CS 4414 Operating Systems – Fall 2017

Homework #1

Maurice Wong

Problem Description:

The goal of this assignment was to implement a simple UNIX shell capable of running programs and supporting pipes and file redirection. The assignment was successfully completed.

Approach:

The shell was written as a continuously running while loop that reads in a line of input at a time. The body of the shell was divided into four main phases:

1. Reading in a line into an array of characters.
2. Lexing the lines into tokens (words and operators).
3. Parsing the list of tokens into groups that represent each command and its arguments.
4. Interpreting each token group and executing the command.

Initially, the shell begins with the *readLine()* function that reads in individual characters from stdin into an array of characters until a newline character appears. Error checks for exceeding the maximum number of characters in a line occur here.

For the lexing phase, a *tok_list* struct was created to store an array of strings and the size of the array for ease of iterating through. This struct was used to store all the tokens within a line. In the *tokenizeLine()* function, the stdlib function *strtok()* is utilized to divide the line of characters into tokens separated by spaces. Each token is then validated to ensure it is either an operator or a word and that it does not have invalid characters.

In the parsing phase, tokens are grouped together into structs called *tok_group*. This struct has a number of fields to represent the possible options that a token group has for an instruction:

- *command* – the command string
- *args* – a *tok_list* of arguments for that instruction
- *input_redirect & output_redirect* – boolean fields indicating if the instruction performs a file input or output redirection, respectively
- *input_file & output_file* – string fields with the name of the file used for input/output redirection, respectively

By first iterating through the tokens and counting the number of pipe operators "|", the number of groups can be determined. The *parseTokens* function loops through the tokens and forms these *tok_group* structs, setting the fields of the structs as appropriate based on what operators and words are encountered. Any parsing errors are detected within this function, which are improper orderings of the tokens. The results are then stored in a created struct called *tok_group_list* which contains an array of *tok_groups* and the size of the array for ease of iterating.

Once the line is divided into token groups, the token groups can be iterated over and executed in separate processes. This is done in the *runCommands()* function. Based on the number of token groups *n*, it creates a set of *n-1* pipes to direct input/ouput between the list of token groups so that all the forked processes will have access to the pipes. Each token group is then iterated over and a new child process is forked to execute each instruction. In the child process, the appropriate files are opened for input/output file redirection as well as setting up pipe redirection using *dup2()*. The commands are then run with *execve* and the appropriate

arguments. The parent process stores the child process id's into an array and iterates over them, waiting for them to finish and printing their exit codes to stderr.

The main shell loop concludes with freeing allocated memory used for storing tokens and groups. The while-loop continues to process the next line of input.

Results:

The resulting simple shell successfully works. Commands were tested that involved combinations of input/output file redirection and pipes were all used successfully. Commands that took up the terminal (like *emacs*) were also tested and were successful. Invalid commands that had too many characters per line and improper syntax were all tested and failed correctly. Invalid programs were attempted to be exec'd and invalid files were attempted to be opened for redirection, failing as intended.

Problems Encountered:

The most difficult part of the shell was figuring out how to setup the pipes. Initially, only the ends of the pipes that were not used in each process were closed, but child processes hung when this code was tested. It was later deduced after some experimentation that all the file descriptors in both parent and child processes pointing to the write-end of the pipe needed to be closed. This is because the child process continues to read from the write-end of the pipe until all file descriptors pointing to it are closed.

Some strange behavior was observed when printing to the terminal. When redirecting command lines into the shell executable on a local VM. The result of the commands printed first, and then all the shell prompts ">" that should have printed between the commands were all printed at the end.

In addition, the output from the exec'd processes sometimes came after the printed exit codes, even though the exit codes were explicitly to be printed after the child processes had terminated.

Analysis:

Some experimentation with the code was performed in attempts to debug this strange behavior shell prompt printing behavior for file input redirection of commands. It was found that when printing text with a newline, the behavior went away and shell prompts were printed between output from lines of command. This can be explained by I/O buffering, where the OS saves input to be printed to the terminal in a buffer and prints it when ready. In the case of the shell prompt, using *fflush(stdout)* flushed out the buffer to the screen, fixing the strange order.

Also it was discovered that redirecting the output of *./msh* to another file would correct the order of the child processes' output and the exit codes, which is also likely an issue that can be attributed to I/O buffering.

Conclusion:

This was an informative assignment on understanding how to create and handle multiple processes, as well as pipe communication and file redirection for processes. It was challenging to get pipe communication and other input/output redirections to work, but some experimentation with file descriptors yielded successful results.

On my honor, I have neither given nor received aid on this assignment: Maurice Wong

```c
/**
 * Homework 1: Writing Your Own Shell
 *
 * CS4414 Operating Systems
 * Fall 2017
 *
 * Maurice Wong - mzw7af
 *
 * main.c - shell program
 *
 * COMPILE:     make
 * OBJECTS:     main.o
 * RUN:         ./msh
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wait.h>
#define MAXCHAR 100

typedef struct tok_list {
    char **tokens;
    int size;
} tok_list;

typedef struct tok_group {
    char *command;
    tok_list args;
    int input_redirect; // booleans for if file redirection is required
    int output_redirect;
    char *input_file;
    char *output_file;
} tok_group;

typedef struct tok_group_list {
    tok_group *groups;
    int size;
} tok_group_list;

int readLine(char line[]);
tok_list tokenizeLine(char line[]);
int isValidToken(char token[]);
int isValidWordChar(char c);
tok_group_list parseTokens(tok_list tokens_list);
tok_list readArgs(tok_list token_list, int *tok_iter);
int** createPipes(int num);
void destroyPipes(int** pfds, int num);
int openFiles(tok_group group);
int isOpString(char *str);
void runCommands(tok_group_list groups);
int strEqual(char *str1, char *str2);
void freeMemory(tok_list tokens, tok_group_list groups);
```

```c
int main() {
    char line[MAXCHAR + 1]; // +1 for \0
    while (1) {
        printf(">");
    fflush(stdout);
        // read a line
        int lineStatus = readLine(line);
        if (!lineStatus) {
            printf("Error lexing line. Skipping line\n");
            continue;
        }
        // separate line into tokens
        tok_list token_list = tokenizeLine(line);
        if (token_list.tokens == NULL && token_list.size != 0) {
            printf("Error lexing line. Skipping line\n");
            continue;
        }
        // parse into commands:
        tok_group_list groups = parseTokens(token_list);
        if (groups.groups == NULL) {
            printf("Error parsing line. Skipping line\n");
            continue;
        }
        // run commands
        runCommands(groups);

        freeMemory(token_list, groups);
    }
    return 0;
}

/**
 * Reads in a line and returns status code:
 * 1 if line was read correctly
 * 0 if number of characters was too great
 */
int readLine(char line[]) {
    int c, i;
    // read up until MAXCHAR
    for (i = 0; i < MAXCHAR; i++) {
        c = getchar();
        if (c == EOF) {
            if (i == 0) {
                exit(0);
            } else {
                printf("Error, EOF detected in a line");
                exit(1);
            }
        }
        if (c == '\n') {
            line[i] = '\0';
            return 1;
        }
        line[i] = c;
    }
```

```
        // Max char exceeded, consume all char until end of line:
        while (1) {
            c = getchar();
            if (c == EOF) {
                printf("Error, EOF detected in line");
                exit(1);
            }
            if (c == '\n')
                return 0;
        }
    }

    /**
     * Divide a line of characters into tokens and validate each one.
     * Token list is set as NULL if an error occurs while tokenizing
     */
    tok_list tokenizeLine(char line[]) {
        char **results = malloc(50 * (sizeof(char*)));
        int numTokens = 0;
        int invalidTokenFound = 0;
        char *token =  strtok(line, " ");
        while (token != NULL) {
            if (!isValidToken(token)) {
                invalidTokenFound = 1;
                break;
            }
            int tok_len = strlen(token);
            char *token_mem = malloc(tok_len + 1);
            strcpy(token_mem, token);
            results[numTokens] = token_mem;

            // get next token
            token = strtok(NULL, " ");
            numTokens++;
        }
        tok_list tokens;
        if (invalidTokenFound) {
            // deallocate unused memory
            int i = 0;
            for (i = 0; i < numTokens; i++) {
                free(results[i]);
            }
            free(results);
            tokens.tokens = NULL;
            tokens.size = 1;
        } else {
            tokens.tokens = (char **) realloc(results, numTokens * sizeof(char*));
            tokens.size = numTokens;
        }
        return tokens;
    }

    /**
     * Validate a token as an operator or word
     */
```

```
int isValidToken(char token[]) {
    int len = strlen(token);
    // check if it's an operator:
    if (len == 1 && (token[0] == '<' || token[0] == '>' || token[0] == '|')) {
        return 1;
    }
    //check if it's a word
    int isWord = 1;
    int i;
    for (i = 0; i < len; i++) {
        isWord &= isValidWordChar(token[i]);
    }
    return isWord;
}

/**
 * Check if a character is valid for a word
 */
int isValidWordChar(char c) {
    if (c >= 'a' && c <= 'z')
        return 1;
    if (c >= 'A' && c <= 'Z')
        return 1;
    if (c >= '0' && c <= '9')
        return 1;
    if (c == '.' || c == '-' || c == '/' || c == '_')
        return 1;
    return 0;
}

/**
 * Parse tokens into token groups aka commands
 */
tok_group_list parseTokens(tok_list tokens_list) {
    char **tokens = tokens_list.tokens;
    // count max number of groups (num of "|" + 1);
    int numGroups = 1;
    int i;
    for (i = 0; i < tokens_list.size; i++) {
        if (strEqual("|", tokens[i]))
            numGroups++;
    }
    tok_group *groups = malloc(numGroups * sizeof(tok_group));
    // iterate through tokens and form groups
    int tok_iter = 0;
    int group_iter = 0;
    int parse_error = 0;
    while (tok_iter < tokens_list.size && !parse_error) {
        tok_group group;
        group.input_redirect = 0;
        group.output_redirect = 0;
        // read command
        if (isOpString(tokens[tok_iter])) {
            parse_error = 1;
            break;
        }
```

```
    group.command = tokens[tok_iter];
    tok_iter++;
    // read arguments
    group.args = readArgs(tokens_list, &tok_iter);
    // handle redirects and check valid redirect syntax
    while (tok_iter < tokens_list.size && !strEqual(tokens[tok_iter], "|"))
        {
        if (strEqual(tokens[tok_iter], "<") && !group.output_redirect) {
            group.input_redirect = 1;
            tok_iter++;
            // parse error if file name is not a word or end of group
            if (tok_iter >= tokens_list.size || isOpString(tokens[tok_iter]
                )) {
                parse_error = 1;
                break;
            }
            group.input_file = tokens[tok_iter];
            tok_iter++;
        }
        else if (strEqual(tokens[tok_iter], ">")) {
            group.output_redirect = 1;
            tok_iter++;
            // parse error if file name is not a word
            if (tok_iter >= tokens_list.size || isOpString(tokens[tok_iter]
                )) {
                parse_error = 1;
                break;
            }
            group.output_file = tokens[tok_iter];
            tok_iter++;
        } else {
            parse_error = 1;
            break;
        }
    }
    if (group.input_redirect && group_iter > 0) {
        parse_error = 1;
    }
    if (group.output_redirect && tok_iter < tokens_list.size && strEqual
        (tokens[tok_iter], "|")) {
        parse_error = 1;
    }
    // iterate past "|" separator and make sure that there is still a field
    // group after to process if there is a pipe
    if (tok_iter < tokens_list.size && strEqual(tokens[tok_iter], "|")) {
        tok_iter++;
        if (tok_iter >= tokens_list.size) {
            parse_error = 1;
        }
    }
    groups[group_iter] = group;
    group_iter++;
}
tok_group_list result;
if (parse_error) {
    // deallocate unused memory
```

```
        for (i = 0; i < group_iter; i++) {
            free(groups[i].args.tokens);
        }
        free(groups);
        result.groups = NULL;
        result.size = 1;
    } else {
        result.groups = groups;
        result.size = group_iter;
    }
    return result;
}

/**
 * Returns a tok_list of arguments for the command
 * Since execve requires a list of arguments with arg[0] as the command and
 * with a NULL pointer at the end of the array. This is not included in the
     "count".
 */
tok_list readArgs(tok_list tokens_list, int *tok_iter) {
    tok_list args;
    char **tokens = tokens_list.tokens;
    int arg_count = 1;
    while (*tok_iter < tokens_list.size && !strEqual(tokens[*tok_iter], "|")) {
        // arguments ended, redirect detected
        if (isOpString(tokens[*tok_iter])) {
            break;
        }
        arg_count++;
        (*tok_iter)++;
    }
    // +1 for extra NULL terminator
    args.tokens = malloc((arg_count + 1) * sizeof(char*));
    int i;
    for (i = 0; i < arg_count; i++) {
        args.tokens[i] = tokens[*tok_iter - arg_count + i];
    }
    args.tokens[arg_count] = NULL;
    args.size = arg_count;
    return args;
}

/**
 * Runs each command and handles pipes/file redirects.
 * Prints status codes.
 */
void runCommands(tok_group_list groups) {
    pid_t pids[groups.size - 1];
    int **pfds = createPipes(groups.size);
    int i;
    for (i = 0; i < groups.size; i++) {
        if (strEqual(groups.groups[i].command, "exit")) {
            exit(0);
        }
        pid_t pid = fork();
        tok_group group = groups.groups[i];
```

```
            char command[1024 + strlen(group.command)];
            if (pid == 0) { // child process
                if (openFiles(group) < 0) {
                    destroyPipes(pfds, groups.size - 1);
                    printf("Error, file could not be opened\n");
                    exit(1);
                };
                // setup writing to pipe
                if (i < groups.size - 1) {
                    dup2(pfds[i][1], STDOUT_FILENO);
                    close(pfds[i][0]);
                    close(pfds[i][1]);
                }
                //setup reading from pipe:
                if (i > 0) {
                    dup2(pfds[i-1][0], STDIN_FILENO);
                    close(pfds[i-1][0]);
                    close(pfds[i-1][1]);
                }
                if (group.command[0] != '/') {
                    getcwd(command, 1024);
                    strcat(command, "/");
                    strcat(command, group.command);
                } else {
                    strcpy(command, group.command);
                }
                char *const env[] = {"TERM=xterm", 0};
                int error = execve(command, group.args.tokens, env);
                if (error == -1) {
                    exit(errno);
                }
            } else { // parent process
                pids[i] = pid;
                if (i < groups.size - 1) {
                    close(pfds[i][1]);
                }
                if (i > 0) {
                    close(pfds[i-1][0]);
                    close(pfds[i-1][1]);
                }
            }
        }
        int statuses[groups.size];
        destroyPipes(pfds, groups.size - 1);
        for (i = 0; i < groups.size; i++) {
            waitpid(pids[i], &statuses[i], 0);
            int exitCode = WEXITSTATUS(statuses[i]);
            char *command = groups.groups[i].command;
            fprintf(stderr, "%d\n", exitCode);
            if (exitCode == ENOENT) {
                printf("Command %s failed to execute\n", command);
            } else {
                printf("%s exited with exit code %d\n", command, exitCode);
            }
        }
    }
```

```
/**
 * Allocate memory for pipe file descriptors
 */
int** createPipes(int num) {
    int **pfds = malloc(num * sizeof(int*));
    int i;
    for (i = 0; i < num; i++) {
        pfds[i] = malloc(2 * sizeof(int));
        pipe(pfds[i]);
    }
    return pfds;
}

/**
 * free memory holding pipe file descriptors
 */
void destroyPipes(int** pfds, int num) {
    int i;
    for (i = 0; i < num; i++) {
        if (i < num) {
            free(pfds[i]);
        }
    }
    free(pfds);
}

/**
 * Opens files for reading and writing STDIN and STDOUT
 * returns -1 if there's an error opening files, 0 if ok
 */
int openFiles(tok_group group) {
    int in, out;
    if (group.input_redirect) {
        in = open(group.input_file, O_RDONLY);
        if (in < 0) {
            return -1;
        }
        dup2(in, STDIN_FILENO);
        close(in);
    }
    if (group.output_redirect) {
        mode_t mode_rights = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
        out = open(group.output_file, O_WRONLY | O_TRUNC | O_CREAT, mode_rights
            );
        if (out < 0) {
            return -1;
        }
        dup2(out, STDOUT_FILENO);
        close(out);
    }
    return 0;
}

/**
 * Checks if a string is a operator string
```

```
 */
int isOpString(char *str) {
    return strEqual(str, "<") || strEqual(str, ">") || strEqual(str, "|");
}

/**
 * Returns a boolean int (0 or 1) to check string equality
 */
int strEqual(char *str1, char *str2) {
    return strcmp(str1, str2) == 0;
}

/**
 * Frees allocated memory related to tokens and groups
 */
void freeMemory(tok_list tokens, tok_group_list groups) {
    int i;
    for (i = 0; i < tokens.size; i++) {
        free(tokens.tokens[i]);
    }
    free(tokens.tokens);
    for (i = 0; i < groups.size; i++) {
        free(groups.groups[i].args.tokens);
    }
    free(groups.groups);
}
```