documents/linux/proj2$

**`ls –l`:**

/home/Students/mw1053/Documents/linux/proj2$ ls -l

total 152

-rw------- 1 mw1053 student 336 Apr 21 22:53 Makefile

-rwx------ 1 mw1053 student 12323 Apr 23 14:05 mycat

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:05 mycat\_src

-rwx------ 1 mw1053 student 18231 Apr 23 14:05 mycp

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:05 mycp\_src

-rwx------ 1 mw1053 student 14167 Apr 23 14:05 myls

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:05 myls\_src

-rwx------ 1 mw1053 student 25954 Apr 23 14:29 mysh

drwxr-xr-x 2 mw1053 student 4096 Apr 23 14:40 mysh\_src

-rw------- 1 mw1053 student 169 Apr 10 13:03 README.md

drwx------ 2 mw1053 student 4096 Apr 23 14:18 test

/home/Students/mw1053/Documents/linux/proj2$

**`mycat <Makefile >out`:**

/home/Students/mw1053/Documents/linux/proj2$ mycat <Makefile >out

/home/Students/mw1053/Documents/linux/proj2$ mycat out

SHELL\_FOLD = mysh\_src

CAT\_FOLD = mycat\_src

CP\_FOLD = mycp\_src

LS\_FOLD = myls\_src

.PHONY: clean run mysh mycat mycp myls

all: mysh myls mycat mycd

mysh:

$(MAKE) -C $(SHELL\_FOLD)

mycat:

$(MAKE) -C $(CAT\_FOLD)

mycd:

$(MAKE) -C $(CP\_FOLD)

myls:

$(MAKE) -C $(LS\_FOLD)

clean:

rm -f mysh mycat mycp myls

run:

@./mysh

/home/Students/mw1053/Documents/linux/proj2$

**`myls | mycat`:**

/home/Students/mw1053/Documents/linux/proj2$ myls | mycat

Makefile mycat\_src mycp\_src myls\_src mysh\_src README.md

mycat mycp myls mysh out test

/home/Students/mw1053/Documents/linux/proj2$