CS 4414 Operating Systems – Fall 2017

Homework 1: Writing a Simple Shell

Jonathan Colen

Problem Description:

The goal of this assignment was to implement a simple Unix shell that could run programs, handle file redirection, and manage pipes. This assignment was completed and the program runs correctly.

Approach:

The program operates by running the **shell** method, which contains an endless while loop. At each iteration, a line is taken from standard input, parsed into a sequence of token groups, and executed. The input is handled using the **fgets** command. The user can enter as many commands as they desire and can end the program at any time using the "exit" command. If a file has been redirected to the standard input for the shell program, then the program will continue reading lines until the end-of-file marker has been reached.

Each line is parsed and stored a structure called **token_group**, which will be referred to as **Group**. Each Group contains the command to be executed and its arguments, file redirections, and pipe file descriptors. Token groups that are connected via a pipe will have pointers to each other stored in their Group structure. In addition, each Group has a place to store the process id and exit status of the resulting process for management purposes.

Input lines are parsed using the method **parse_line**. This method separates the input string into a sequence of tokens separated by spaces using **strtok**. Next, it creates a new Group and sets the command of that Group as the first token in that line. All subsequent tokens before the next occurrence of the pipe symbol ("|") are stored in a list string for later parsing. When the pipe symbol is reached, the list of associated tokens is processed and a new token group is created and connected via a pointer to the previous token. This is repeated until there are no more tokens in the line.

The associated tokens for each command are processed using the **finish_group** method. The tokens are checked for the file redirection operators ">" and "<". If they exist, the subsequent strings to each operator are stored in the Group structure as the input or output filenames. In addition, their compatibility with the existing pipes for the token group are checked. If the token group already has an associated input pipe, then any input redirections will be disallowed. Similarly, output redirections will be disallowed if an output pipe exists. All other associated tokens are stored in the string list storing the arguments of the Group.

After the line has been parsed, the **execute_line** method is called with a pointer to the first token group of the line set as an argument. This method runs through a while loop where at each iteration, the current Group is executed and the pointer is incremented to the Group associated with the current command's output pipe. The loop iterates until there are no more commands to be executed.

Execution of a Group is done in several steps. First, if the Group has another process that it should be piped *to*, a new pipe is created and it is set as the output pipe for the current Group and the input pipe for the current Group's successor. This means that no Group ever has to create its own input pipe. This method of creating pipes was chosen because the first Group in a line could not have an input pipe due to the constraints of the language. Thus, the Groups were made to simultaneously manage their own output pipes and their successor's input pipes in order to allow each Group to be handled in the exact same way.

Next, a child process is forked. The child process either closes the write end of its input pipe and sets its standard input as the read end of that pipe, or it opens a file with the name stored during the line parsing and sets standard input as the resulting file descriptor. If there is no input pipe and no redirected input file, then nothing is done and standard in is left unchanged. Similarly, the read end of the output pipe can be closed and the write end can replace standard out, or a file can be opened and set as standard out if the Group structure contains an output file redirection. Finally, the command is executed using **execve**. If the program fails to execute, an error message is written to the console.

Meanwhile, the parent process simply stores the process id returned by **fork** into the Group structure for later access. Next, the Group structures are iterated through again and the parent calls **waitpid** on the process ids stored for each Group. The returned status is stored in the Group and the write ends of the output pipes and read ends of the input pipes are closed. It was found that neglecting to close these pipe ends would result in the overall shell failing to terminate correctly due to processes continuing to run. After all processes have finished executing, their exit codes are printed out by once more iterating through the Group structures. Finally, the Groups are deallocated and another input line is accepted.

Results:

The program executes as expected and all specified functionalities have been implemented. If an input line is invalid based on the language specification supplied, an error message will be output with the form 'Command "*command_here*" is invalid'. The program also handles incompatible redirections and pipes to nowhere. Relative file paths are supported if the filename does not begin with the forward slash character. If a process fails to execute, a message is printed to the standard error console, and otherwise, the exit code is printed to standard output.

All forked processes terminate as expected. This was verified by running the **ps** Unix command in a terminal and checking the output while the shell program operated.

Analysis:

Parsing input strings was the most difficult and involved portion of this assignment. The initial attempt involved tokenizing the string and then assigning each token a numerical flag to designate it as a "command", "argument", "operator", or "file descriptor". The idea behind this was that execution of a line would require using the flags to link arguments and redirections to their associated commands. This approach became very difficult to manage and was abandoned quickly.

The second approach, which is the one described above, is the result of planning the string parsing with actual execution in mind. I realized that the goal of the program was to execute commands, and therefore the goal of the parsing should be to create a structure for each line that allowed for easy execution of the commands stored in that line. The parsing subroutine then shifted from one that attempted to flag and link tokens to one that sought to fill up a structure. This method made many later problems much simpler to solve. Checking compatibility of pipes and file redirections now only requires checking whether two pointers in a Group structure are NULL. The filenames for input and output redirection can be accessed instantly for a given command. The Group structure also allows for easier process cleanup, as the process ids and statuses are stored in each Group as well.

It should be noted that each Group structure stored its own input and output pipe information. Because one Group's output pipe is another Group's input pipe, this means that there is some data redundancy. This was a conscious decision intended to allow each token group to be independent. For cleanup purposes, it was simpler to associate each command with its "own" pipes. This could also have been accomplished by using a global pipe array that could be filled and accessed by all processes. It was ultimately a stylistic decision to put the pipes in the Group structures, as it kept all necessary information for each process contained within a single structure.

Utilization of data structures simplified both the parsing and execution stages of the program and saved hours of work. They provided significant organizational advantages and made many of the process management procedures very straightforward.

Conclusions:

The shell program operates by accepting input lines, parsing those lines to fill token group structures, and executing the commands associated with each token group. Using data structures to represent commands and token groups simplified the parsing and process management. The program properly utilizes the **fork(), exec(), open(), dup2(),** and **pipe()** system calls. All facets of the specified language are supported and the program executes commands as expected.

Code:

The project is stored in jc8kf.tar
The code is contained in **shell.c**
The code can be compiled by running **make**
The program can be run by executing **./msh**

Pledge:

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

```
/*
*       Name:          Jonathan Colen
*       Email:         jc8kf@virginia.edu
*       Class:         CS 4414
*       Professor:     Andrew Grimshaw
*       Assignment:    Homework 1
*
*       The purpose of this program is to simulate a shell. Commands can be input via stdin
*       and they will be executed. This shell supports file redirection and piping. All
*       commands specified will be assumed to be executable files in the local directory,
*       otherwise, the full path will be provided. Output via stderr will be shown in the
*       event the command does not execute.
*
*       This program can be compiled via
*               gcc -o msh shell.c
*       And can be run via
*               ./msh
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>

/*
*       Structure for a doubly linked list of strings
*       This structure was used to simplify string parsing
*/
typedef struct string_node    {
        char * str;
        struct string_node * next;
        struct string_node * prev;
} Snode;

/*
*       This structure represents a token group input into the program.
*       It stores the command and a list of arguments, as well as necessary information for redirection.
*       In addition, each token group contains pointers to the next and previous token groups. This
allows
*       an entire line to be executed by calling execute_line with the first token group in the line, as
*       that method will execute until the pointer to the next group is null.
*/
typedef struct token_group    {
        char * cmd;                                 //Stores the command of this token group
        char ** args;                       //Stores the arguments
        int num_args;                       //Stores number of arguments
        int pid;                                    //Stores pid of command process
        int status;                                 //Stores exit status code of process
        char * redir_out;                   //Stores name of file for output redirection
        char * redir_in;                    //Stores name of file for input redirection
        struct token_group * pipe_to; //Stores the function to be piped to, or NULL if there is none
        struct token_group * pipe_from;     //Stores the function that is piped from, or NULL if there
is none
        int pipe_in[2], pipe_out[2];  //Stores the pipe file descriptors generated for the input and
output pipes
} Group;

/*
*       Create a new group with all fields initialized to NULL. Status and pid are set to -1 to start.
*/
Group * create_empty_group()  {
        Group * ret = (Group*)malloc(sizeof(Group));
        ret->cmd = NULL;
        ret->args = NULL;
        ret->num_args = 0;
        ret->pid = -1;
        ret->status = -1;
        ret->redir_out = NULL;
        ret->redir_in = NULL;
        ret->pipe_to = NULL;
```

```
	ret->pipe_from = NULL;
	return ret;
}

/*
*	Create an empty string list node
*/
Snode * create_empty_snode()  {
	Snode * ret = (Snode*)malloc(sizeof(Snode));
	ret->str = NULL;
	ret->next = NULL;
	ret->prev = NULL;
	return ret;
}

/*
*	Assigns a status value to the given token based on what kind of token it is. Status value are 
summarized below
*	Word	-> 0
*	<		-> 1
*	>		-> 2
*	|		-> 3
*	Invalid -> -1
*/
int characterize_token(char * token) {
	//It is a valid word if it matches the pattern A-Z a-z 0-9 - . / _
	//According to an ascii table, the word pattern [A-Za-z0-9-./_] is 45-57, 65-90, 95, and 97-122
	//It is a valid operator if it is one character long and contains <>|
	int ind = 0;
	int status = -1;

	if ((token[ind] >= 45 && token[ind] <= 57) ||
		(token[ind] >= 65 && token[ind] <= 90) ||
		(token[ind] >= 97 && token[ind] <= 122) || token[ind] == 95)	{
		status = 0;
	} else if (token[ind] == '<')	{
		status = 1;
	} else if (token[ind] == '>') {
		status = 2;
	} else if (token[ind] == '|') {
		status = 3;
	} else  {
		return -1;
	}

	ind ++;
	while (token[ind] != '\0')	{
		//Only allowable continuation past one character is a word
		if (token[ind] < 45 || (token[ind] > 57 && token[ind] < 65) ||
			(token[ind] > 90 && token[ind] < 97 && token[ind] != 95)
				|| token[ind] > 122 || status != 0)  {
			return -1;
		}
		ind ++;
	}

	return status;
}

/*
*	Clear all memory assoociated with a string linked list
*/
void clear_all_snodes(Snode * curr)  {
	Snode * tmp = curr;
	while(curr)	{
		tmp = curr -> next;
		free(curr->str);
		free(curr);
		curr = tmp;
	}
}

/*
*	Clear all memory associated with a series of linked token groups
```

```
*/
void clear_all_groups(Group * curr)  {
	int i;
	Group * tmp = curr;
	while(curr)	{
		tmp = curr -> pipe_to;
		free(curr->cmd);
		for (i = 0; i < curr->num_args; i ++)	{
			free((curr->args)[i]);
		}
		free(curr->args);
		free(curr->redir_in);
		free(curr->redir_out);
		free(curr);
		curr = tmp;
	}
}

/*
*	Fill all relevant fields in the token group curr by going throgh the string linked list args
*	@param curr - the group to be populated
*	@param args - a pointer to the head of the arguments linked list
*	@param total_args - the number of arguments in args
*	@param redirects - the number of redirection arguments
*
*	NOTE: If there is some problem with parsing the line (arguments invalid, etc.), then the command
*	will be set to NULL. This will let the shell() wrapper know that the line has a problem and an
*	error message will be output.
*/
void finish_group(Group * curr, Snode * args, int total_args, int redirects)	{
	if (curr -> cmd == NULL)	{
		return;
	}
	int tind = 0, aind = 0;	//tind stores the index in args, and aind stores the index in 
curr->args
	Snode * args_head = args;
	int word_args = total_args - 2 * redirects + 1;	//Number of non-redirect arguments, 1 extra 
for cmd in spot 0
	//Allocate one extra spot since argv is null-terminated in execve
	curr->args = (char**) malloc(sizeof(char*) * (word_args + 1));	//Allocate argument list with 
needed size
	(curr->args)[aind] = (char*) malloc(sizeof(char) * 100);	//Place command name in first 
argument slot
	strcpy((curr->args)[aind], curr->cmd);
	aind ++;
	curr->num_args = word_args;		//Set number of arguments
	while(tind < total_args)	{
		if (strcmp(args->str, ">") == 0 || strcmp(args->str, "<") == 0)	{
			int type = -1; //0 for input, 1 for output
			if (strcmp(args->str, ">") == 0)	{
				if(curr->redir_out)	{		//Only one of each redirect allowed
					clear_all_snodes(args);
					curr->cmd = NULL;
					return;
				}
				type = 1;
				curr->redir_out = (char*) malloc(sizeof(char) * 100);
			} else {
				if (curr->redir_in)	{
					clear_all_snodes(args);
					curr->cmd = NULL;
					return;
				}
				//Make sure output has not been set since order matters according to Prof 
Grimshaw's email
				if (curr->redir_out)	{
					clear_all_snodes(args);
					curr->cmd = NULL;
					return;
				}
				type = 0;
				curr->redir_in = (char*) malloc(sizeof(char) * 100);
			}
```

```
                        //Check next string to get filespec, if possible
                        args = args->next;

                        if (args->str == NULL || strcmp(args->str, ">") == 0 || strcmp(args->str, "<") ==
0)      {
                                //Invalid filespec so we should return. Make sure to clear memory
                                clear_all_snodes(args);
                                curr->cmd = NULL;
                                return;
                        }

                        if (type)      {       //Check to place filespec in the proper slot
                                strcpy(curr->redir_out, args->str);
                        } else  {
                                strcpy(curr->redir_in, args->str);
                        }
                        args = args->next;

                        tind ++;       //Increment tind extra since we're going ahead to get filespec
                } else {       //If it's a normal word then add it to the normal arguments list
                        (curr->args)[aind] = (char*)malloc(sizeof(char) * 100);
                        strcpy((curr->args)[aind], args->str);
                        aind ++;
                        args = args->next;
                }
                tind ++;
        }

        (curr->args)[word_args] = NULL;      //Null terminate the args list
        clear_all_snodes(args_head); //Free the linked list
}

/*
*       Check to see that the redirections contained in g are compatible with the pipes
*/
int invalid_redirs(Group * g) {
        return (g->redir_out && g->pipe_to) || (g->redir_in && g->pipe_from);
}

/*
*       Parse a line into a sequence of connected token groups
*       @param input - the line to be parsed
*       @param cwd - the path of the current working directory, to be appended to any commands that are
*                               not absolute paths
*       NOTE: If there is some problem with parsing the line, the command of the first token group will
*       be set to NULL. This will tell the shell() method that something is wrong, and an error message
*       will appear.
*/
Group * parse_line(char * input, char * cwd){
        Group * ret = create_empty_group();  //Pointer to the first command in the line
        Group * curr = ret;    //Pointer to the current command being updated
        char seps[] = " ";     //Commands are separated by spaces
        char * token;          //Stores current token
        int status;                    //Status (pipe, file redirect, token, invalid) of current token
        int redirects, total_args;    //Counts number of redirects and total arguments in an arguments
list for later
        Snode * args_head;                     //Head of the arguments linked list being created for a
token group
        Snode * args_curr;                     //Current position of the arguments linked list being
created for a token group

        token = strtok(input, seps);
        while(token != NULL)   {
                status = characterize_token(token);  //Check if it is a word or some kind of operator
                if (status == -1)      {
                        ret -> cmd = NULL;
                        return ret;
                }
                if (curr->cmd) {
                        if (status == 3)       {       //If it is a pipe, flatten and evaluate the args
list and create a new group
                                finish_group(curr, args_head, total_args, redirects);
                                if (curr -> cmd == NULL)      {       //If invalid redirects or similar,
return immediately
```

```
                                        ret -> cmd = NULL;
                                        return ret;
                                }
                                curr->pipe_to = create_empty_group();
                                if (invalid_redirs(curr))     {       //Make sure redirects are compatible
with the pipes
                                        ret->cmd = NULL;
                                        return ret;
                                }
                                curr->pipe_to->pipe_from = curr;      //Set up pipe pointer
                                curr = curr->pipe_to;                         //Move to next token group
                        } else {
                                if (status == 1 || status == 2)       {       //file redirect operators
                                        redirects ++;
                                }
                                args_curr->str = (char*)malloc(sizeof(char) * 100); //Allocate space in
the word list
                                strcpy(args_curr->str, token);
                                args_curr->next = create_empty_snode();
        //Create next node in word list
                                args_curr->next->prev = args_curr;                                //Set
up pointers
                                args_curr = args_curr->next;  //Go to next node
                                total_args ++;                                //Increment total words in
word list
                        }
                } else {
                        if (status)    {       //If there is no command and the first token is an
operator, it is invalid
                                ret -> cmd = NULL;
                                return ret;
                        }
                        //Otherwise, fill in the command in the struct and initialize the arguments
linked list
                        curr->cmd = (char*)malloc(sizeof(char) * 100);
                        if (token[0] == '/')  {
                                strcpy(curr->cmd, token);     //First character of '/' means it is a
direct path
                        } else {       //If first character is not /, then it is a relative pathname
                                strcpy(curr->cmd, cwd);
                                strcat(curr->cmd, "/");               //getcwd() doesn't include a
terminating slash
                                strcat(curr->cmd, token);
                        }
                        args_head = create_empty_snode();     //Initialize the argument word linked list
                        args_curr = args_head;
                        redirects = 0; //Store the number of redirects in the post command word list
                        total_args = 0;        //Store the total number of post command words/operators
before next pipe
                }
                token = strtok(NULL, seps);
        }

        finish_group(curr, args_head, total_args, redirects);
        if (curr -> cmd == NULL || invalid_redirs(curr))     {       //Make sure pointers etc for final
group are valid
                ret -> cmd = NULL;
                return ret;
        }

        return ret;
}

/*
*       Executes all commands in the sequence of connected token groups stored in line
*       @param line - the first token group in the line to be executed
*/
int execute_line(Group * line)       {
        int pid, fin, fout, status, exec_err;
        Group * curr = line;
        char * envp[1] = {NULL};
        int pipe_ends[2];
        int pipe_status;
        //Execute commands in the line until there are none left
        while(curr)    {
                //Handle pipes before fork
                if (curr->pipe_to)    {       //0 is pipe read end, 1 is pipe write end
                        pipe_status = pipe(pipe_ends);
                        if (pipe_status == -1){
                                return -1;
                        }
                        //Ensure the next command has the proper pipe set up
                        (curr->pipe_out)[0] = pipe_ends[0];
                        (curr->pipe_out)[1] = pipe_ends[1];
                        (curr->pipe_to->pipe_in)[0] = pipe_ends[0];
                        (curr->pipe_to->pipe_in)[1] = pipe_ends[1];
                }

                pid = fork();

                if(pid == 0)   {       //If child process, execute the command
                        // I/O redirection using file redirects or pipes
                        //According to gnu.org/software/libc/manual/html_node/Descriptors-and-
Streams.html,
                        //stdin has value 0 and stdout has value 1
                        if (curr->pipe_from)  {
                                close((curr->pipe_in)[1]);    //Close write end
                                dup2((curr->pipe_in)[0], 0);
                        } else if (curr->redir_in)    {
                                fin = open(curr->redir_in, O_RDONLY);
                                if (fin == -1) {              //Issue opening file
                                        fprintf(stderr, "Error opening file %s\n", curr->redir_in);
                                        exit(1);
                                }
                                dup2(fout, 0);
                        }
                        if (curr->pipe_to)    {
                                close((curr->pipe_out)[0]);   //Close read end
                                dup2((curr->pipe_out)[1], 1);
                        } else if (curr->redir_out)   {
                                fout = open(curr->redir_out, O_CREAT | O_WRONLY, 0666);    //Allow all r/w
permissions in mode
                                dup2(fout, 1);
                        }

                        exec_err = execve(curr->cmd, curr->args, envp);
                        if (exec_err == -1)    {       //If the command did not exist or was called
incorrectly
                                fprintf(stdout, "Command %s failed to execute\n", curr->cmd);
                                exit(1);       //
                        }
                } else {                       //If parent process, store pid in the token group struct
                        curr->pid = pid;
                }

                if (curr->pipe_to)    {       //Close the pipe or the program will never terminate
                        close((curr->pipe_out)[1]);
                }
                if (curr->pipe_from)  {
                        close((curr->pipe_in)[0]);
                }

                curr = curr -> pipe_to;       //Advance to next token group to be executed
        }
        curr = line;
        while(curr)    {
                //Wait for child process to exit and look at status code
                waitpid(curr->pid, &status, 0);
                curr->status = status;
                curr = curr -> pipe_to;
        }

        //Print out exit codes now that all processes have completed
        curr = line;
        while(curr)    {
                //Exit code is stored in lower 8 bits of status
                fprintf(stdout, "%s exited with exit code %d\n", curr->cmd, curr->status & 0xFF);
```

```
		curr = curr -> pipe_to;
	}

	//When output of this program is redirected to a file, the exit codes do not print until
	//the end. This happens because stdout data is buffered differently in the C program than
	//from the child processes. To avert this problem, flush the line buffers now using
	//fflush as per The C Programming Language 2nd Edition pg 242
	fflush(stdout);

	return 0;
}

/*
*	Handles operations of the entire shell. Accepts input via stdin, parses it, and executes it
*/
int shell()	{
	int input_length = 100;
	char input[input_length];
	char input_cpy[input_length];
	char cwd[100];
	Group * line;
	getcwd(cwd, 100);

	while(1)	{
		printf("> ");		//Collect input
		if(fgets(input, sizeof input, stdin) == NULL)	{
			return 0;
		}
		fflush(stdout);		//Make sure > is flushed in case there is batch input or else
things will print oddly
		//Need to check if final character in the line is \n - if not then the line is too long
		if (strlen(input) == 1 && input[0] == '\n') {	//Ignore if line is empty
			continue;
		}
		//Condition for line being longer than max allowed input length
		if(strlen(input) == input_length - 1 && input[input_length - 1] != '\n')  {
			while(input[strlen(input) - 1] != '\n')	{	//continue to read string
until done
				fgets(input, sizeof input, stdin);
			}
			fprintf(stderr, "Input line longer than limit of %d characters\n", input_length);
			continue;
		}

		if(input[strlen(input)-1] == '\n')	{
			input[strlen(input) - 1] = '\0';	//terminate one character sooner because of
\n
		}
		if(strcmp(input, "exit") == 0)	{	//terminate program if command is exit
			return 0;
		}

		strcpy(input_cpy, input);
		line = parse_line(input, cwd);	//a pointer to the first command in the line
		if (!(line->cmd))	{			//line->cmd = NULL is code for "this line
has an error"
			fprintf(stderr, "Command \"%s\" is invalid\n", input_cpy);
			//Make sure to clear the memory associated with line
			clear_all_groups(line);
			continue;
		}

		execute_line(line);
		clear_all_groups(line);
	}
}

int main()	{
	return shell();
}
```