Brian Team
dot4qu

CS 4414 Operating Systems - Homework 1: Writing Your Own Shell

Pledge: On my honor, I have neither given nor received help on this assignment. Brian Team

Problem Description:

The purpose of this homework assignment was to familiarize students with the creation and management of separate processes. By forking a separate process for each token group command's execution, data and variables needed to be managed in between parent and children processes. I ended up being successful in implementing the entire program, and all features appear to be working for the test cases used.

Approach:

This assignment was written entirely within a single header and source pair, msh.h and msh.c respectively. The reason I decided not to split it up more was because the overall functionality of this assignment was fairly straightforward. There are two functions written other than the main function: parse_input and test_args. The former takes the full input string entered by the user and breaks it up into an array of individual tokens and operators, along with counting and storing other variables like the number of pipes. The latter is used to test each token group before it is executed to make sure it is syntactically correct in accordance with the assignment guidelines.

The first step was to get the string input and parsing correct. A simple call to 'fgets' was used for input. This return value was checked for every call to ensure it had not returned NULL, which signified the end of file. If it did return NULL, the overall while-loop was broken out of and the program finished executing. Within the parse_input function, the entire string is iterated over twice. The first iteration is to count the length for a later 'malloc' call for the array to hold each token. It also counts the number of pipes and checks for any inconsistencies in the ordering of operators versus tokens. If two operators are found next to each other without a valid command or args in between, then the invalid_flag variable is set to 1. This variable is checked after the completion of the function to print the 'invalid input' response if there are errors with the input string.

Once each token and operator had been counted, a call to 'malloc' used the number of values multiplied by the size of a char pointer to initialize the first dimension of the array. This would hold the pointers to each string. For the second iteration, the commands and args were built up char-by-char each iteration until a file redirection or a pipe was found. Once this occurred, the text was copied into the corresponding array spot with the operator symbol in the index following it. The end of the function simply returned this array of char pointers.

Once back in the main function, a quick check was performed to make sure that the special argument 'exit' had not been entered. If it had, the main function called the exit function with a return value of zero, and the shell program halted.

The last step before beginning to fork processes off for each function group was to initialize all of the pipes necessary. This was done through the use of the pipe_count variable, which was passed by reference into the parse_input function and set to equal the number of pipes in the input string. The pipe_array variable was an array of int pointers, each pointing to an array

of 2 values. The for-loop iterated through and allocated and assigned each of these pipes through a call to the 'pipe' function, which returned the file descriptors for each end of the pipe.

The while-loop designed to iterate through every token group within a single line of text worked by checking the value of token_groups against the number zero. Every iteration of the loop decremented the variable so it would eventually kick out and wait on all of the forked processes to finish. After the call to 'fork', the PID integer returned was checked for its value. If it equaled 0, then the current frame executing was the newly spawned process. Otherwise, it held the PID for the recently created process.

If the PID equaled zero, then the first step of the code was to check if there needed to be any pipe redirection. Through a series of if/else checks, the program determined if its STDIN, STDOUT, or both needed to be redirected into a pipe. After, each end of every pipe created was closed. This was necessary for each process to receive an EOF notification to let it know it could print or receive text. Without closing the pipes, the processes would hang and become zombies.

A large switch-statement encompassed in a while loop checking for a null-terminator formed the bulk of the processing within the child process. It checked the split_tokens array at the current index and performed different operations based upon its value. If it was one of the file redirection operators, a new file was opened or created and then dup2 was used to redirect it to its respective input or output. If the current token was anything other than an operator, an args array was built up continuously. This would hold ach command and the arguments used in conjunction with them. If the switch-statement encountered a pipe, it broke out of the while-loop and called the test_args function. This checked the ASCII values of every character in the line to ensure that none of the characters that were input were outside of the range described by the assignment. If this test passed, then the next thing called was 'execv', passing in both args[0] and the remainder of the args array as the parameters. Here is where the child process code completed.

If it were the parent process still executing, then it would have skipped all of the processing and executing code above and performed the 'else' clause of the if-statement. This section simply handled the incrementing of pipe and PID variables while also storing each process's PID in the pids array.

After every process had been started for each line in the input file, the parent once more closed all of the pipes created, since it had no need of them. The very last step of the program was to cycle through a for-loop for every stored PID and call 'waitpid' on them. This ensured that the overall parent program would wait for every forked process to finish executing before either finishing or moving onto the next line of the input file. Each process's exit code was also printed to STDERR, as per the assignment instructions. Finally, the earlier memory malloc'ed for arrays was freed, and the program terminated with a return value of zero.

Problems Encountered / Analysis:

There were many hiccups along the way with this assignment, but only a few caused intensive debugging issues.

Initially, the file open function was not working as performed. The Linux man pages state that if you pass the flag O_CREAT, then the file should be created if it does not exist; this behavior was not happening. To counter this, I implemented an error checking statement, and if the 'open' call failed, then I used the 'creat' function. This too, however, caused issues, as the file created would be un-openable due to permissions. This issue was finally resolved through

the use of 'creat's flags parameter, by passing a value of 0777 to give every user full read/write/exec permissions.

By far the biggest headache within this lab was dealing with creating, opening, and closing pipes correctly. My initial implementation only hard-coded a single pipe and only used test cases with one pipe. This continued to error no matter how much I inspected individual values within the debugger. The issue finally became clear when I realized that the return value from calling 'dup2' on the pipes was a negative value, implying an error redirecting STDIN/STDOUT. Using this, I traced the problem back to my implementation of the pipes themselves. I had attempted to make a Pipe struct which held a single, two-element int-array. The for-loop to create the pipes and allocate the memory for each of them seemed to be working when examined within GDB, but something was obviously weird about it. When I finally implemented my pipe_array variable simply as an array of 2-element int-arrays, the pipes were correctly initialized and the logic for closing and opening pipes began to work properly.

Results

After many hours of troubleshooting, the shell program was found to be working correctly for all applied test cases. Each of the cases is listed below:

- Single command
- Single command with args
- Single command with input file
- Single command output to a file
- Single command with input file that also output to a file
- Single command with args with input file that also output to a file
- Single command with args piped into second command with args
- Three commands with args all piped into one another
- Single operand with expected failure
- Two operands without a valid command in between them with expected failure
- A command line ending with an operand with expected failure
- A command line with invalid ASCII characters in it with expected failure
- An input file with multiple lines of commands on it
- Multiple commands with their stderror redirected

While it is recognized that this list is not 100% exhaustive of every single invalid operand combination, all of the major variations of code input were tested and deemed to be correctly handled.

It is also acknowledged that some of this code could be optimized for much better efficiency. The input string is iterated through twice within the parse_input function, and there are perhaps better ways to reduce this down to one. Likewise, multiple other arrays are used throughout the function to hold different variations of the input string, whether it's split up by single tokens, spaces, and operators, or by full tokens groups. This isn't very memory efficient, and there is most likely a more consolidated way to plan out and cycle through the data.

Conclusion

While this project took much longer than expected (even though the professor warned that it would) I believe that it is not truly indicative of the difficulty. Most of the implementation

was finished before the due date, it was just the immense amount of time that it took to find the issues with the piping that caused me to fall behind schedule. This assignment was also a great way to jump back in and re-familiarize oneself with the small details and nuances of C programming, and will be extremely useful throughout the course. Although it took a while, a fully working shell was finally completed in accordance with the guidelines set within the assignment instructions, and a much further understanding of processes and how their data is managed when forking children was obtained.

Source and header files are included below.

makefile

```
CC=gcc
CFLAGS=
DEPS=msh.h

%.o:%.c  $(DEPS)
	$(CC) -c -o $@ $<

msh:  msh.o
	$(CC) -o msh  msh.o
```

msh.h

```
#include  <string.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <stdio.h>
#include  <fcntl.h>
#include  <errno.h>

#define  DEBUG  0

#ifndef  MSH_H
#define  MSH_H

char** parse_input(char  *input,  int *pipe_count,  int *invalid_flag,  int *dup_stderr);
int test_args(char  **args);

#endif
```

msh.c

```
#include "msh.h"

char** parse_input(char *input, int *pipe_count, int *invalid_flag, int *dup_stderr) {
	int i = 0;
	char *head, *tail;	//pointers within array to keep track of command lengths
	int input_length = strlen(input);		//input string length for iteration
	int length = 0;		//holds temp length when parsing args in second loop
	int num_tokens = 0;		//tracking total number of tokens
	int came_from_op = 0;		//boolean for error checking within code
	int came_from_space = 0;//boolean for error checking within code
	int recieved_cmd = 0; //boolean for error checking within code
	int no_op_at_term = 0;		//boolean for error checking at end of loop to ensure we
didnt end with an op
	char **tokens; //array to hold each token string
	char op;		//holds current operand temporarily when saving previous
command
	int num_ptrs = 0;	//holding nuber of pointers (tokens or ops) so we know how many
pointer indices to malloc for array of tokens/ops

	*pipe_count = 0;	//initializing var passed by reference to zero

	*invalid_flag = 0;
	//counting number of pointer spots needed to malloc array
	for (i = 0; i < input_length; i++) {
		//reached an operator
		if (input[i] == ' ' || input[i] == '|' || input[i] == '<' || input[i] == '>' || (input[i] == '2'
&& input[i + 1] == '>')) {
			//need to pipe stderr in the parent function
			if (input[i] == '2' && input[i + 1] == '>') {
				*dup_stderr = 1;
			}

			if (input[i] == '|')
				(*pipe_count)++;

			if (came_from_op && input[i] != ' ' && input[i - 1] != '2') {
				//error, invalid input
				*invalid_flag = 1;
			}
			if (came_from_space || came_from_op) {
				num_ptrs++;
			}
			else {
```

```
                        //needs a spot malloced for previous cmd/arg and also delimiter it
just hit
                        num_ptrs += 2;
                }

                //checking to see if we've gotten a command or if op is very first non-
whitespace in string
                if (recieved_cmd == 0) {
                        *invalid_flag = 1;
                }

                //used for checking next time around validity of hav
                if (input[i] != ' ') {
                        came_from_op = 1;
                        came_from_space = 0;
                } else {
                        came_from_space = 1;
                }

                no_op_at_term = 0;
        } else if (input[i] == '\n') {
                //end of input line, add last ptr space for command and null term
                num_ptrs += 2;
                came_from_space = 0;
                //theoretically break is unneccessary b/c only place we should get this is
very end of string but why not
                if (no_op_at_term == 0) {
                        //just ended with an op, failure
                        *invalid_flag = 1;
                }
                break;
        } else {
                if (input[i] == '2' && input[i + 1] == '>') {
                        *dup_stderr = 1;
                }

                came_from_op = 0;
                came_from_space = 0;
                //verifies check that we didnt get op as very first char
                recieved_cmd = 1;
                //makes sure that an op isnt the last thing before end of input
                no_op_at_term = 1;
        }
}
```

```
	//this handles case if command did not have newline from being input from a file and
simply reached end of loop
	if (i == input_length) {
		num_ptrs++;
	}
	//mallocing enough room for an array of pointers for the pointers to each token or op
	tokens = malloc((num_ptrs + 1) * sizeof(char*));

	//h & t keep track of building command to substring when we hit an op
	head = input;
	tail = input;
	num_tokens = 0;
	for (i = 0; i < input_length; i++) {
		//reached an operator, need new
		if (input[i] == ' ' || input[i] == '|' || input[i] == '<' || input[i] == '>') {
			//grabbing current op delim and storing in op for adding to array after
command
			op = input[i];
			//length now holds the length of the command we just iterated over to use
for malloc
			length = tail - head;
			// length will == 0 if we have spaces and ops right next to eacher other.
dont need to malloc/store a non-existent cmd
			if (length != 0) {
				//mallocing space for cmd string
				tokens[num_tokens] = malloc((length + 1) * sizeof(char));
				//copying actual command/arg into mom that was just malloced
then incrementing to store op
				strncpy(tokens[num_tokens++], head, length);
			}
			//mallocing space for op (always only 1 char)
			tokens[num_tokens] = malloc(sizeof(char));
			//copying actual value into mem that was just allocated and then
incrementing num_tokens for next token
			strncpy(tokens[num_tokens++], &op, sizeof(char));
			//resetting substring pointers. tail skips over delim & first char, head points
to first char after delim
			tail++;
			head = tail;

		} else if (input[i] == '\n' || (i == (input_length - 1))) {
			//if this is the last iteration but without a newline, need to add one more to
grab last char of final cmd/arg
			if ((i == (input_length - 1)) && (input[i] != '\n')) {
				tail++;
			}
```

```
                        //end of input line, add last cmd to array
                        length = tail - head;
                        //mallocing space for cmd string
                        tokens[num_tokens] = malloc((length + 1) * sizeof(char));
                        //copying actual command/arg into mom that was just malloced then
incrementing to store op
                        strncpy(tokens[num_tokens++], head, length);
                        //adding null terminator for later iterating
                        tokens[num_tokens] = malloc(sizeof(char));
                        tokens[num_tokens++] = NULL;
                        break;
                } else {
                        //continue iterating chars, must be building up cmd or arg
                        tail++;
                }
        }

#if DEBUG
        printf("parse_input: printing array of split up tokens\n");
        for (i = 0; i < num_tokens; i++) {
                printf("%d: %s\n", i, tokens[i]);
        }
#endif

        return tokens;
}

int test_args(char **args) {
        int i, j;
        char *test_arg, *iter;

        for(i = 0; args[i] != '\0'; i++) {
                test_arg = args[i];
                j = 0;
                while (test_arg[j] != '\0') {
                        char c = test_arg[j];
                        if ( (c >= 44 && c <= 57) || (c >= 65 && c <= 90) || (c >= 97 && c <=
122) ) {
                                //valid char, continue
                                j++;
                                continue;
                        } else {
                                //invalid
                                return 1;
                        }
```

```
		}
	}

	return 0;
}

int main(int argc, char *argv[]) {

	while(1) {
	//allocating string to hold entire input string
		char input_text[128], *cwd;
		char **split_tokens, **args, **env;
		int i = 0;                                          // iterator
		int j = 0;                                          //iterator
		int arg_idx = 0;                                    //holds index for current arg
being replaced in switch statement
		int new_file = 0;                                   //holds file descriptors for
created files
		int stderr_file = 0;                                //holds file descriptor for stderr file if
needed
		int pipe_count = 0;                                 //holds number of pipes in
line returned from parse_input
		int pipe_idx = 0;                                   //keeps track of current pipe
index for opening and closing
		int token_groups = 0;                               //number of token groups in current
line
		int pid_status = 0;                                 //saves return value from
waitpid
		int invalid_flag = 0;                               //flag that is set for invalid input
within parse_input
		int dup_stderr = 0;                                 //boolean used to check if we
need to dup stderr or not
		int **pipe_array;                                   //array of 2 index int arrays of pipes
		pid_t pid, pid_idx;                                 //hold pid info for forked processes
		pid_t pids[20] = { 0 };                             //statically allocated array for pid values to
wait on them at end

		//rest input buffer in case this command is shorter than previous line of file
		memset(input_text, 0, sizeof(input_text));
		//pulling in input from user into input_text
		if ( (fgets(input_text, sizeof(input_text), stdin) == NULL) )
			break;

		if (strlen(input_text) > 100) {
			printf("invalid input.\n");
		}
```

```
        //splits single string into single token/op array and also counts the pipes, saved in the pipe_count address
                dup_stderr = 0;
                split_tokens = parse_input(input_text, &pipe_count, &invalid_flag, &dup_stderr);
                if (invalid_flag) {
                //invalid op/token order in string, error
                        printf("invalid input.\n");
                        continue;
                }

                //handles exiting overall shell for special input command
                if (strcmp(split_tokens[0], "exit") == 0)
                        exit(0);

                //mallocing array of pointers to each Pipe object (two-element array of ints)
                pipe_array = malloc(sizeof(int*) * pipe_count);
                for (pipe_idx = 0; pipe_idx < pipe_count; pipe_idx++) {
                        pipe_array[pipe_idx] = malloc(sizeof(int[2]));
                        if (pipe(pipe_array[pipe_idx]) != 0)
                                perror("pipe()");
                }

                //resetting pipe_idx to start from begiining
                pipe_idx = 0;
                pid_idx = 0;
                token_groups = pipe_count;
                //one more token group than number of pipes
                token_groups++;
                i = 0;

                while (token_groups > 0) {
                        //reduce pipe count by one regardless, since overall loop checks for negative number (consider case of single command no pipe)_
                        token_groups--;

                        //create new process
                        pid = fork();

                        if (pid == 0) {

                                if (pipe_count > 0) {
                                //if there was an output_pipe from the parent we just came from, that means we need to tie our STDIN to the read side of that pipe
                                        if (token_groups > 0) {
                                                if (token_groups == pipe_count) {
```

```
                                        //very first process. open write of first pipe
and close read\
                                        //pipe_idx should equal zero everytime were
here
                                        close(pipe_array[pipe_idx][0]);
//close(pipe_array[pipe_idx].rdwr[0]);
                                        //dup2 STDOUT into the write next pipe end
                                        dup2(pipe_array[pipe_idx][1],
STDOUT_FILENO);
                                } else {
                                        //there is definitely one following us, but we
are not the very first in line
                                        //close the read pipe end for next pipe
                                        close(pipe_array[pipe_idx][0]);
                                        //dup2 STDOUT into the write next pipe end
                                        dup2(pipe_array[pipe_idx][1],
STDOUT_FILENO);

                                        //close the write of current preceding pipe
                                        close(pipe_array[pipe_idx - 1][1]);
                                        //dup2 STDOUT into the read current
preceding pipe end
                                        dup2(pipe_array[pipe_idx - 1][0],
STDIN_FILENO);
                                }
                        } else {
                                //we are last token group in line of one or more
pipes
                                //close write end of preceding pipe
                                close(pipe_array[pipe_idx - 1][1]);
                                //dup2 STDIN to the read end of preceding pipe
                                dup2(pipe_array[pipe_idx - 1][0],
STDIN_FILENO);
                        }
                                //close all remaining pipes (closing ones weve
already dup2d simply lowers their filecount from 2 to 1)
                        for (j = 0; j < pipe_count; j++) {
                                close(pipe_array[j][0]);
                                close(pipe_array[j][1]);
                        }
                }

                if (dup_stderr) {
                        dup2(stderr_file, STDERR_FILENO);
                }
```

```
                        //allocing array to hold single command + args for each token
group.
                        //assuming no command will have more than 20 args per token
group
                        args = malloc(sizeof(char*) * 20);

                        arg_idx = 0;
                        while (split_tokens[i] != '\0') {
                                //normally this would have gone in the while loop above,
but the 2d [i][0] access segfaults for the null term
                                if (split_tokens[i][0] == '|') {
                                        //advance past the pipe so we come in next time
(child process) with the new command
                                        i++;
                                        //break out of while loop to exec this token group
                                        break;
                                }

                                switch (split_tokens[i][0]) {
                                        case ' ':
                                                //do nothing, we're about to hit an op,
command, or arg
                                        break;
                                        case '<':
                                        //iters over whitespace until filename is found
                                        i++;
                                        while (split_tokens[i][0] == ' ') i++;

                                        new_file = open(split_tokens[i], O_RDONLY);
                                        if (new_file < 0) {
                                                perror("Error opening file to read");
                                        }
                                        //dup2 command output into file input
                                        dup2(new_file, STDIN_FILENO);
                                        break;
                                        case '>':
                                        //moving to next split token. either space or
filename
                                        i++;
                                        //clearing any more whitespace until we get to
filename
                                        while (split_tokens[i][0] == ' ') i++;
                                        //create/open file following this. should be able to
handle relative or absolute
                                        new_file = open(split_tokens[i], O_RDWR ||
O_CREAT, S_IRWXU || S_IRWXG || S_IRWXO);
```

```
                                        if (new_file < 0) {
                                        //printf("File does not exist, creating it.\n");
                                                new_file = creat(split_tokens[i], 0777);
                                                if (new_file < 0) {
                                                        perror("creat()");
                                                }
                                        }
                                        //dup2 command output into file input
                                        dup2(new_file, STDOUT_FILENO);
                                        break;
                                        default:
                                        if (split_tokens[i][0] == '2') {
                                                if (split_tokens[i + 1][0] == '>') {
                                                //need to redirect stderr
                                                //move past one space to > then one more to the space
                                                        i += 2;
                                                //clearing any more whitespace until we get to filename
                                                        while (split_tokens[i][0] == ' ') i++;
                                                //create/open file following this. should be able to handle relative or absolute
                                                        stderr_file = open(split_tokens[i], O_RDWR || O_CREAT, S_IRWXU || S_IRWXG || S_IRWXO);
                                                        if (stderr_file < 0) {
                                                        //printf("File does not exist, creating it.\n");
                                                                stderr_file = creat(split_tokens[i], 0777);
                                                                if (stderr_file < 0) {
                                                                        perror("creat()");
                                                                }
                                                        }
                                                        dup2(stderr_file, STDERR_FILENO);
                                                }
                                                break;
                                        }
                                        //hits this when it's a command or arg for preceding command
                                        //build args array by strncopying command or arg currently in split_tokens into args at the current arg_idx
                                        args[arg_idx] = malloc(strlen(split_tokens[i]) * sizeof(char));
                                        strncpy(args[arg_idx++], split_tokens[i], strlen(split_tokens[i]));
```

```
                                        break;
                                }       //switch

                                i++;

                        }       //while != '|' or '\0'


                        //checks token group to make sure no illegal combos of ops and/or
cmds/args are used
                        if (test_args(args)) {
                                //error with invalid char
                                printf("invalid input.\n");
                        }
#if DEBUG
                        else {
                                printf("Valid characters!!\n");
                        }
#endif

                        execv(args[0], args);

                } else {
                        //we're in the parent process still

                        //increment pipe_idx for next token group to use if necessary
                        pipe_idx++;

                        //store PID
                        pids[pid_idx] = pid;
                        pid_idx++;

                        //moves shell parent i value (pointer to tokens/args/ops in
split_tokens) up until pipe and then one more for next fork to be at correct spot in array
                        while (split_tokens[i] != '\0') {
                                if (split_tokens[i][0] == '|') {
                                        i++;
                                        break;
                                } else {
                                        i++;
                                }
                        }
                }               //end of if (pid == 0) else
        }               //end of while(token_groups > 0)

        //close all pipes
```

```
                for (j = 0; j < pipe_count; j++) {
                        close(pipe_array[j][0]);
                        close(pipe_array[j][1]);
                }

                //cycling throug and waiting on all processes. checks for <= instead of < bc there
are pipe_count + 1 token groups executed
                if (dup_stderr) {
                        dup2(stderr_file, STDERR_FILENO);
                }

                for (pid_idx = 0; pid_idx <= pipe_count; pid_idx++) {
                        //resetting to zero for next process
                        pid_status = 0;
                        //blocks until current process is finished, then returns PID of finished
process. we continue onto checking next stored pid
                        waitpid(pids[pid_idx], &pid_status, 0);
                        fprintf(stderr, "%d\n", pid_status);
                }

                //release malloced mem
                free(pipe_array);
                free(split_tokens);
        }

        return 0;

}
```