Shell Report

Rounak Das (A2 - 001210501036)
Operating Systems Lab
3/25/15

Introduction:

MyShell is a shell for UNIX-like Operating Systems written in C. It uses the readline library provided in most linux distributions for reading commands.

Compiling MyShell from source requires libreadline-dev to be installed in your machine.

Compile by: `gcc myshell_final.c -o myshell -lreadline`

Features of MyShell include:

1. Installing and executing any Linux executable
2. Setting and passing environment variables to executable
3. Piping commands via the use of the '|' delimiter (So far only one chaining possible)
4. Input and Output redirection via '>', '<', '>>'
5. Conditional command execution via '&&', '||'
6. Command History (using readline)
7. Tab completion (using readline)

MyShell features and examples:

1. The intro text.

```
myshell>help
        **********************MyShell: A rather limited shell**********************

*) SPECIAL COMMANDS:
        --Install your program by using the install command.
                "install program_name program_location".
        --Set your environment variables using the set command.
                "set variable value".
        --Unset your environment variables using the unset command.
                "unset variable".

*) FEATURES:
        --Supports piping
                ONLY ONE PIPE though
                E.g. "cmd1 ... | cmd2 ..."
        --Supports IO redirection
                input redirection: "cmd ... < file"
                output redirection: "cmd ... > file" or "cmd ... >> file"
        --Supports conditional command execution
                run next command on success "cmd1 ... && cmd2 ..."
                run next command on failure "cmd1 ... || cmd2 ..."

*) BUGS:
        --Having more than one pipe leads to stray processes that don't terminate
        --A command may only be followed by A SINGLE command delimiter i.e &&, ||, |, > etc.
                so something like "cmd1 ... > file && cmd2 ..." is not allowed

***I take no responsibilty if the lack of features of this shell frustrates you into punching your monitor.***
                                Good luck!
myshell>
```

'help' describes the features and syntax of the shell in brief

2. The `'ls'` command lists the files in the current directory or the directory provided as argument

```
myshell>ls
myshell_final.c echoes.txt      ls      envtest.c       envtest makefile        locs    echo.c  ls.c    echo    pwd.c   Other   pwd     checkpatch.sh   final_myshell
myshell>ls -l
-       664     22029   myshell_final.c
-       664     1023    echoes.txt
-       775     11794   ls
-       664     237     envtest.c
-       775     9743    envtest
-       664     612     makefile
-       664     455     locs
-       664     1719    echo.c
-       664     1721    ls.c
-       775     14709   echo
-       664     219     pwd.c
d       775     4096    Other
-       775     9829    pwd
-       764     345     checkpatch.sh
-       775     30770   final_myshell

myshell>ls -al
-       664     22029   myshell_final.c
-       664     1023    echoes.txt
d       775     4096    .git
-       775     11794   ls
-       664     237     envtest.c
-       775     9743    envtest
d       775     4096    ..
d       775     4096    .
-       664     612     makefile
-       664     455     locs
-       664     1719    echo.c
-       664     1721    ls.c
-       775     14709   echo
-       664     219     pwd.c
d       775     4096    Other
-       775     9829    pwd
-       764     345     checkpatch.sh
-       775     30770   final_myshell

myshell>
```

   a. The `ls` command provides 2 switches
      - The `-l` switch turns on long listing
      - The `-a` switch enables listing hidden files
   b. Longlisting format:

      `type  permissions      size(bytes)      name`

      i. type is `'d'` for directory, `'-'` for regular file, `'l'` for link
      ii. Permissions are in octal as they are in chmod
         1. `U(rwx) G(rwx) O(rwx)`
      iii. Size is in bytes

3. The ‘echo’ command echoes its input onto stdout.

```
myshell>echo this is a test
this is a test
myshell>
myshell>echo "This is also a test"
This is also a test
myshell>
myshell>echo -n No newline after output.
No newline after output. myshell>
myshell>
myshell>echo -e "escape \n sequences \t are recognised"
escape
 sequences       are recognised
myshell>
myshell>echo -ne "Getopts\nis awesome"
Getopts
is awesome myshell>
myshell>
myshell>
```

The ‘echo’ command provide 2 switches

- The -n switch omits the appended new line at the end of output.
- The -e switch enables checking for escape characters
  - \b – backspace
  - \n – new line
  - \t – tab
  - \a – bell
  - \r – carriage return
  - \v – vertical tab
  - \\ – a forward slash

4. The `pwd` and `cd` commands, and `PS`
    a. The `pwd` command prints the current working directory to `stdout`
    b. The `cd` command changes the current working directory to its provided argument
        i. If no argument is provided then the CWD changes to the location pointed to by the HOME environment variable
        ii. Spaces in the provided path need to be escaped or the whole argument needs to be enclosed in double quotes
    c. The `PS` is the prompt string (printed when the shell is waiting for a command)
        i. By default this is set to `'myshell'`
        ii. Is this is set to `'pwd'` then the current working directory is printed as the `PS`

```
myshell>set PS pwd
/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell>
/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell>
/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell>cd /home/rounak/
/home/rounak>
/home/rounak>
/home/rounak>pwd
/home/rounak

/home/rounak>cd /
/>
/>
/>pwd
/
/>cd

/home/rounak>set PS myshell
myshell>
myshell>pwd
/home/rounak
myshell>
myshell>env
PS = myshell
USER = rounak
PWD = /home/rounak
TERM = xterm
HOME = /home/rounak
myshell>
```

5. The `set, unset` and `env` built-in commands
    a. `set variable value` – sets the value of the variable called `'variable'` to `'value'`, creating a new variable if it didn't already exist
    b. `unset variable` – unsets the variable called `'variable'`; this variable is not passed to subsequent executed commands
    c. the `env` command prints the current list of environments variables being passed to commands

```
myshell>env
PS = myshell
USER = rounak
PWD = /home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell
TERM = xterm
HOME = /home/rounak
myshell>
myshell>set PS "here is my new PS"
here is my new PS>
here is my new PS>
here is my new PS>unset TERM
here is my new PS>
here is my new PS>env
PS = here is my new PS
USER = rounak
PWD = /home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell
HOME = /home/rounak
here is my new PS>
here is my new PS>set HOME  /home/rounak/Dropbox/

here is my new PS>env
PS = here is my new PS
USER = rounak
PWD = /home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell
HOME = /home/rounak/Dropbox/
here is my new PS>clear
TERM environment variable not set.
here is my new PS>set TERM xterm

here is my new PS>env
PS = here is my new PS
USER = rounak
PWD = /home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell
TERM = xterm
HOME = /home/rounak/Dropbox/
here is my new PS>clear
here is my new PS>
```

6. The install command
    a. `install program location` – sets the location of the command '`program`' to '`location`'; this new program will not show "`Command not found`" from now on.
    b. An equivalent(and safer) way to install a program is to edit the '`locs`' file provided.
        i. Add a line of the format '`program:location`' to install program

```
myshell>cat locs
ls:/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell/ls
pwd:/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell/pwd
echo:/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell/echo
envtest:/home/rounak/Dropbox/Current Sem/Operating Systems/Lab/MyShell/envtest
clear:/usr/bin/clear
reset:/usr/bin/reset
cat:/bin/cat
cp:/bin/cp
rm:/bin/rm
mv:/bin/mv
wc:/usr/bin/wc
true:/bin/true
false:/bin/false
awk:/bin/awk
tr:/usr/bin/tr
myshell>
myshell>
```

7. Piping
    a. Using the '`|`' delimiter between commands one can pipe the output of one command into the input of another
    b. Due to an unknown issue this feature is restricted to only single pipes
        i. i.e. '`cmd1 ... | cmd2 ... | cmd3 ...`' will not work now

```
myshell>cat testfile.txt
"So many books, so little time." - Frank zappa

"It was a pleasure to burn." - Farenheit 451, Ray Bradbury

"To die hating them, that was freedom" - 1984, George Orwell

"Good friends, good books, and a sleepy conscience: this is the ideal life." - Mark Twain

"To die would be an awfully big adventure." - J.M. Barrie, Peter Pan

"Not all those who wander are lost." - J.R.R. Tolkien, The Fellowship of the Ring

"Memories warm you up from the inside. But they also tear you apart." - Haruki Murakami, Kafka on the Shore

"It's the possibility of having a dream come true that makes life interesting." - Paulo Coelho, Alchemist
myshell>
myshell>
myshell>cat testfile.txt | wc -l
15
myshell>
myshell>
```

8. Input Output Redirection
    a. Output of a command may be redirected to a file via the '>' delimiter which will rewrite and truncate the destination file (creating it if it did not already exist)

```
myshell>cat testfile.txt | tr [a-z] [A-Z] > anotherfile.txt
myshell>
myshell>cat anotherfile.txt
"SO MANY BOOKS, SO LITTLE TIME." - FRANK ZAPPA

"IT WAS A PLEASURE TO BURN." - FARENHEIT 451, RAY BRADBURY

"TO DIE HATING THEM, THAT WAS FREEDOM" - 1984, GEORGE ORWELL

"GOOD FRIENDS, GOOD BOOKS, AND A SLEEPY CONSCIENCE: THIS IS THE IDEAL LIFE." - MARK TWAIN

"TO DIE WOULD BE AN AWFULLY BIG ADVENTURE." - J.M. BARRIE, PETER PAN

"NOT ALL THOSE WHO WANDER ARE LOST." - J.R.R. TOLKIEN, THE FELLOWSHIP OF THE RING

"MEMORIES WARM YOU UP FROM THE INSIDE. BUT THEY ALSO TEAR YOU APART." - HARUKI MURAKAMI, KAFKA ON THE SHORE

"IT'S THE POSSIBILITY OF HAVING A DREAM COME TRUE THAT MAKES LIFE INTERESTING." - PAULO COELHO, ALCHEMIST
myshell>
myshell>
```

    b. Output of a command may be appended to an existing file via the '>>' delimiter.

```
myshell>echo this is a line of text > eg.txt
myshell>cat eg.txt
this is a line of text
myshell>echo this is another line of text >> eg.txt
myshell>
myshell>cat eg.txt
this is a line of text
this is another line of text
myshell>
myshell>
```

c. A command may be made to take its input from a file via the '<' delimiter

```
myshell>
myshell>tr [a-z] [A-Z] < testfile.txt
"SO MANY BOOKS, SO LITTLE TIME." - FRANK ZAPPA

"IT WAS A PLEASURE TO BURN." - FARENHEIT 451, RAY BRADBURY

"TO DIE HATING THEM, THAT WAS FREEDOM" - 1984, GEORGE ORWELL

"GOOD FRIENDS, GOOD BOOKS, AND A SLEEPY CONSCIENCE: THIS IS THE IDEAL LIFE." - MARK TWAIN

"TO DIE WOULD BE AN AWFULLY BIG ADVENTURE." - J.M. BARRIE, PETER PAN

"NOT ALL THOSE WHO WANDER ARE LOST." - J.R.R. TOLKIEN, THE FELLOWSHIP OF THE RING

"MEMORIES WARM YOU UP FROM THE INSIDE. BUT THEY ALSO TEAR YOU APART." - HARUKI MURAKAMI, KAFKA ON THE SHORE

"IT'S THE POSSIBILITY OF HAVING A DREAM COME TRUE THAT MAKES LIFE INTERESTING." - PAULO COELHO, ALCHEMIST
myshell>
myshell>wc < testfile.txt
 15 112 629
myshell>
myshell>
```

9. Conditional execution
   a. `'cmd1 … && cmd2 …'` : `'cmd2'` will only execute if and only if `'cmd1'` finishes and returns a true (i.e. zero) exit status.
   b. `'cmd1 … | | cmd2 …'` : `'cmd2'` will only execute if and only if `'cmd1'` finishes and returns a false (i.e. non-zero) exit status.

```
myshell>true && echo this will be printed
this will be printed
myshell>
myshell>false && echo this will not be printed
myshell>
myshell>true || echo nor will this
myshell>
myshell>false || echo but this will be
but this will be
myshell>
myshell>
```

10. Delimiters
    a. The strings `'|', '>', '<', '>>', '&&', '||'` are treated as delimiters
    b. Two commands may only be separated by a single delimiter
    c. Any further delimiters(and in fact commands) are ignored

```
myshell>cat testfile.txt && wc -l < testfile.txt
"So many books, so little time." - Frank zappa

"It was a pleasure to burn." - Farenheit 451, Ray Bradbury

"To die hating them, that was freedom" - 1984, George Orwell

"Good friends, good books, and a sleepy conscience: this is the ideal life." - Mark Twain

"To die would be an awfully big adventure." - J.M. Barrie, Peter Pan

"Not all those who wander are lost." - J.R.R. Tolkien, The Fellowship of the Ring

"Memories warm you up from the inside. But they also tear you apart." - Haruki Murakami, Kafka on the Shore

"It's the possibility of having a dream come true that makes life interesting." - Paulo Coelho, Alchemist
15
myshell>
```

Source code:

- myshell_final.c

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<readline/readline.h>
#include<readline/history.h>

#define MAX_COMMAND_LEN 1024
#define MAX_PROGS 100
#define MAX_ARGS 200
#define MAX_TOKENS 200
#define MAX_TOKEN_LEN 256
#define MAX_PROG_NAME 64
#define MAX_PROG_LOC 512
#define MAX_ENV_VARS 100
#define ENV_VAR_NMSIZE 256
#define ENV_VAR_SIZE 1024

#define DUMP_CLEAR 42
#define PIPE 36
#define DUMP_APPEND 49
#define TAKE 69
#define RUN_SUCCESS 71
#define RUN_FAILURE 88

#define DELAY 5000000

/* global variables */
char command_line[MAX_COMMAND_LEN];
char *command_tokens[MAX_TOKENS];
int num_tokens;

char *programs[MAX_PROGS];
char *prog_locs[MAX_PROGS];
int num_progs;

char *c_argv[MAX_ARGS];
char *c_envp[MAX_ENV_VARS];

int ps_is_pwd;

/* environment variables */
char *env_variables[MAX_ENV_VARS];
char *env_var_values[MAX_ENV_VARS];
int env_var_count;
char *prompt_string;   /* buffer for regular use in prompt subroutine */
char *pwd;             /* buffer for regular use in redirection */

/** Update the prompt string if set has been used **/
void update_prompt_string(void)
{
        int i = 0;

        for (; i < env_var_count; i++) {
                if (strcmp(env_variables[i], "PS") == 0) {
                        strcpy(prompt_string, env_var_values[i]);
                        return;
                }
        }
        perror("PS not found in list of environment variables");
}

/** Update the pwd if set has been used **/
void update_pwd_string(void)
{
        int i = 0;

        for (; i < env_var_count; i++) {
```

```
                if (strcmp(env_variables[i], "PWD") == 0) {
                        strcpy(pwd, env_var_values[i]);
                        return;
                }
        }
        perror("PS not found in list of environment variables");
}

/** Trailing whitespace removal **/
void rstrip(char *str)
{
        int len = strlen(str)-1;

        while (str[len] == '\n' || str[len] == '\t' || str[len] == ' ')
                len--;
        str[len+1] = 0;
}

/*--------------------------------------------------------------------------------*/
/** Built-in commands **/
void change_directory(char *whereto go)
{
        int i = 0;

        if (!whereto go) {
                for (; i < env_var_count; i++) {
                        if (env_variables[i]) {
                                if (strcmp(env_variables[i], "HOME") == 0)
                                        chdir(env_var_values[i]);
                        }
                }
        } else {
                int e = chdir(whereto go);

                if (e != 0)
                        perror("chdir() error");
        }

        for (i = 0; i < env_var_count; i++) {
                if (strcmp(env_variables[i], "PWD") == 0)
                        getcwd(env_var_values[i], ENV_VAR_SIZE);
        }
        update_pwd_string();
        if (ps_is_pwd)
                strcpy(prompt_string, pwd);
}

void print_environment(void)
{
        int i;

        for (i = 0; i < env_var_count; i++)
                if (env_variables[i] && (strlen(env_variables[i]) != 0))
                        printf("%s = %s\n", env_variables[i], env_var_values[i]);
}

void set_env_variable(char *key, char *value)
{
        int i;

        for (i = 0; i < env_var_count; i++) {
                /*printf("ev: %s\n", env_variables[i]);*/
                if (env_variables[i]) {
                        /*printf("ev: %s\n", env_variables[i]);*/
                        if (strcmp(key, env_variables[i]) == 0) {
                                if ((strcmp(key, "PS") == 0) && (strcmp(value, "pwd") == 0)) {
                                        strcpy(prompt_string, pwd);
                                        ps_is_pwd = 1;
                                        return;
                                }
                                if (strcmp(key, "PS") == 0)
                                        ps_is_pwd = 0;
                                free(env_var_values[i]);
                                env_var_values[i] = (char *)malloc(ENV_VAR_SIZE);
                                strcpy(env_var_values[i], value);
                                update_prompt_string();
                                return;
                        }
                }
```

```
        }
        if (env_var_count != MAX_ENV_VARS) {
                i = 0;

                while ((env_variables[i] != NULL) && (strlen(env_variables[i]) != 0))
                        i++;
                env_variables[i] = (char *)malloc(ENV_VAR_NMSIZE);
                env_var_values[i] = (char *)malloc(ENV_VAR_SIZE);
                strcpy(env_variables[i], key);
                strcpy(env_var_values[i], value);
                env_var_count++;

        } else {
                printf("Can not set any more environment variables. Sorry.");
        }
}

void unset_env_variable(char *key)
{
        int i;

        for (i = 0; i < env_var_count; i++) {
                if (env_variables[i]) {
                        if (strcmp(key, env_variables[i]) == 0) {
                                free(env_variables[i]);
                                free(env_var_values[i]);
                                env_variables[i] = malloc(1);
                                env_variables[i][0] = 0;
                                env_var_values[i] = NULL;
                                if (strcmp(key, "PS") == 0)
                                        set_env_variable("PS", "");   /*do not unset PS*/
                                return;
                        }
                }
        }
        printf("Could not find specified variable.");
}

void install(char *progname, char *progloc)
{
        programs[num_progs] = (char *)malloc(MAX_PROG_NAME);
        prog_locs[num_progs] = (char *)malloc(MAX_PROG_LOC);
        strcpy(programs[num_progs], progname);
        strcpy(prog_locs[num_progs], progloc);
        num_progs++;
        FILE *progfile = fopen("locs", "a");

        if (!progfile) {
                perror("Could not open locs file");
                exit(EXIT_FAILURE);
        }

        fprintf(progfile, "%s:%s\n", progname, progloc);
        fclose(progfile);
}

void show_help(void)
{
        printf("\t************************MyShell: A rather limited shell************************\n\n");  /*how
british of me*/
        printf("*) SPECIAL COMMANDS:\n");
        printf("\t--Install your program by using the install command.\n\t\t\"install program_name
program_location\".\n");
        printf("\t--Set your environment variables using the set command.\n\t\t\"set variable value\".\n");
        printf("\t--Unset your environment variables using the unset command.\n\t\t\"unset variable\".\n\n");
        printf("*) FEATURES:\n");
        printf("\t--Supports piping\n\t\tONLY ONE PIPE though\n\t\tE.g. \"cmd1 ... | cmd2 ...\"\n");
        printf("\t--Supports IO redirection\n\t\tinput redirection: \"cmd ... < file\"\n\t\toutput
redirection: \"cmd ... > file\" or \"cmd ... >> file\"\n");
        printf("\t--Supports conditional command execution\n\t\trun next command on success \"cmd1 ... && cmd2
...\"\n\t\trun next command on failure \"cmd1 ... || cmd2 ...\"\n\n");
        printf("*) BUGS:\n");
        printf("\t--Having more than one pipe leads to stray processes that don't terminate\n");
        printf("\t--A command may only be followed by A SINGLE command delimiter i.e &&, ||, |, > etc.\n\t\tso
something like \"cmd1 ... > file && cmd2 ...\" is not allowed\n\n");
        printf("***I take no responsibilty if the lack of features of this shell frustrates you into punching
your monitor.***\n\t\t\t\tGood luck!\n");
}
```

```
/*-----------------------------------------------------------------------------------------*/
/** Set the default environment variables and their default values **/
void set_default_env_vars(void)
{
        env_variables[0] = (char *)malloc(ENV_VAR_NMSIZE);
        strcpy(env_variables[0], "PS");
        env_var_values[0] = (char *)malloc(ENV_VAR_SIZE);
        strcpy(env_var_values[0], "myshell");
        prompt_string = env_var_values[0];

        env_variables[1] = (char *)malloc(ENV_VAR_NMSIZE);
        strcpy(env_variables[1], "USER");
        env_var_values[1] = (char *)malloc(ENV_VAR_SIZE);
        getlogin_r(env_var_values[1], ENV_VAR_SIZE);

        env_variables[2] = (char *)malloc(ENV_VAR_NMSIZE);
        strcpy(env_variables[2], "PWD");
        env_var_values[2] = (char *)malloc(ENV_VAR_SIZE);
        getcwd(env_var_values[2], ENV_VAR_SIZE);
        pwd = env_var_values[2];

        env_variables[3] = (char *)malloc(ENV_VAR_NMSIZE);
        strcpy(env_variables[3], "TERM");
        env_var_values[3] = (char *)malloc(ENV_VAR_SIZE);
        strcpy(env_var_values[3], "xterm");

        env_variables[4] = (char *)malloc(ENV_VAR_NMSIZE);
        strcpy(env_variables[4], "HOME");
        env_var_values[4] = (char *)malloc(ENV_VAR_SIZE);
        strcpy(env_var_values[4], getenv("HOME"));

        env_var_count = 5;
}

/** Initialize the environment variables and build installed program list **/
void init(void)
{
        set_default_env_vars();

        num_progs = 0;
        ps_is_pwd = 0;
        int i;

        for (i = 0; i < MAX_PROGS; i++) {
                programs[i] = (char *)malloc(MAX_PROG_NAME);
                prog_locs[i] = (char *)malloc(MAX_PROG_LOC);
        }

        FILE *progfile = fopen("locs", "r");

        if (!progfile) {
                perror("Could not open locs file");
                exit(EXIT_FAILURE);
        }

        char buf[256];

        char *split_line;

        while (fgets(buf, sizeof(buf), progfile)) {
                split_line = strtok(buf, ":");
                strcpy(programs[num_progs], split_line);
                split_line = strtok(NULL, ":");
                rstrip(split_line);
                strcpy(prog_locs[num_progs], split_line);
                num_progs++;
        }
        fclose(progfile);
}

/** Show Prompt **/
void prompt(void)
{
        printf("%s>", prompt_string);
}

/** Handle signal **/
void handle_signal(int signo)
```

```
{
        prompt();
        fflush(stdout);
}

/** Clear command string and tokens **/
void clear_command(void)
{
        command_line[0] = 0;
        /*free(command_line);*/
        int i;

        for (i = 0; i < num_tokens; i++) {
                free(command_tokens[i]);
                command_tokens[i] = NULL;
        }
        num_tokens = 0;
}

/** checking if a character is whitespace **/
int is_whitespace(char c)
{
        switch (c) {
        case '\t':
        case ' ':
                return 1;
        default:
                return 0;
        }
}

/** Tokenize the line read **/
void tokenize_command(void)
{
        if (!command_line) {
                frpintf(stderr, "Nul passed to tokenizer\n");
                return;
        }
        char *acc = (char *)calloc(MAX_TOKEN_LEN, sizeof(char));

        int quoted = 0, escaped = 0, i = 0, j = 0, k = 0, l;

        char cur = command_line[i];

        while (cur) {
                /*printf("@cur: %c :::: acc: %s, quoted: %d, escaped: %d\n", cur, acc, quoted, escaped);*/
                if (cur == '\"') {
                        quoted++;
                        cur = command_line[++i];
                        continue;
                }
                if (quoted == 1) {
                        acc[j++] = cur;
                        cur = command_line[++i];
                        continue;
                } else if (quoted == 2) {
                        l = strlen(acc);
                        if (l > 0) {
                                command_tokens[k] = (char *)malloc(l*sizeof(char));
                                strcpy(command_tokens[k], acc);
                                ++k;
                                free(acc);
                                acc = (char *)calloc(MAX_TOKEN_LEN, sizeof(char));
                                j = 0;
                        }
                        quoted = 0;
                }

                if (cur == '\\') {
                        escaped = 1;
                        cur = command_line[++i];
                        continue;
                }

                if (escaped) {
                        acc[j++] = cur;
                        cur = command_line[++i];
                        escaped = 0;
                        continue;
```

```
                }
                if (is_whitespace(cur)) {
                        l = strlen(acc);
                        if (l > 0) {
                                command_tokens[k] = (char *)malloc(l*sizeof(char));
                                strcpy(command_tokens[k], acc);
                                ++k;
                                free(acc);
                                acc = (char *)calloc(MAX_TOKEN_LEN, sizeof(char));
                                j = 0;
                        }
                } else {
                        acc[j++] = cur;
                }
                cur = command_line[++i];
        }
        l = strlen(acc);
        if (l > 0) {
                command_tokens[k] = (char *)malloc(l*sizeof(char));
                strcpy(command_tokens[k], acc);
                ++k;
                free(acc);
                acc = (char *)calloc(MAX_TOKEN_LEN, sizeof(char));
                j = 0;
        }
        num_tokens = k;
}

/** Search command_name against list of installed programs in locs **/
int is_installed(int index)
{
        int i;

        for (i = 0; i < num_progs; i++)
                if (strcmp(programs[i], command_tokens[index]) == 0)
                        return i;
        return -1;
}

/** Run the installed executable with the rest of the tokens passed as arguments **/
void execute_single_command(void)
{
        int loc_index = is_installed(0);

        if (loc_index != -1) {
                char *com = prog_locs[loc_index];
                int i = 1, t, j;

                /*set arguments*/
                for (; i < num_tokens; i++)
                        c_argv[i] = command_tokens[i];
                c_argv[0] = com;
                c_argv[i] = NULL;

                /*set environment*/
                for (i = 0, j = 0; i < env_var_count; i++)
                        if (env_variables[i]) {
                                c_envp[j] = (char *)malloc(1024);
                                sprintf(c_envp[j], "%s=%s", env_variables[i], env_var_values[i]);
                                j++;
                        }
                c_envp[j] = NULL;

                pid_t pid = fork();

                if (pid == 0) {
                        execve(com, c_argv, c_envp);
                        perror("execve failed");
                        exit(EXIT_FAILURE);
                } else if (pid > 0) {
                        pid_t cpid;

                        if (wait(NULL) == -1)
                                perror("wait() error");

                } else {
                        perror("fork() failed");
                }
        } else {
```

```
                printf("Command not found!\n");
        }
}

/** check if a command token is a pipe '|' or arrow '>', '<', '>>' or conditional '&&', '||' **/
int is_delimiter(char *token)
{
        if (!token) {
                printf("NULL passed\n");
                exit(EXIT_FAILURE);
        }
        if (strcmp(token, "|") == 0)
                return PIPE;
        else if (strcmp(token, ">") == 0)
                return DUMP_CLEAR;
        else if (strcmp(token, "<") == 0)
                return TAKE;
        else if (strcmp(token, ">>") == 0)
                return DUMP_APPEND;
        else if (strcmp(token, "&&") == 0)
                return RUN_SUCCESS;
        else if (strcmp(token, "||") == 0)
                return RUN_FAILURE;
        else
                return 0;
}

/** open a token as a file for redirection, return a file descriptor **/
int open_next_token(char *filename, int decide)
{
        char *filepath = malloc(4096);

        if (strchr(filename, '/')) {  /*if the token contains a '/' treat it as a path*/
                strcpy(filepath, filename);
        } else {       /*else use relative addressing*/
                strcpy(filepath, pwd);
                strcat(filepath, "/");
                strcat(filepath, filename);
        }
        int fd;

        /*fprintf(stderr,"filepath: %s\n", filepath);*/

        switch (decide) {
        case TAKE:
                fd = open(filepath, O_RDONLY);
                break;
        case DUMP_APPEND:
                fd = open(filepath, O_WRONLY | O_APPEND);
                break;
        case DUMP_CLEAR:
                fd = open(filepath, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
S_IROTH);
                break;
        default:
                fd = -1;
        }

        if (fd == -1) {
                perror("open failed");
                exit(EXIT_FAILURE);
        }

        /*fprintf(stderr,"fd: %d\n", fd);*/

        return fd;
}

/** This function forks the children **/
void execute_command_chain(void)
{
        /*set environment*/
        int i, j, k;

        for (i = 0, j = 0; i < env_var_count; i++) {
                if (env_variables[i]) {
                        c_envp[j] = (char *)malloc(1024);
                        sprintf(c_envp[j], "%s=%s", env_variables[i], env_var_values[i]);
                        j++;
```

```
                }
        }
        c_envp[j] = NULL;

        /* Flags and utilities
        numcom    = number of commands to run
        to_pipe   = does the current command go to a pipe
        from_pipe = does the current command come from a previous pipe
        success   = is the current command subject to '&&'
        failure   = is the current command subject to '||'
        oldpipe   = a pipe (the pipe set up in the previous iteration; used for input if from-pipe is set)
        newpipe   = another pipe (the pipe set up in this iteration ; used for output if to_pipe is set)
        fdi       = the file descriptor used for input redirection
        fdo       = the file descriptor used for output redirection
        */
        int numcom = 1, child_num = 0; /*number of commands seen so far, and the number of children forked so
far*/
        int to_pipe = 0, from_pipe, success = 0, failure = 0;
        int oldpipe[2], newpipe[2], fdi, fdo;
        int was_piped = 0, was_redirected = 0;
        int loc_index, next;

        i = 0;

        /*decide flags for the first command*/
        next = i;
        int decide = is_delimiter(command_tokens[next]);

        while (1) {
                if ((decide != 0) || next == num_tokens-1) /*break if delimiter found or this is the last
token*/
                        break;
                ++next;
                decide = is_delimiter(command_tokens[next]);
        }

        switch (decide) {
        case PIPE:
                to_pipe = 1;
                was_piped = 1;
                numcom++;
                break;
        case RUN_SUCCESS:
                success = 1;
                numcom++;
                break;
        case RUN_FAILURE:
                failure = 1;
                numcom++;
                break;
        default:
                /*nothing to do here*/
                break;
        }

        from_pipe = 0; /*first command doesn't come from a pipe*/

        do {
                /*printf("\nStart of shell loop:\n");
                printf("to_pipe: %d, from_pipe: %d, numcom: %d, success: %d, failure: %d\n", to_pipe,
from_pipe, numcom, success, failure);
                printf("i: %d, next: %d\n", i, next);*/

                loc_index = is_installed(i);
                if (loc_index == -1) {
                        fprintf(stderr, "Command not found\n");
                        return;
                }

                /*set up argv for this command*/
                j = 1;
                k = i+1;
                if (next == num_tokens-1)
                        ++next;
                for (; k < next; k++, j++)
                        c_argv[j] = command_tokens[k];
                char *com = prog_locs[loc_index];

                c_argv[0] = com;
```

```
                c_argv[j] = NULL;

                /*take care of the pipes, save the pipe of the last iteration in oldpipe*/
                oldpipe[0] = newpipe[0];
                oldpipe[1] = newpipe[1];
                if (to_pipe)                    /*create a new pipe if needed*/
                        if (pipe(newpipe) == -1) {
                                perror("Pipe error");
                                exit(EXIT_FAILURE);
                        }

                /*printf("old: <-%d==%d<-\n",oldpipe[0], oldpipe[1]);
                printf("new: <-%d==%d<-\n",newpipe[0], newpipe[1]);*/

                /*fork here*/
                pid_t cpid = fork();

                numcom--;
                if (cpid < 0) {
                        perror("fork failed.");
                        exit(EXIT_FAILURE);
                }

                if (cpid == 0) {        /*child*/
                        /*handle the input side*/
                        if (from_pipe) {
                                /*if child comes from a pipe, use oldpipe for input*/
                                /*printf("%s is gonna pipe its input from %d=%d\n", com, oldpipe[0],
oldpipe[1]);*/
                                was_redirected = 1;
                                if (dup2(oldpipe[0], STDIN_FILENO) == -1) {
                                        perror("dup2 error");
                                        exit(EXIT_FAILURE);
                                }
                        } else if (decide == TAKE) {
                                /*use next token as file for input*/
                                /*printf("%s is gonna take its input from %s/%s\n", com, pwd,
command_tokens[next+1]);*/
                                was_redirected = 1;
                                fdi = open_next_token(command_tokens[next+1], decide);
                                if (dup2(fdi, STDIN_FILENO) == -1) {
                                        perror("dup2 error");
                                        exit(EXIT_FAILURE);
                                }
                                close(fdi);
                        } else {
                                /*use stdin*/
                                /*printf("%s is gonna take its input from stdin/args\n", com);*/
                        }

                        /*handle the output side*/
                        if (to_pipe) {
                                /*if child comes from a pipe, use newpipe for output*/
                                /*printf("%s is gonna pipe its output to %d=%d\n", com, newpipe[0],
newpipe[1]);*/
                                if (dup2(newpipe[1], STDOUT_FILENO) == -1) {
                                        perror("dup2 error");
                                        exit(EXIT_FAILURE);
                                }
                        } else if (decide == DUMP_CLEAR) {
                                /*use next token as file for output */
                                /*printf("%s is gonna dump its output to %s/%s\n", com, pwd,
command_tokens[next+1]);*/
                                was_redirected = 1;
                                fdo = open_next_token(command_tokens[next+1], decide);
                                if (dup2(fdo, STDOUT_FILENO) == -1) {
                                        perror("dup2 error");
                                        exit(EXIT_FAILURE);
                                }
                                close(fdo);
                        } else if (decide == DUMP_APPEND) {
                                /*use next token as file for output */
                                /*printf("%s is gonna append its output to %s/%s\n", com, pwd,
command_tokens[next+1]);*/
                                was_redirected = 1;
                                fdo = open_next_token(command_tokens[next+1], decide);
                                if (dup2(fdo, STDOUT_FILENO) == -1) {
                                        perror("dup2 error");
                                        exit(EXIT_FAILURE);
```

```
                        }
                        close(fdo);
                } else {
                        /*use stdout*/
                        /*printf("%s is gonna use stdout for output\n", com);*/
                }

                /*execute the process*/
                /*int l=0;
                printf("******argv:*******\n");
                while(c_argv[l]) {
                        printf("%s\n", c_argv[l]);
                        l++;
                }
                printf("******envp:*******\n");
                while(c_envp[l]) {
                        printf("%s\n", c_envp[l]);
                        l++;
                }*/

                /*close all the unused fds*/
                if (to_pipe) {
                        close(newpipe[0]);
                        close(newpipe[1]);
                }
                if (from_pipe) {
                        close(oldpipe[0]);
                        close(oldpipe[1]);
                }
                /*actually exec it*/
                execve(com, c_argv, c_envp);
                perror("Nobody expects the spanish inquisition");
                exit(EXIT_FAILURE);

        } else {                /*parent*/
                child_num++;
                if (!numcom)
                        break;

                /*if success or failure flag was set, then wait for current child to finish before
forking the next one*/
                int prev_stat;

                if (success) {
                        if (waitpid(cpid, &prev_stat, 0) == -1) {
                                perror("success wait() failed");
                                exit(EXIT_FAILURE);
                        }
                        child_num--;
                        if (WIFEXITED(prev_stat))
                                if (WEXITSTATUS(prev_stat))
                                        break;
                } else if (failure) {
                        if (waitpid(cpid, &prev_stat, 0) == -1) {
                                perror("failure wait() failed");
                                exit(EXIT_FAILURE);
                        }
                        child_num--;
                        if (WIFEXITED(prev_stat))
                                if (!WEXITSTATUS(prev_stat))
                                        break;
                }

                /*decide flags for next loop here*/
                i = next+1;
                from_pipe = to_pipe;
                ++next;                 /*next is now the command*/
                decide = is_delimiter(command_tokens[next]);
                while (1) {
                        if ((decide != 0) || next == num_tokens-1) /*break if delimiter found or this is
the last token*/
                                break;
                        ++next;
                        decide = is_delimiter(command_tokens[next]);
                }

                to_pipe = success = failure = 0;

                switch (decide) {
```

```
                        case PIPE:
                                to_pipe = 1;
                                was_piped = 1;
                                numcom++;
                                break;
                        case RUN_SUCCESS:
                                success = 1;
                                numcom++;
                                break;
                        case RUN_FAILURE:
                                failure = 1;
                                numcom++;
                                break;
                        default:
                                /*nothing to do here*/
                                break;
                        }
                        /*parent continues with loop, next iteration would use decide, to_pipe, old_pipe*/
                        /*printf("End of shell loop:\n");
                        printf("to_pipe: %d, from_pipe: %d, numcom: %d, success: %d, failure: %d\n", to_pipe,
from_pipe, numcom, success, failure);
                        printf("i: %d, next: %d\n", i, next);*/
                }

        } while (numcom);

        /*printf("\n\nNumber of children: %d\n", child_num);*/

        int status;

        if (child_num) {
                if (waitpid(-1, &status, 0) == -1) {
                        perror("wait() failed");      /*wait for atleast one child to exit*/
                        exit(EXIT_FAILURE);
                }
                for (j = 0; j < child_num-1; j++)
                        if (waitpid(-1, &status, WNOHANG) == -1) {   /*don't wait for zombies*/
                                perror("wait() failed");
                                exit(EXIT_FAILURE);
                        }
        }

        /*close all file descriptors used in this invocation*/
        if (was_piped) {
                close(oldpipe[0]);
                close(newpipe[0]);
                close(oldpipe[1]);
                close(newpipe[1]);
        }
        if (was_redirected) {
                close(fdi);
                close(fdo);
        }

        struct timespec hack_delay;

        hack_delay.tv_sec = 0;
        hack_delay.tv_nsec = DELAY;
        nanosleep(&hack_delay, NULL);
}

/** Search command_name against list of built ins **/
int is_builtin(void)
{
        if (strcmp("exit", command_tokens[0]) == 0) {
                exit(EXIT_SUCCESS);
        } else if (strcmp("help", command_tokens[0]) == 0) {
                show_help();
                return 1;
        } else if (strcmp("cd", command_tokens[0]) == 0) {
                if (num_tokens < 2)
                        change_directory(NULL);
                else if (num_tokens < 3)
                        change_directory(command_tokens[1]);
                else
                        printf("Invalid number of arguments passed to built-in cd");
                return 1;
        } else if (strcmp("env", command_tokens[0]) == 0) {
                print environment();
```

```
                return 1;
        } else if (strcmp("set", command_tokens[0]) == 0) {
                if (num_tokens == 3)
                        set_env_variable(command_tokens[1], command_tokens[2]);
                else
                        printf("Invalid number of arguments passed to built-in set\n");
                return 1;
        } else if (strcmp("unset", command_tokens[0]) == 0) {
                if (num_tokens == 2)
                        unset_env_variable(command_tokens[1]);
                else
                        printf("Invalid number of arguments passed to built-in unset\n");
                return 1;
        } else if (strcmp("install", command_tokens[0]) == 0) {
                if (num_tokens == 3)
                        install(command_tokens[1], command_tokens[2]);
                else
                        printf("Invalid number of arguments passed to built-in install\n");
                return 1;
        }
        return 0;
}


/*---------------------------------------------------------------------------------------*/
/** Main **/
int main(void)
{
        init();
        while (1) {
                signal(SIGINT, handle_signal);
                prompt();
                strcpy(command_line, readline(""));
                if (strcmp(command_line, "") == 0)
                        continue;
                if (*command_line && command_line)
                        add_history(command_line);
                tokenize_command();
                if (!is_builtin())
                        execute_command_chain();
                clear_command();
        }
        return 0;
}
```

ls.c

```
#include<dirent.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<dirent.h>
#include<sys/types.h>
#include<sys/stat.h>

//print info about a single file from stat call
void print_file_info(struct dirent* file, struct stat filestat, int longlist, int all)
{
        if (!all && (file->d_name[0] == '.') ) {
                return;
        }

        char firstchar;
        if (S_ISREG(filestat.st_mode)) {
                firstchar = '-';
        } else if (S_ISDIR(filestat.st_mode)) {
                firstchar = 'd';
        } else if (S_ISLNK(filestat.st_mode)) {
                firstchar = 'l';
        } else {
                firstchar = 'u';
        }

        int mode = filestat.st_mode & 0777;
```

```
        int size = filestat.st_size;

        if (longlist) {
                printf("%c\t%o\t%d\t%s\n", firstchar, mode, size, file->d_name);
        } else {
                printf("%s\t", file->d_name);
        }
}

int main(int argc, char* argv[], char* envp[])
{
        if (argv[1] && (strcmp(argv[1], "--help")) == 0) {
                printf("Usage:\n\t-a: Do not ignore hidden entries\n\t-l: use longlisting\n");
                exit(EXIT_SUCCESS);
        }

        //process options
        int all=0, longlist=0, opt;
        while ((opt = getopt(argc, argv, "al")) != -1) {
                switch (opt) {
                        case 'a':
                                all = 1;
                                break;
                        case 'l':
                                longlist = 1;
                                break;
                        default:
                                printf("Unrecognised option. Please see help.\n");
                                exit(EXIT_FAILURE);
                }
        }

        DIR *argdir;

        //no arguments given
        if (optind >= argc) {
                argdir = opendir(getenv("PWD"));
        } else {
                argdir = opendir(argv[1]);
        }

        if (!argdir) {
                printf("Could not open directory stream\n");
                exit(EXIT_FAILURE);
        }

        struct dirent *files;
        while ((files = readdir(argdir)) != NULL){
                struct stat filestat;
                stat(files->d_name, &filestat);
                print_file_info(files, filestat, longlist, all);
        }
        printf("\n");
        return 0;
}
```

pwd.c

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[], char* envp[])
{
        char cwd[512];
        if (getcwd(cwd, sizeof(cwd)) != NULL)
                printf("%s\n", cwd);
        else
                printf("Could not get CWD\n");
        return 0;
}
```

echo.c

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

int main( int argc, char* argv[], char* envp[] )
{
	//get rid of this later
	if ((strcmp(argv[1], "es"))== 0) {
		system("cat echoes.txt");
		exit(EXIT_SUCCESS);
	}

	//process the options
	int escape = 0, newline = 1, opt;
	while ((opt = getopt(argc, argv, "ne")) != -1) {
		switch (opt){
			case 'n':
				newline = 0;
				break;
			case 'e':
				escape = 1;
				break;
			default:
				printf("Unrecognised option. Please see help.\n");
				exit(EXIT_FAILURE);
		}
	}
	//handle case of no arguments
	if (optind >= argc) {
		if (newline) {
			printf("\n");
		}
		exit(EXIT_SUCCESS);
	}
	//handle each argument
	int i;
	for (i=optind; i<argc; i++) {
		char dest[1024];
		if (!escape) {
			strcpy(dest, argv[i]);
		} else {
			int si=0, di=0, eflag=0;
			for (; si<strlen(argv[i]); si++) {
				if (eflag == 0) {
					if (argv[i][si] != '\\')
						dest[di++] = argv[i][si];
					else
						eflag = 1;
				} else if (eflag == 1) {
					switch (argv[i][si]) {
						case 'b': dest[di++] = '\b'; break;
						case 't': dest[di++] = '\t'; break;
						case 'n': dest[di++] = '\n'; break;
						case 'a': dest[di++] = '\a'; break;
						case 'r': dest[di++] = '\r'; break;
						case 'v': dest[di++] = '\v'; break;
						case '\\': dest[di++] = '\\'; break;
						default: dest[di++] = '\\'; dest[di++] = argv[i][si];
					}
					eflag = 0;
				}
			dest[di] = 0;
			}
		}
		printf("%s ", dest);
	}
	 if (newline) {
		printf("\n");
	 }

	return 0;
}
```