# CS 4414: Operating Systems – Spring 2018
# Machine Problem 4: Writing a Simple Shell
Cassie Willis

## Completion Level:

This project works to the full extent that was tested by the developer with a single exception. This project cannot handle an input redirection and output redirection both in the same token group when there are no pipes present. The code will process correctly the first redirection given but will not process the second redirection. It should process all other inputs as tested regardless of the number of pipes, however.

## Problem Description:

The goal of this assignment was to develop a program that uses forking, pipes, and parent/child processes to develop a simple shell. This shell has the ability to open programs and redirect program input and output originally to standard out (stdout) and from standard in (stdin) to instead come from and to files. The program also has piping capability, which redirects the output of the first program into the input of the second program. Input for the shell was given in lines, with lines separated by pipes into token groups. Each token group included a single command, which was the program or executable to be run. The token group then can additionally include arguments for a program, a redirection, and the file to redirect to or from. These last three items are not necessarily required, depending on the command to be executed. Each token group is run in a child process, that is created using the fork() command. The child processes are completed in the order listed on the command line, and pipes are set up between each token group to pipe each of the commands together. The output of the final command after all piping has completed is sent to the shell's standard output, and the appropriate error code determining whether the child process exited normally is sent to standard error for each child process.

## Approach:

The approach to developing this shell was to break some of the functionality into individual components but to keep the majority of the code needed for the child process in the main method itself. This was done because child processes all need their own unique variables that cannot be touched by other child processes in order to execute properly, and adding additional functions ran the risk of these variables not being kept local to the child if the function was developed incorrectly. This design decision also aided with making clearer the exact code that was in the child process and what was happening in each child. After deciding how to split up the functions, design decisions had to be made on how to execute the commands. The execv() function was chosen for command execution because this function requires only the command name as a const char* type and a char** type argument array. The argument array includes the command name at the beginning and a NULL terminator. Requiring only two arguments, execv was considered a simple execution function to implement and thoroughly understand. The remainder of the execution is described in the subsections below.

### get_input():

This function handles getting the input from each line of the shell and parsing the input to ensure the input consists of only valid commands, characters, and spacing. The function is called

at the very beginning of the while(1) loop in main, and so will continue to execute for every line until an exit command or a newline is reached. If an exit command is reached, the function sets a global Boolean variable called "endOfInput" to true, which is used later in the main function. The first command in get_input() sends the shell prompt ('>') to stdout. The function then uses getline() to receive a line from stdin and add it to an input string. Then error checking is completed. The following errors are checked for against the input string in this function:

1. Ensure the size of the input string is not greater than 100
2. Ensure the command is not empty, exit, a single space, or EOF (end of file)
3. Ensure all of the characters are valid (one of eight special characters or a letter – upper or lowercase - or number)
4. Ensure the first and last words are not pipe or redirection operators
5. Ensure operations are not connected to words (i.e. cat|)
6. Ensure input file redirect does not occur after a pipe operation
7. Ensure output file redirect does not occur before a pipe operation

If the input string is verified to be a valid input (and a global Boolean 'valid' is set appropriately), the string is split into two different global string vectors. The first, splitString, splits the input up by spaces so each token, command, word, operator is its own string in the vector. The second, partial, splits the input string into token groups, separated by the pipe operator. The number of token groups is also counted in the function and set to a global 'commands' integer variable. Because all of the variables being set in get_input() are global variables, this function does not have a return value.

**clearVector():**

This function clears the global variables that are used in each child so that after the child finishes execution the new line from the shell starts with clean variables. The string vectors splitStrings and argsList are cleared, valid is reset to true, the currCommand is set to NULL, and the commands integer is reset to zero. This function is called at the very end of the while loop in main() and does not have a return value.

**createFile(const char* filename):**

This function is used in the child process if an output redirection is found in the shell input string and there does not yet exist a file with the name given after the output redirection operator. This function creates a new file with that name. A FILE* type is created and fopen() is used to create the file as both a read and write. The file is created as read and write in case the user wants to read from the file at a later time, however in the output redirection the file is only opened for reading. If the file is created properly, the file is closed using fclose() so that it can be reopened in the child with an integer file descriptor. If the file could not be created, an error statement is printed to the shell. This function has no return value.

**main():**

The main function handles the pipe creation, duping for both pipes and redirects, forking into child processes, process execution, and error code printing. The entire main function is housed inside a while(1) loop in order to ensure the shell continues to receive additional input lines until the exit or end of file commands are encountered.

Inside of the while loop, the main function first calls get_input() to get the command line of the shell and check for valid input, end of shell, and set up the two string vectors as described above. Main() then checks the endOfInput Boolean, and if the Boolean is true the function breaks out of the while loop and exits the shell. The function then checks if the Boolean valid is true. If it is not true, the function prints invalid input to stdout and reads the next line. If the input is valid, however, the piping and forking can be completed. The pipes are created as a two-dimensional array, pipes[x][y]. Here, x is the number of token groups, set by the global commands integer, minus one to represent the number of pipes. The value for y is always equal to two because each pipe has both an write end and a read end. To create the pipes, the pipe() command is called on each pipe iteratively in a for loop. If the pipe could not be created, invalid input is printed to stdout. After the pipes are created, a for loop calls the fork command for every token group. An invalid input is returned if the fork could not be created. From there, the child and parent processes as described below.

Child (pid == 0):

The majority of the shell functionality is inside the child process. A child process is created for every token group, and the child also handles piping, which has three cases. The primary case is where the process pipes the output of the current command to the next pipe's write end and reads the input from the previous process from the read end of the previous pipe, then closes the read and write ends of both of the pipes used and the write ends for all pipes after the current pipe. If the process is for the first token group, only the write takes place and the proceding pipe is closed on both ends and the write end is closed for all pipes. If the process is for the last token group, only the read takes place and the preceding pipe is closed at both ends.

After piping using the dup2() function, the child process creates two local vector strings, temp and beforeRedirect. The former holds the entire argument list for the current token group, which is then placed into a const char* array args[]. The latter holds the argument list up until a file redirection symbol, as the file redirection should not be included in the argument list for execv(). This string vector is then placed into a const char* array argsbeforeRedirect[]. Both of the const char* arrays have NULL added to the last element of the arrays, which is needed for execv().

The args array is then used to determine if a redirection occurs. Recall that the get_input function ensures that redirection before or after a pipe does not occur when not allowed, so this did not need to be handled in the child process. A combination of a for loop and an if statement determine if an output or input redirection occur by comparing every element of args to the redirection operators. If an input redirection occurs, the current working directory is attached to the file name, and the file is opened read-only with the open() command. An invalid input is printed to stdout if the file cannot be opened for reading. If an output file redirection occurs, the cwd is attached to the file name and access() is used to determine if the file exists. If the file does not exist, the createFile() function is called on the file name. The file is then opened for writing, and if it cannot be opened prints invalid input to stdout. Dup2 is again used to redirect from stdout to the file or from stdin to the file, depending on the redirection present, and the file is closed.

With file redirection handled, the child process then checks whether the path given for the command is relative or absolute. Absolute commands are set to a string called fullPath, while relative commands have the cwd attached to them and then are set to fullPath. Finally, the execv() command is used to execute the command in the current token group, with the fullPath

used as the command name and argsBeforeRedirect set as the argument list. The child uses _exit(-1) to exit from the process, with -1 used in the parent as an error code.

Parent (else):

The parent process, while still in the for loop, closes all of the read ends of all of the pipes as a hygienic move. It then waits on the child to complete using waitpid(), and uses the value of the exit status "WIFEXITED" to either print the exit status code of the command, saying that the child exited normally or with an error, or prints invalid input. If an exit code is printed, this is sent to stderr instead of stdout.

Finally, outside of the for loop, indicating all children are finished running, the parent closes all of the pipes and uses clearVector() to clean everything for the next line of shell input. All global variables and include statements, as well as function declarations, are inside a header file, which is included at the beginning of the C++ file.

## Results:

The program is able to successfully receive input from the shell prompt and give output based on the commands given on the input line. The program parses through the input and gives invalid input to stdout if the line given is not valid. The program can execute the commands properly for each token group with all of the arguments for the token group, including the appropriate file redirection. The token groups execute in the order given to the command prompt and the pipes correctly send data to the correct processes. The exit code for each child is given on stderr, and any other exit condition besides an exit code is sent to stdout as "invalid input".

The program was tested thoroughly by the author. The author tested with a single command both relative and absolute paths. A single command was also tested with a single output or a single input redirection. A command was tested with an input and an output redirection, which unfortunately is the only token group known to the author not to function properly. After single commands were thoroughly tested, the author tested with pipes. Pipes were tested both with and without arguments and with and without file redirection. Both single pipes and multiple pipes were tested to ensure complete functionality of the end and middle token group cases, all of which were shown fully operational in testing.

## Analysis:

As evident by the near-complete functionality, to the author's knowledge, of the shell implementation code, this project was mostly a success. This finding was particularly important to the author as previous machine problems have not worked properly, however, the author feels as though she has a greater understanding of forking, child versus parent processes, and piping.

Though some errors occurred in developing this program, the author also learned a great deal from this machine problem. With this assignment, the author was able to more deeply understand pipes, multiprocessing, and forking. The author also learned more about the C++ programming language.

The author did run into a few errors in creating this solution. The biggest setback was in attempting to close the correct pipes. The pipes were mainly closed correctly, however it was not initially realized that the write ends of pipes after the current one have to be closed in the child and the read ends of all the pipes should be closed in the parent after every child. This error actually aided the author in a better understanding of how pipes work, which the author may not

have fully realized had the pipes been implemented correctly in the first attempt. Once the author's pipes were implemented, one last problem was encountered where the program was hanging and not completing execution. This issue turned out to be the parent not fully closing all of the pipes after the children had completed execution.

Finally, the author continued to learn about time management strategies and planning. This project was completed over a greater period of time, which caused less stress and less feeling of a time crunch. The project was also mapped out on paper before any code implementation. Each of the new functions were pseudo-coded to ensure that the author had a clear path of how to implement the functions in the C++ programming language, and the author took advantage for the first time of TA and professor office hours, which was a significant help in completing the project.

## Conclusion:

Pipes and forking are by no means simple to understand, however, because of this machine problem the author has a much better understanding of how they operate and how they are essential to programs that require running processes concurrently or to process subprograms while a main program is still executing. It is interesting to see how pipes and redirects are used in even very simple shell programs, however, it is also clear that they could be used for many very robust and complex implementations as well. Multiprocessing is also very interesting because speed and efficiency, as well as the ability to keep a program running in the background, are always desired, and in many cases critical, particularly in real time operating systems such as many embedded applications. This is of particular interest to the author as a computer engineer, as embedded systems are the author's first choice for future career paths. It was useful to take a hands-on approach to piping, redirection, and forking in order to more thoroughly understand it, as the author feels as though much was learned through this process.

## Pledge:

On my honor as a student, I have neither given nor received aid on this assignment.

Cassie Willis

# Code Dump:

### MP4.cpp

```
/* Cassie Willis
 *
 * CS 4414 - Operating Systems
 * Spring 2018
 * 4-18-18
 *
 * MP4 - Writing Your Own Shell
```

```
 *
 * The purpose of this project is to become familiar with shell scripting.
 * This project recieves input from STDIN and appropriately performes file
 * opening or piping operations from |, <, and > operators, returning either
 * return information to STDOUT or exit codes to STDERR
 * Refer to the writeup for complete details.
 *
 * Compile with MAKE
 *
 */

// > should send STDOUT to a file, create the file if it doesn't exist, error if cannot create
// < should send from file to STDIN, error if file cannot be read
// | redirects STDOUT of first to STDIN of second
//wait for subprocesses to finish and print return values to STDOUT, one per line in order listed

#include "MP4.h"
using namespace std;


//Parse a single line of input into a vector of strings
void get_input() {
	string input;
	cout << ">";
	fflush(stdout);
	getline(cin, input);

	//Make sure input is not too long
	if (input.size() > 100) {
		fprintf(stdout, "invalid input\n");
		valid = false;
	}

	//Check if end of input and can exit shell
	if (input.empty() || input == "exit" || input == "Exit" || input == "EXIT" || input == " " || input ==
		"EOF") {
		endOfInput = true;
		return;
	}

	//Make sure the input contains only valid characters
	int size = input.size();
	int i;
	for (i = 0; i < size; i++) {
		if(!isalnum(input[i]) && input[i] != '|' && input[i] != '>' && input[i] != '<') {
			if(input[i] != ' ' && input[i] != '/' && input[i] != '.' && input[i] != '-' && input[i]
			!= '_') {
				fprintf(stdout, "invalid input\n");
				valid = false;
				break;
			}
```

```
        }
}

//Take the input string and separate by spaces into a vector of strings
istringstream iss(input);
for(string input; iss >> input; ) {
        splitStrings.push_back(input);
}

//Make sure first and last inputs are words not operations
if(splitStrings[splitStrings.size()-1] == "|" || splitStrings[splitStrings.size()-1] == ">" ||
        splitStrings[splitStrings.size()-1] == "<" || splitStrings[0] == "|" || splitStrings[0] == ">" ||
        splitStrings[0] == "<") valid = false;

//Make sure operations are not connected to words
for(unsigned int j = 0; j < input.size(); j++) {
        if (((input[j] == '|') || (input[j] == '<') || (input[j] == '>'))
                && ((input[j-1] != ' ') || (input[j+1] != ' '))) {
                valid = false;
        }
}

//Check to make sure < and > do not come before/after pipes where not allowed
// < input file redirection cannot have a pipe before it
// > output file redirection cannot have a pipe after it
for(unsigned int x = 0; x < input.size(); x++) {
        if(input[x] == '<') { //check for pipe before <
                for(unsigned int e = 0; e < x; e++) {
                        if(input[e] == '|') {
                                valid = false;
                        }
                }
        }
        else if(input[x] == '>') { //check for pipe after >
                for(unsigned int e = x+1; e < input.size(); e++) {
                        if(input[e] == '|') {
                                valid = false;
                        }
                }
        }
}

for (unsigned int k = 0; k < splitStrings.size(); k++) {
        if (splitStrings[k] == "|") commands++;
}
commands ++;

//Separate commands/arguments in a line
stringstream in(input);
string partial;
while(getline(in, partial, '|')) argsList.push_back(partial);
```

```
}

//After each line is processed, clear the vector to process the next line
void clearVector() {
	splitStrings.clear();
	argsList.clear();
	commands = 0;
	valid = true;
	currCommand = NULL;
}

//create a new file
void createFile(const char * filename) {
	FILE *newFile;
	newFile = fopen((const char*)filename, "w+");
	if(newFile == NULL) fprintf(stdout, "invalid input\n");
	else fclose(newFile); //create new file and check if created
}

int main() {
	while(1) {

		get_input();

		//If end of input, exit shell
		if(endOfInput == true) {
			break;
		}

		if(valid == true) { //input line is valid input
			//create pipes
			int pipes[commands-1][2]; //Pipes = #token groups - 1
			for (int y = 0; y < commands-1; y++) {
				if((pipe(pipes[y])) < 0) fprintf(stdout, "invalid input\n");
			}

			//create child processes
			pid_t pid;
			for(int p = 0; p < commands; p++) {
				processID = p;
				pid = fork();
				//fork not successful
				if (pid < 0) {
					fprintf(stdout, "invalid input\n");
					return -1;
				}

				//child process
				else if(pid == 0) {
					bool fileOutput = false; //checks if file redirection is required
					bool fileInput = false;
```

```
//Piping dup2 - curr token group = i, read from pipe i, write to
pipe i-1
//STDIN = 0       /  STDOUT = 1
//If first command, write to i
if(p == 0) {
        dup2(pipes[p][1], 1);
        close(pipes[p][0]);
        close(pipes[p][1]);
        for(int l = p; l < commands-1; l++) close(pipes[l][1]);
}
//If last command, read from i-1
else if(p == commands-1) {
        dup2(pipes[p-1][0], 0);
        close(pipes[p-1][0]);
        close(pipes[p-1][1]);
}
//If middle command
else {
        dup2(pipes[p][1], 1);
        dup2(pipes[p-1][0], 0);
        close(pipes[p][0]);
        close(pipes[p][1]);
        close(pipes[p-1][0]);
        close(pipes[p-1][1]);
        for(int l = p; l < commands-1; l++) close(pipes[l][1]);
}

//Each argument in token group into char array
vector<string> temp;
string tempArg = argsList[p];
istringstream iss(tempArg);
for(string tempArg; iss >> tempArg; ) {
        temp.push_back(tempArg);
}
int vSize = temp.size();
const char* args[vSize];
for(int i = 0; i < vSize; i++) {
        args[i] = temp[i].c_str();
}
args[vSize] = NULL;

//Each argument in token group before redirect into char array
string tempA = argsList[p];
istringstream isa(tempA);
vector<string> beforeRedirect;
for(string tempA; isa >> tempA; ) {
        if((tempA.compare(">") != 0) && (tempA.compare("<")
        != 0)) {
                beforeRedirect.push_back(tempA);
        }
```

```
            else break;
    }

    int rSize = beforeRedirect.size();
    const char* argsBeforeRedirect[rSize];
    for(int i = 0; i < rSize; i++) {
            argsBeforeRedirect[i] = beforeRedirect[i].c_str();
    }
    argsBeforeRedirect[rSize] = NULL;

    //Open files for redirection
    for(int oc = 0; oc < vSize-1; oc++) {
            //output - open as write, need to check if exists
            if(strcmp(args[oc], ">") == 0) {
                    fileOutput = true;
                    char cwd[100];
                    getcwd(cwd, 100);
                    string tempPath = args[oc+1];
                    if(args[oc+1][0] != '/') {
                            tempPath = string(cwd) + '/' +
                            args[oc+1];
                    }
                    //file does not exist - create file
                    if(access(tempPath.c_str(), F_OK) == -1) {
                            createFile(tempPath.c_str());
                    }
                    fileO = open(tempPath.c_str(), O_WRONLY);
                    if(fileO < 0) {
                            fprintf(stdout, "invalid input\n");
                            return -1;
                    }
                    break;
            }
            //input - open as read
            if(strcmp(args[oc], "<") == 0) {
                    fileInput = true;
                    char cwd[100];
                    getcwd(cwd, 100);
                    string tempPath = args[oc+1];
                    if(args[oc+2][0] != '/') {
                            tempPath = string(cwd) + '/' +
                            args[oc+1];
                    }
                    fileI = open(tempPath.c_str(), O_RDONLY);
                    if(fileI < 0) {
                            fprintf(stdout, "invalid input\n");
                            return -1;
                    }
                    break;
            }
    }
```

```
                //For commands: If path is not absolute, get cwd and attach to
                        path (command at beginning of each pipe)
                char cwd[100];
                getcwd(cwd, 100);
                string fullPath;
                if(args[0][0] != '/') {
                        fullPath = string(cwd) + '/' + temp[0];
                        temp[0] = fullPath;
                }
                else {
                        fullPath = temp[0];
                }

                //Run the command
                //dup2(new, old) - redirect file output, 1 = stdout
                if(fileOutput) {
                        fileOutput = false;
                        dup2(fileO, STDOUT_FILENO);
                        close(fileO);

                }
                if(fileInput) {
                        fileInput = false;
                        dup2(fileI, STDIN_FILENO);
                        close(fileI);
                }

                currCommand = temp[0].c_str();
                execv((const char*)fullPath.c_str(),
                        (char**)argsBeforeRedirect);
                _exit(-1);
        }

        //parent process, wait for children to end
        else {
                //close all read pipes
                for (int y = 0; y < commands-1; y++) {
                        close(pipes[y][1]);
                }

                //wait for child to finish, print exit code
                int status;
                waitpid(pid, &status,0);
                if(WIFEXITED(status) > 0) {
                        fprintf(stderr, "%d\n", WEXITSTATUS(status));
                }
                else fprintf(stdout, "invalid input\n");
        }
}
```

```
                        //close all pipes
                        for (int y = 0; y < commands-1; y++) {
                                for(int s = 0; s < 2; s++) {
                                        close(pipes[y][s]);
                                }
                        }

                }

                else {
                        fprintf(stdout, "invalid input\n");
                }

                clearVector();
        }

        return 0;
}
```

## MP4.h

```
#ifndef __MP4_H__
#define __MP4_H__

#include <iostream>
#include <string.h>
#include <vector>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fstream>
#include <algorithm>
#include <fcntl.h>

using namespace std;

//Function Declarations
extern void get_input();
extern void clearVector();
extern void createFile(string filename);

vector<string> splitStrings;
vector<string> argsList;
bool valid = true;
bool endOfInput = false;
int commands = 0;
```

```
int fileO;
int fileI;
const char * currCommand;
int processID;

#endif
```