Operating Systems and Systems Programming 1 Assignment

2017/2018

Members: Valerija Holomjova

Course Code: CPS1012-SEM2-A-1718

Table of Contents

Bugs and Issues

Each section of the assignment has been completed except for the 'Process Management' section. It is important to report that the bugs and issues that have been encountered and a solution for which has not been found are the following:

- The input redirection does not function for the 'print' internal command. However, the output redirection functions correctly for both internal and external commands.
- The 'print' Internal command produces errors when piping is involved.
- The 'execvpe()' function could not be used as the warning "Can't resolve variable execvpe" was displayed. I have tried defining '_GNU_SOURCE' before the <unistd.h> library but my macOS and Linux system could not identify the function. It must be noted that, 'execle' was used temporarily to test whether the environment variables were passed through successfully, and 'execvp' has been used as a substitute in the code as it passes the arguments in the same method (as an array). Thus, the code will not pass any environment variables through. The array of environmental variables is prepared before the 'execvp' function but is not used.

The rest of the sections were tested and should function correctly. The report will now proceed by displaying the test cases and results obtained when testing the eggshell, followed by describing the code implemented in each section of the assignment.

Testing the System

Please refer below to the tables showing how the eggshell was tested for each section. The 'input' column displays the commands entered into the eggshell and the 'output' column displays the output produced when the shell executes the commands.

Shell Variables

Input	Output
> USER=valerija > print $USER	valerija
> TEST=example > USER=$TEST > print $USER	example
> print USER	USER

Command Line Interpreter

Internal Commands

Input	Output
> exit	(Program Terminates)
> print The following string is a test.	The following string is a test.
> print The user is $USER.	The user is valerija.
> print "The user is $USER."	The user is $USER.
> print "This is a string.	"This is a string.
> print	Error – No arguments inputted after the command 'print'.
> print $CWD > chdir .. > print $CWD	/home/valerija/Documents/OS Assignment - Valerija Holomjova/Source Code/cmake-build-debug Directory has been changed successfully. /home/valerija/Documents/OS Assignment - Valerija Holomjova/Source Code
> chdir nofile	Error – chdir(): No such file or directory.
> chdir	Error – No arguments inputted after the command 'chdir'.
> all	SHELL=/home/student/cps1012/bin/eggshell USER=valerija PROMPT=> PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin HOME=/home/valerija TERMINAL=/dev/pts/4 CWD=/home/valerija/Documents/OS Assignment - Valerija Holomjova/Source Code/cmake-build-debug
The test.txt file contains the following lines: print This is test.txt print Hello $USER > source test.txt	This is test.txt Hello valerija
> source nofile	Error – fopen(): No such file or directory
> source	Error – No arguments inputted after the command 'source'.

External Commands

Input	Output
> echo test string	test string
The test.txt file contains the following lines: This is a test string. 1, 2, 3 > cat test.txt	This is a test string. 1,2,3
> cat nofile	Cat: nofile: No such file or directory

Please note that C executable files have also been tested in this section and the eggshell successful launched the program.

I/O Redirection

Please note that 'print' internal command does not work for input redirection as shown in the highlighted test below. However, external commands do work for input redirection and both internal and external commands work for output redirection.

Input	Output
> print Hello this is a test > test.txt > print Testing 1,2,3 > test.txt > cat test.txt	Testing 1,2,3
> print Hello > test.txt > print This is another test >> test.txt > cat test.txt	Hello This is another test
> echo Hello > test.txt > echo This is another test using external commands >> test.txt > cat test.txt	Hello This is another test using external commands
> cat <<< This is a here string.	This is a here string.
> cat test.txt > sort < test.txt	(Output for cat test.txt) d c a (Output for sort < test.txt) a c d
> print < test.txt	Error – no arguments inputted after the command 'print'.

Piping

Please note that 'print' internal command does not work for piping as shown in the highlighted test below. However, all the external commands and other internal commands do work for piping. Please note that the eggshell can successfully pipe multiple commands.

Input	Output
> cat test.txt > cat test.txt | sort	(Output for cat test.txt) d c a (Output for sort < test.txt) a c d
> all | sort	CWD=/home/valerija/Documents/OS Assignment - Valerija Holomjova/Source Code/cmake-build-debug HOME=/home/valerija PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin PROMPT=> SHELL=/home/student/cps1012/bin/eggshell TERMINAL=/dev/pts/4 USER=valerija
> ls | head -3 | tail -1	Makefile
> cat test.txt | head | wc -l	1
> print | print Hello	Error – No arguments inputted after the command 'print'. Hello

Command-Line Interpreter

The ‘main()’ function

At the start of the code a header file is included, containing the definitions of all the functions, libraries and global variables that will be used in the implementation of the eggshell. In the main function, the ‘define_var()’ function is used to set up the starting environment variables. For a description of this function please refer to the ‘Shell Variables’ section of this report. Next, the ‘start()’ function is called to commence the loop that will interpret the commands.

```
//Includes all functions, libraries and global variables.
#include "header.h"

int main() {
    define_var(); //Sets up the environment variables.
    start(); //Starts the terminal.
}
```

The ‘start()’ function

This function will enter a command loop which prints the command prompt by displaying the value of the ‘PROMPT’ environment variable using a function called ‘return_var_value()’. Next, a line is read from the command-line through the ‘read_line()’ function and tokenized into individual arguments using the ‘split_line()’ function. These arguments are then executed using the ‘execute()’ function. The loop will cease when the ‘execute()’ function returns the value of 0 which indicates that the user has exited the terminal.

```
void start(){
    char *line, **args;
    int status;
    //Loops until the exit command is typed into shell terminal (returning 0).
    do {
        //Prints the command prompt.
        printf("%s",return_var_value("PROMPT"));
        //Reads line.
        line = read_line();
        //Tokenizes line.
        args = split_line(line);
        //Executes line.
        status = execute(args);
    } while (status != 0);
}
```

The ‘read_line()’ function

This function reads the user’s inputted line from the command-line using the ‘getline()’ library function and returns it as a string.

```
char *read_line(){
    char *line = NULL;
    size_t size = 0;
    getline(&line, &size, stdin);
    return line;
}
```

The ‘split_line ()’ function

This function tokenizes a given line into individual arguments. At the start of the function, memory is allocated for the token array. Next, the first token is extracted using the space, tab, new line, and carriage return as delimiters. In the ‘while’ loop, the extracted token is stored in the appropriate index which iterates through every loop. An ‘if’ statement is used to determine whether more space needs to be allocated to the token array. At the end of the loop, the next token is extracted from the line using the same delimiters. The loop stops once the final token is NULL, signifying the end of the line. Hence, the last element in the tokens array is set to null and the array is returned. This array will represent an array of arguments where each argument is a string.

```
//Tokenizes the line and returns an array of arguments.
char **split_line(char *line){
    int size = MAX_SIZE, index = 0;
    //Allocates space for the array of arguments.
    char **tokens = malloc(size * sizeof(char *));
    char *token;
    //Acquires the first token.
    token = strtok(line, DELIMITERS);
    //Splits string into tokens.
    while(token != NULL) {
        tokens[index] = token;
        index++;
        //Checks if more space needs to be allocated for the array of arguments.
        if (index >= size) {
            size += MAX_SIZE;
            tokens = realloc(tokens, size * sizeof(char *));
        }
        //Acquires the next tokens.
        token = strtok(NULL, DELIMITERS);
    }
    tokens[index] = NULL;
    //Returns an array of tokens.
    return tokens;
}
```

The ‘execute()’ function

This function is responsible for determining which actions to take based on a given array of arguments. It identifies whether there is piping, I/O redirection, variable assignments, internal commands or external commands. At the start of the function, the code checks whether there is a pipe in the commands using the ‘is_pipe()’ function. If this condition is satisfied, the function ‘execute_pipe()’ will be called and returned. This function handles the piping, where the variable ‘first’ is the left-hand arguments of the pipe and the variable ‘second’ is the right-hand arguments of the pipe. Next, an ‘if’ statement checks whether there is a input or output redirection in the commands using the ‘is_redirect()’ function. If this condition is satisfied, the function ‘execute_redirect()’ will be called and returned. This function handles the input and output redirection, where the variable ‘first’ is the left-hand arguments of the redirection and the variable ‘second’ is the right-hand arguments of the redirection and the variable ‘choice’ represents the type of redirection operator found.

```
int execute(char **args){
    char *first[MAX_SIZE], *second[MAX_SIZE];
    int choice;
    //Executes pipe commands.
    if(is_pipe(args,first,second) != 0) {
        return execute_pipe(first, second);
    }
    //Executes redirection commands.
    if((choice = is_redirect(args,first,second)) != 0){
        return execute_redirect(first,second,choice);
    }
```

Subsequently, the code proceeds by checking whether the first argument is an environment variable assignment. If this condition is true, the process will be carried out in the 'is_var_assignment()' function and a value of 1 will be returned. Otherwise, the function will return a value of 0 indicating that there is no environment variable assignment, and the code will resume identifying the arguments.

```
//Executes variable assignment.
if(is_var_assignment(args[0]) != 0){
    return 1;
}
```

Next, the code checks whether the arguments are internal commands or external commands. To determine whether the arguments are internal commands, a 'for' loop is used to access each element in the command name array that has been defined in the header file. If the first argument is equivalent to one of the commands name, the function corresponding to its name is called and returned. This is done using an array of pointers to the command functions which has been defined in the header file. For instance, if the first argument is 'print', the 'print_comm' function will be pointed to as it is on the same index. If none of the argument satisfies none of the above statements, the arguments will be considered an external command and 'launch()' function will be called.

```
//Executes internal commands.
    for (int i = 0; i < sizeof(commands_names) / sizeof(char *); i++) {
        //If the name is one of the in-built function names, executes that
function.
        if (strcmp(args[0], commands_names[i]) == 0)
            return (*commands[i])(args);
    }
    //Executes external commands.
    launch(args);
    return 1;
}
```

The 'get_size_args()' function

This function is used to return the number of arguments that is given. It functions by taking a list of arguments and looping till the last argument, which should be NULL, is reached. It then returns the index of the last argument which represents the number of arguments in the given argument array.

```
//Returns the number of arguments inputted.
int get_size_args(char **args){
    int i = 0;
    //Loops until the current argument is NULL, signifying the last
argument.
    while(args[i]!= NULL){
        i++;
    }
    return i;
}
```

The 'split_args()' function

This function is used to assist in the detection of piping or I/O redirection. Its purpose is to split the arguments array into a left part and right part where a given string acts as a divider. For instance, if the "|" is one of the arguments and chosen as the divider, the function will take the arguments on the left side of the "|" character and store them in the 'first' string array. Then the arguments on the right side of the "|" character will be stored in the 'second' string array. It will then return a value based on whether the array has been successfully split (1) or the character hasn't been found as one of the arguments (0).

```
//Splits the arguments array from a character.
int split_args(char **args, char **first, char **second, char *c){
    int n = get_size_args(args); //Determines size of arguments in args.
    //Goes through every argument.
    for(int i=0;i<n;i++){
        //If the current argument is the character, split the 'args' array from
this index.
        if(strcmp(args[i],c) == 0){
            //Copies the left half of the argument array.
            memcpy(first, args, (i+1) * sizeof(char *));
            first[i] = NULL;
            //Copies the right half of the argument array.
            memcpy(second, args+(i+1), ((n-i)+1) * sizeof(char *));
            second[n] = NULL;
            return 1;
        } else
            continue;
    }
    return 0; //If the character wasn't one of the arguments returns 0.
}
```

The 'is_pipe ()' function

This function is used to determine whether there is a pipe as one of the arguments by passing the arguments through the 'split_args()' function and checking whether it was successful. If the 'split_args()' function is successful, a value of non-zero will be returned as well as the left side and right-side arguments of the pipe. Next, a value of 1 is returned and the two split arguments are passed back through the function parameters. These sections will later be used to execute the pipe. Elsewise, the function will return a value of 0 indicating that there is no pipe found.

```
//Checks if there is a pipe and returns argument split into pipe sections.
int is_pipe(char **args, char **pipe1, char **pipe2){
    //If '|' is found, returns the left and right side of pipe arguments.
    if(split_args(args,pipe1,pipe2,"|") != 0){
        return 1;
    } else
        return 0; //Return 0 if '|' not found.
}
```

The 'is_redirect ()' function

This function is used to determine whether there is I/O redirection in the given list of arguments. It functions by passing the arguments through the 'split_args()' function to check whether either of the operators are one of the arguments. If the redirection operators are one of the operators, the 'split_args()' function will return the left side arguments and right side arguments of the found operator through the function arguments to later be used when executing the redirection. It is important to note that each I\O redirection operator will return a different value. This is so that the appropriate file could be opened later based on the operator found in the 'execute_redirection()' function. It is important to note that the split arguments are also passed through the functions parameters when the function is returned as 'red1' and 'red2'.

```
//Checks if there is redirection and returns argument split by redirect operator.
int is_redirect(char **args, char **red1, char **red2){
    //If a redirection operator is found, returns a value other than 0.
    if (split_args(args,red1,red2, ">") != 0) {
        return 1;
    } else if (split_args(args,red1,red2, ">>") != 0) {
        return 2;
    } else if (split_args(args,red1,red2, "<") != 0) {
        return 3;
    } else if (split_args(args,red1,red2, "<<<") != 0) {
        return 4;
```

```
    } else
        return 0; //Return 0 if none of the operators were found.
}
```

Internal Commands

The ‘exit_comm’ function

This function returns a value of 0 when called which will terminate the command loop in the ‘start()’ function.

```
//The 'exit' internal command - Tells shell to exit.
int exit_comm(char **args){
    return 0;
}
```

The ‘print_comm’ function

This function prints out exact strings, or strings that can contain variable values. If no arguments are inputted after “print”, the function will display an error message. Next, it determines whether it has to print a string exactly how it is, or whether it needs to convert environment variables. It does this by checking whether the beginning and end of the arguments after the “print” command are quotation marks (“). If this condition is true, the function will remove the quotation marks from the first argument to be printed and prints the rest of arguments until the last one. The last argument has the ending quotation marks removed and is printed as well.

```
//The 'print' internal command - Displays variable values and strings.
int print_comm(char **args){
    //Executes if no arguments were inputted after 'print'.
    if(args[1] == NULL){
       fprintf(stderr,"Error -- No arguments inputted after the command
\'print\'.\n");
    } else {
        int index = 1;
        int n = get_size_args(args); //The number of arguments.
        //If it starts and ends with " - variable insensitive.
        if(args[1][0] == '\"' && args[n-1][strlen(args[n-1])-1] == '\"'){
            //Executes if there is only one word in " " - to avoid errors.
            if(n == 2) {
                char *new_arg = args[index];
                new_arg++; //Removes the first ".
                new_arg[strlen(new_arg)-1] = '\0'; //Removes the second ".
                printf("%s\n",new_arg); //Prints the string.
                return 1;
            }
            //Removes the " from first argument and prints it.
            char *new_arg = args[index];
            new_arg++, index++;
            printf("%s ",new_arg);
            //Prints the rest of the string until the last token.
            while (args[index][strlen(args[index])-1] != '\"') {
                printf("%s ", args[index]);
                index++;
            }
            //Removes the " from last argument and prints it.
            new_arg = args[index];
            new_arg[strlen(new_arg)-1] = '\0';
            printf("%s\n",new_arg);
```

If quotation marks are not found at the start and end of the arguments to be printed, the code prints each argument and checks whether one of the arguments is an environment variable using the 'set_var_value()' function. If it is an environment variable, its value will be displayed. Elsewise, the code will resume printing out each argument until the end is reached.

```
} else {
            //Prints all text after print - variable sensitive.
            do {
                //If the start of a token has $ check if it is a possible variable
and replace it with the variable value.
                if (args[index][0] == '$') {
                    args[index] = set_var_value(args[index]);
                }
                printf("%s ", args[index]);
                index++;
            } while (args[index] != NULL);
            printf("\n");
        }
    }
    return 1;
}
```

The 'chdir_comm()' function

This function changes the directory as specified by the user. If no arguments are inputted after "chdir", the function will display an error message. The function works by using the 'chdir()' function to change the path. A message is displayed if function changes the directory successfully. Elsewise, an error message is displayed.

```
//The 'chdir' internal command - Changes directories.
int chdir_comm(char **args){
    //Executes if no arguments were inputted after 'chdir'.
    if (args[1] == NULL){
        fprintf(stderr,"Error -- No arguments inputted after the command
\'chdir\'.\n");
    } else {
        //Changes the directory using the 'chdir()' function.
        if (chdir(args[1]) == 0){
            printf("Directory has been changed successfully.\n");
            set_cwd(); //Updates the environment variable 'CWD'.
        } else {
            perror("Error - chdir()");
        }
    }
    return 1;
}
```

The 'all_comm()' function

This function displays all the eggshell's variables as well as their values. It works by accessing each environment variable in the global variables array and displaying its details.

```
//The 'all' internal command - Displays all variables and their values.
int all_comm(char **args){
    //Displays the environment variable and it's value.
    for(int i=0;i<VAR_SIZE;i++){
        printf("%s=%s\n", variables[i].name, variables[i].value);
    }
    return 1;
}
```

The ‘source_comm()’ function

This function opens a text file then reads and executes each line. If no arguments are inputted after “source”, the function will display an error message. Next, a file is opened for reading and an error message is displayed if there is trouble opening the file. Subsequently, each line is scanned from the file and tokenized into an array of arguments using the ‘split_line()’ function. The array of arguments is then executed by passing it through the ‘execute()’ function. When every line in the file has been read and executed, the file is closed and the function returns.

```
//The 'source' internal command - Opens a text file, reads it and uses the input to
execute commands.
int source_comm(char **args){
    //Executes if no arguments were inputted after 'source'.
    if (args[1] == NULL){
        fprintf(stderr,"Error -- No arguments inputted after the command
\'source\'.\n");
    } else {
        FILE *f;
        //Displays error if there are problems opening the file.
        if((f = fopen(args[1], "r")) == NULL){
            perror("Error - fopen()");
        }
        //Scans each line of text file, parsing it and executing command.
        char line[MAX_SIZE];
        while(fgets(line,sizeof(line), f)){
            //Splits line.
            args = split_line(line);
            //Executes command.
            execute(args);
        }
        //Closing the file.
        fclose(f);
    }
    return 1;
}
```

External Commands

The ‘launch()’ function

This function executes external commands by searching for a matching program binary to launch as a separate process. At the start of the function, a process is forked. An chain of ‘if’ statements determine whether the process is a parent, child or had an error forking by checking the process ID number. If the process had an error forking, an error is displayed.

```
//Executes external commands, searching for a program and launching a process.
int launch(char **args){
    int status;
    //Creating a process.
    pid_t pid = fork();
    if (pid == -1) {
        perror("Error - fork()");
```

If the process ID number has a value of 0, it is a child process. The ‘signal()’ function is used in order to trap the ‘CTRL-C’ signal if sent to terminate the process. An array of environment variables is created where each variable is obtained individually using the ‘return_env_var()’ function. The array is then to be passed onto the new process. The ‘execvp’ call is then used to replace the child with a

new program. It is important to note that ‘execvp’ was used in the code below as issues were encountered when trying to use ‘execvpe’ as discussed in the ‘Bugs and Issues’ section in the documentation. It is essential to note the ‘execle’ system call was used temporarily to test whether the environment variables were passed through successfully and functioned correctly.

```
} else if (pid == 0) { //If PID is the child process.
    //Signal handling for processes.
    if(signal(SIGINT, signals) == SIG_ERR)
        perror("Error - signal()");
    //Creates an array of environment variables to be sent to the process.
    char *env[] = {return_env_var("TERMINAL"),return_env_var("CWD"),NULL};
    //Launches the process.
    //execvpe(args[0],args,env) - does not work.
    if (execvp(args[0], args) < 0) {
        perror("Error - execvp()");
    }
```

If the process is a parent, it will wait until the child process stops and the ‘EXIT’ environment variable is updated.

```
} else { //If PID is the parent process.
        //Waits for the child process and returns exit code if waitpid() is
successful.
            if(waitpid(pid, &status, WUNTRACED) == -1)
                perror("Error - waitpid()");
            else
                set_exitcode(status); //Sets the exitcode environment variable.
    }
    return 1;
}
```

Shell Variables

In the implementation of the eggshell, the shell variables have been stored in an array of structs called ‘variables’ where each struct is considered an environment variable with a name variable and value variable. The struct and array of structs are defined in the header file. An integer called ‘VAR_SIZE’ is used to keep track of the number of environment variables in the eggshell. In the documentation below, all the functions involving the shell variables will be described.

The ‘define_var()’ function

This function is used to define the shell’s environment variables. Certain values are obtained using the ‘getenv()’ function. The ‘SHELL’ and ‘PROMPT’ variable values are inputted using custom strings. The ‘TERMINAL’ and ‘CWD’ variables are added and set using the ‘set_terminal()’ and ‘set_cwd()’ functions respectively. It is important to note that the variables and their values are added to the environment variable array using the ‘modify_var()’ function which re-assigns a value to an existing environment variable or creates a new one.

```
//Defines environment variables for current shell.
void define_var(){
    //Adding variables to the environment variables array.
    modify_var("SHELL", "/home/student/cps1012/bin/eggshell");
    modify_var("USER", getenv("USER"));
    modify_var("PROMPT", "> ");
    modify_var("PATH",getenv("PATH"));
    modify_var("HOME",getenv("HOME"));
    set_terminal(); //Finds and sets the terminal value.
    set_cwd(); //Finds and sets the cwd value.
}
```

The ‘modify_var ()’ function

This function re-assigns given values to an environment variable or creates a new one with the parameters given. At the start of the code, an ‘if’ statement checks whether no variables were inputted yet. If this condition is true, the variable counter integer is incremented, and memory is allocated for one ‘variable’ struct (or one environment variable) in the variable array. The values given in the parameters are then copied into the structs name and value variables.

```
//Re-assigns a value to an environment variable or creates a new one.
int modify_var(char *name, char *value){
    //If no variables were inputted yet, allocate memory for one.
    if(VAR_SIZE == 0){
        VAR_SIZE++;
        //Allocating memory for one variable.
        variables = calloc((size_t)VAR_SIZE, sizeof(VARIABLE));
        //Copying the arguments into the new array element.
        strcpy(variables[VAR_SIZE-1].name, name);
        strcpy(variables[VAR_SIZE-1].value, value);
        return 1;
    }
```

If there are existing variables, the function checks whether the given parameter name is one of the variables in the environment variables array. If this is true, the value of the found variable is replaced with the value in the parameter.

```
//If the variable exists, replace it's contents.
for(int i=0;i<VAR_SIZE;i++) {
    //If the current variable has the same name as the argument.
    if (strcmp(name, variables[i].name) == 0) {
        //Replaces the variable value.
        strcpy(variables[i].value, value);
        return 1;
    }
}
```

If none of the above conditions are met, a new variable is created with by incrementing the variable size counter and allocating memory for one more environment variable in the eggshells variables array. The contents in the functions parameters are then copied onto the new environment variable struct.

```
//Else, creates a new variable.
    VAR_SIZE++;
    //Allocating memory for a new variable.
    variables = realloc(variables,(VAR_SIZE)*sizeof(VARIABLE));
    //Copying the arguments into the new array element.
    strcpy(variables[VAR_SIZE-1].name, name);
    strcpy(variables[VAR_SIZE-1].value, value);
    return 1;
}
```

The ‘is_var_assignment ()’ function

This function checks whether the current argument is a variable assignment. It works by checking if there is an ‘=’ in the argument. If this condition is false a value of 0 is returned. If this condition is met, the argument is passed through the ‘set_var_value()’ function to replace any environment variables with their values if found. Next, the argument is tokenized using ‘=’ as a delimiter. If more than two tokens are extracted, the function returns a value of 0 indicating that the argument is not a variable assignment.

```
//Checks if there is an variable assignment - and modifies the environment variable
accordingly.
int is_var_assignment(char *arg){
    //If there is an '=' in the string.
    if(strstr(arg, "=") != NULL){
        //Replace environment variables with their values if found.
        arg = set_var_value(arg);
        char *token;
        //Allocates memory for two tokens.
        char **tokens = malloc(2 * sizeof(char *));
        int index = 0;
        //Extracts the first token.
        token = strtok(arg, "=");
        while(token != NULL) {
            tokens[index] = token;
            index++;
            //If there are more than two tokens, it is not a variable assignment.
            if(index > 2){
                return 0;
            }
            //Extracts the next tokens.
            token = strtok(NULL, "=");
        }
```

Elsewise, if two tokens are extracted successfully, the first token is used as the variable name and the second token is used as the value. These tokens are passed through the 'modify_var()' function which will either reassign an environment variable value or create a new environment variable.

```
//Call function to add a new environment variable or replace its value.
        modify_var(tokens[0],tokens[1]);
        return 1;
    }
    return 0; //If it is not a variable assignment.
}
```

The 'return_var_value ()' function

This function returns the value of an environment variable from the eggshell's variable array. It functions by accessing each environment variable in the array and checking whether the current variables name matches the parameters given name. If this condition is true, the value of the found variable is returned. This function is used to display the user prompt.

```
//Returns the value of a variable.
char *return_var_value(char *name){
    //Accessing each environment variable.
    for(int i=0;i<VAR_SIZE;i++){
        //If the current environment variable has the same name, return its value.
        if(strcmp(name,variables[i].name) == 0) {
            return variables[i].value;
        }
    }
    return 0;
}
```

The 'set_var_value ()' function

This function replaces the occurrences of variables in a given argument with its value and returns the modified string. This is used in the 'print' command to display the value of the variable instead of the name itself. It functions by checking if one of the variable names from the variable array is present in the argument along with a dollar sign '$'. If this is true, the location of the variable in the string is replaced with its value and the modified string is returned. Otherwise, if the previous conditions aren't satisfied, the old string is returned.

```
//Replaces the $VAR with the value of the variable and returns argument.
char *set_var_value(char *arg){
    char *start;
    static char new_arg[MAX_SIZE];
    //Loops through all environment variable.
    for(int i=0;i<VAR_SIZE;i++) {
        //Adds the $ to the current variable name.
        char temp[MAX_SIZE+1] = "$";
        strcat(temp,variables[i].name);
        //If the '$VAR' is found in argument, replace it with the value.
        if (!(start = strstr(arg, temp))) {
            continue;
        } else {
            //Replaces the location of the $VAR with it's value.
            strncpy(new_arg, arg, start-arg);
            new_arg[start-arg] = '\0';
            sprintf(new_arg+(start-arg), "%s%s", variables[i].value,
start+strlen(temp));
            return new_arg; //Returns modified string.
        }
```

```
    }
    return arg; //Returns old string.
}
```

The ‘set_exitcode ()’ function

This function updates the exit code according to the inputted exit code. It does this by converting the integer ‘status’ (which is the exit code) to a string and updating the variable ‘EXITCODE’ using the ‘modify_var()’ function. It is important to note that if the ‘EXITCODE’ variable did not previously exist, the ‘modify_var()’ will create it in the variable array. This function is used to update the exit code after a process terminates.

```
//Update exitcode variable based on input 'status'.
void set_exitcode(int status){
    char exitcode[MAX_SIZE];
    sprintf(exitcode,"%d",status);
    modify_var("EXITCODE",exitcode);
}
```

The ‘set_cwd ()’ function

This function updates the current directory by using the ‘getcwd’ function to obtain the current directory and using the result to update the variable ‘CWD’ using the ‘modify_var()’ function. It is important to note that if the ‘CWD’ variable did not previously exist, the ‘modify_var()’ will create it in the variable array. This function is used after the ‘chdir_comm()’ function is successful.

```
//Finds the cwd variable and updates it's variable.
void set_cwd(){
    char cwd[MAX_SIZE];
    getcwd(cwd, sizeof(cwd));
    modify_var("CWD",cwd);
}
```

The ‘set_terminal ()’ function

This function finds the value of the ‘TERMINAL’ variable using the ‘ttyname()’ function. It then uses the result to update the current ‘TERMINAL’ variable using the ‘modify_var()’ function. It is important to note that if the ‘TERMINAL’ variable did not previously exist, the ‘modify_var()’ will create it in the variable array. This function is used in the ‘define_var()’ function when the basic environment variables are defined.

```
//Finds the terminal variable and updates it's variable.
void set_terminal(){
    char *terminal;
    terminal = ttyname(STDOUT_FILENO);
    modify_var("TERMINAL",terminal);
}
```

The ‘return_env_var ()’ function

This function outputs the shell variables in an appropriate format so that it could be identified by other processes. In other words, it returns a string in the format “VARIABLE=VALUE” where “VARIABLE” is the name of the shell variable and “VALUE” is the value it is assigned. It does this by

accessing each shell variable and checking whether the parameter name is the same as the variable name. If this condition is true, it concatenates a string in the appropriate format and returns it. This function is used when creating a list of environment variables to be sent using the ‘execvpe’ or ‘execle’ functions.

```
//Sends an environment variable name and value as one string.
char *return_env_var(char *name){
    char *temp;
    //Accessing each environment variable.
    for(int i=0;i<VAR_SIZE;i++){
        //If the variable name and argument name are the same.
        if(strcmp(name,variables[i].name) == 0) {
            temp = strcat(variables[i].name,"=");
            //Returns a string in the form VAR=VALUE.
            return strcat(temp,variables[i].value);
        }
    }
    return 0;
}
```

I/O Redirection – ‘execute_redirect()’

The ‘execute_redirect()’ function handles the input and output redirection in the eggshell. The arguments of the function include the left-hand arguments of the operator (the commands), the right-hand argument of the operator (the files, or strings), and an integer called ‘choice’ representing the type of operator that was found. Note that the left-hand arguments are stored in an array of strings called ‘left’ and the right hand arguments are stored in an array called ‘right’. The value of ‘choice’ was assigned in the ‘is_redirect()’ function and will be used to determine the type of file that will be need to be opened.

The function starts by checking whether the arguments are input or output redirection based on the ‘choice’ variable. Next, a process is forked and an error message is displayed if an error occurs.

```
int execute_redirect(char **left, char **right, int choice){
    int status;
    int out_redirection = 0, in_redirection = 0;
    //Determines whether the arguments is an output redirection.
    if(choice == 1 || choice == 2) {
        out_redirection = 1;
    }
    //Determines whether the arguments is an input redirection.
    if (choice == 3 || choice == 4){
        in_redirection = 1;
    }
    //Creating a process.
    pid_t pid = fork();
    if(pid == -1) {
        perror("Error - fork()");
```

If the process ID number indicates that the process is a child, the function will execute a series of commands depending on whether it is dealing with input redirection or output redirection. If it is output redirection, an appropriate file will open based on the found operator. The file is opened with the name given in the ‘right’ variable. If the operator was ‘>’, a file is opened for writing. If the operator was ‘>>’, a file is opened for appending. Next, the standard output is set to the file and the file is closed. The commands from the ‘left’ variable are then executed, and the results will be displayed in the file and the process is exited.

```
} else if (pid == 0) {
    //For output redirection.
    if (out_redirection == 1) {
        FILE *f;
        //Opening an appropriate file depending on whether operator is '>' or '>>'.
        if (choice == 1) {
            if ((f = fopen(right[0], "w")) == NULL)
                perror("Error - fopen()");
        }
        if (choice == 2){
            if ((f = fopen(right[0], "a")) == NULL)
                perror("Error - fopen()");
        }
        //Setting the stdout to the file.
        dup2(fileno(f), STDOUT_FILENO);
        //Closing the file.
        fclose(f);
        //Executing the arguments.
        execute(left);
        exit(1);
    }
```

If it is input redirection, an appropriate file will be open based on the found operator. The file is opened with the name given in the ‘right’ variable. If the operator is ‘<’, a file is opened for reading. If the operator is ‘<<<’, the entire ‘right variable’ is treated as a string and store in a temporary file. This is done by copying each of the ‘right’ arguments into the file and then rewinding it so that it returns to the start of the file. It must be noted

that no quotation marks or apostrophes need to be included at the start or end of the string for this to function. Next, the standard input is set to the file and file is then closed. The arguments from the 'left' variable are executed taking the contents of the chosen file as input for the 'left' arguments. The process is then exited.

```
//For input redirection.
if (in_redirection == 1){
    FILE *f;
    //Opening an appropriate file depending on whether operator is '<' or '<<<'.
    if(choice == 3){
        if ((f = fopen(right[0], "r")) == NULL)
            perror("Error - fopen()");
    }
    if(choice == 4){
        int index = 0;
        //Opening a temporary file ('HERE file') for the input.
        if((f = tmpfile()) == NULL)
            perror("Error - fopen()");
        //Copies the right hand arguments into the file.
        while(right[index]!= NULL){
            fputs(right[index], f);
            fputs(" ", f);
            index++;
        }
        fputs("\n", f);
        rewind(f); //Returns to the start of the file.
    }
    //Setting stdin to the file.
    dup2(fileno(f),STDIN_FILENO);
    //Closing the file.
    fclose(f);
    //Executing the arguments.
    execute(left);
    exit(1);
}
```

If the process ID number indicates that the process is a parent, the process will wait for the child process to terminate through the 'waitpid()' function and updates the exit code.

```
//Parent process.
    } else {
        //Waits for the child process and returns exit code if waitpid() is
successful.
        if(waitpid(pid, &status, WUNTRACED) == -1)
            perror("Error - waitpid()");
        else
            set_exitcode(status); //Sets the exitcode environment variable.
    }
    return 1;
}
```

Piping – ‘execute_pipe()’

The ‘execute_pipe()’ function handles the piping in the eggshell. The arguments of the function include the left-hand arguments of the pipe operator (the ‘left’ variable) and the right-hand argument of the pipe operator (the ‘right’ variable). The function starts by creating a pipe using the ‘pipe()’ function and displaying an error message if the pipe failed. Next, the first process is forked. An error message is displaying if the forking failed.

```
//Executes the pipe commands.
int execute_pipe(char **left, char **right){
    int mypipe[2];
    pid_t pid1, pid2;
    //Creating the pipe.
    if(pipe(mypipe) < 0){
        perror("Error -- pipe()");
    }
    //Creating a process.
    pid1 = fork();
    //If the fork failed.
    if(pid1 == -1){
        perror("Error -- fork()");
```

If the process ID number is a child, the standard output is set to the writing end of the pipe and the input side of the pipe is closed. The arguments in the ‘left’ variable are then executed and the process is terminated.

```
} else if (pid1 == 0) {
    //Executes the left hand side of the commands.
    //Sets stdout to the output side of the pipe (the writing end).
    dup2(mypipe[1], STDOUT_FILENO);
    //Closes input side of the pipe.
    close(mypipe[0]);
    //Executes the arguments.
    execute(left);
    exit(0);
```

If the process ID number is a parent, a second process is forked and error message is displayed if the forking fails. If the second process ID number is a child, the standard input is set to the reading end of the pipe and the output side of the pipe is closed. The arguments in the ‘right’ variable are then executed and the process is terminated.

```
} else {
        //Creating another process.
        pid2 = fork();
        if(pid2 == -1){
            perror("Error - fork()");
        } else if (pid2 == 0) {
            //Executes the right hand side of the commands.
            //Sets stdin to the input side of the pipe (the reading end).
            dup2(mypipe[0], STDIN_FILENO);
            //Closes output side of the pipe.
            close(mypipe[1]);
            //Executes the arguments.
            execute(right);
            exit(0);
```

If the second process ID number is a parent, the input and output side of the pipe is closed and the ‘wait()’ function is used to wait for the child processes to finish.

```
} else {
            //Closes input side of the pipe.
            close(mypipe[0]);
            //Closes output side of the pipe.
            close(mypipe[1]);
            //Waits for processes to finish.
            wait(NULL);
            wait(NULL);
        }
    }
    return 1;
}
```

Process Management

This section has not been completed. I have tried to implement the signal trapping by creating a function that executes a certain command based on the signal acquired. The 'CTRL-C' signal was the only one that was implemented, so that the process would terminate through the exit function when the signal is caught. The ability of the eggshell to run and control background processes has not been implemented.

```
//Signal Handling function.
void signals (int signal){
    switch(signal) {
        //If CTR-C is caught, terminate process.
        case SIGINT:
            printf("Signal %d caught.\n", signal);
            exit(0); //Exits process.
        //If CRT-Z is caught.
        case SIGSTOP:
            printf("Signal %d caught.\n", signal);
            break;
        default:
            printf("Error - caught wrong signal.\n");
    }
}
```

Header File

This file includes the definitions of all the functions, libraries and global variables. The code below demonstrates all the library and function definitions.

```c
#ifndef OS_THING_HEADER_H
#define OS_THING_HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <memory.h>
#define _GNU_SOURCE
#include <unistd.h>
#include <stdbool.h>
#include <errno.h>
#include <fcntl.h>

#define DELIMITERS " \t\r\n"
#define MAX_SIZE 1024

/* --------- FUNCTION DEFINITIONS ------- */

/* Core Functions */
void start();
char *read_line();
char **split_line(char *line);
int execute (char **args);
int execute_redirect(char **left, char **right, int choice);
int execute_pipe(char **left, char **right);

/* Other Functions */
int is_redirect(char **args, char **red1, char **red2);
int is_pipe(char **args, char **pipe1, char **pipe2);
int split_args(char **args, char **first, char **second, char *c);
int get_size_args(char **args);

/* Functions for Variables */
int modify_var(char *name, char *value);
int is_var_assignment(char *arg);
char *return_var_value(char *name);
char *return_env_var(char *name);
char *set_var_value(char *arg);
//Setting Environment Variables
void define_var();
void set_terminal();
void set_exitcode(int status);
void set_cwd();

/* Functions for Commands */
//Internal Commands.
int exit_comm(char **args);
int print_comm(char **args);
int chdir_comm(char **args);
int all_comm(char **args);
int source_comm(char **args);
//External Commands.
int launch (char **args);

/* Functions for Process Management */
//Signalling Functions
void signals (int signal);
```

The code below demonstrates all the global variable definitions.

```
/* ---------- GLOBAL VARIABLES ---------- */

/* Definitions for Variables */
typedef struct variable {
    char name[MAX_SIZE];
    char value[MAX_SIZE];
} VARIABLE;

VARIABLE *variables;
int VAR_SIZE = 0; //Number of environment variables.

/* Definitions for Commands */
//An array of pointers to command functions.
int (*commands[]) (char **) =
{&exit_comm,&print_comm,&chdir_comm,&all_comm,&source_comm};
//An array of commands names.
char *commands_names[] = {"exit","print","chdir","all","source"};

#endif //OS_THING_HEADER_H
```