# Operating Systems - Homework 1

- Simple Shell (SiSH) -

ChangYoon Lee

Dankook University, Korea Republic
32183641@gmail.com

**Abstract.** Shell is a small program that allows users to interact with the operating system directly. The user can command (give orders) to run the program through the shell. To operate the following command, the shell takes the input string from the user and makes it run (Mobile-Os-Dku-Cis-Mse). When the specified program completes its execution, the shell takes another input to run another program. This paper contains the concepts, methods, implementation, and results of our shell, Simple Shell (SiSH). The SiSH is a basic implementation of a shell written in C. We will look over the basics of how the shell works: read, parse, fork, execute, and wait. Then, we will check the implementations with functions of the SiSH.

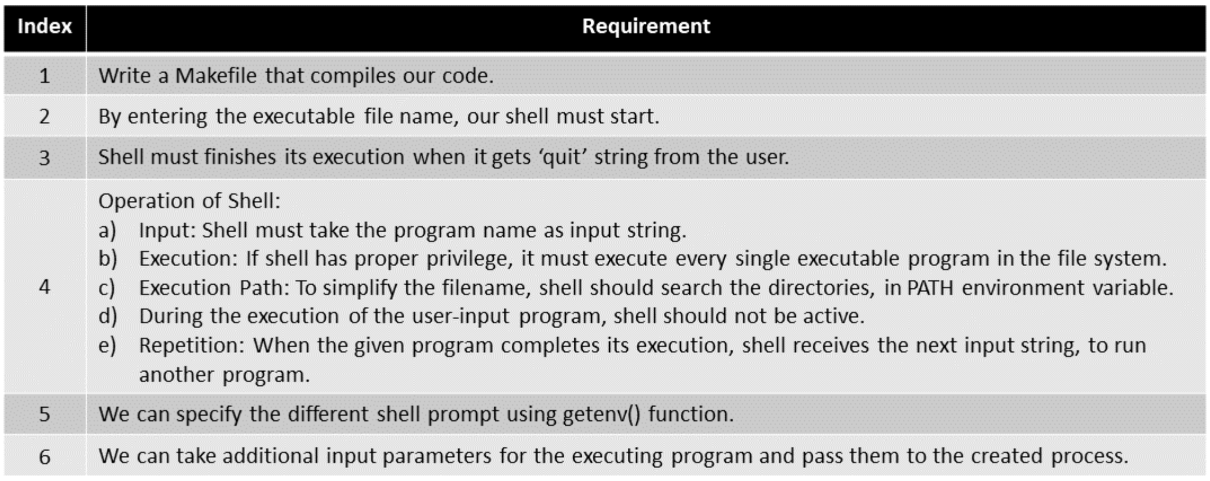
**Keywords:** Shell, C, Command Line Interface (CLI), Command Line Interpreter

## 1 Introduction

**Figure 1 - Bash Shell Logo**

Most operating systems, including LINUX, UNIX, and Windows, contain the command interpreter as a program that runs when a process is initiated or when a user first logs in (on interactive systems). In a system with multiple command interpreters to choose from, interpreters are known as shells (Silberschatz et al., 2014). The main function of the shell, which is also known as the command interpreter, is to get and execute the user-specified commands. Many of the commands given at this level manipulate files: create, delete, list, print, copy and execute, etc. In this paper, we will discuss the concepts and methods that are used in the shell, which is the command interpreter. Also, we will present our implementation and the results of the execution of the shell, which is the simple shell (SiSH).

## 2 Requirements



| Index | Requirement |
|---|---|
| 1 | Write a Makefile that compiles our code. |
| 2 | By entering the executable file name, our shell must start. |
| 3 | Shell must finishes its execution when it gets ‘quit’ string from the user. |
| 4 | Operation of Shell:<br>a) Input: Shell must take the program name as input string.<br>b) Execution: If shell has proper privilege, it must execute every single executable program in the file system.<br>c) Execution Path: To simplify the filename, shell should search the directories, in PATH environment variable.<br>d) During the execution of the user-input program, shell should not be active.<br>e) Repetition: When the given program completes its execution, shell receives the next input string, to run another program. |
| 5 | We can specify the different shell prompt using getenv() function. |
| 6 | We can take additional input parameters for the executing program and pass them to the created process. |

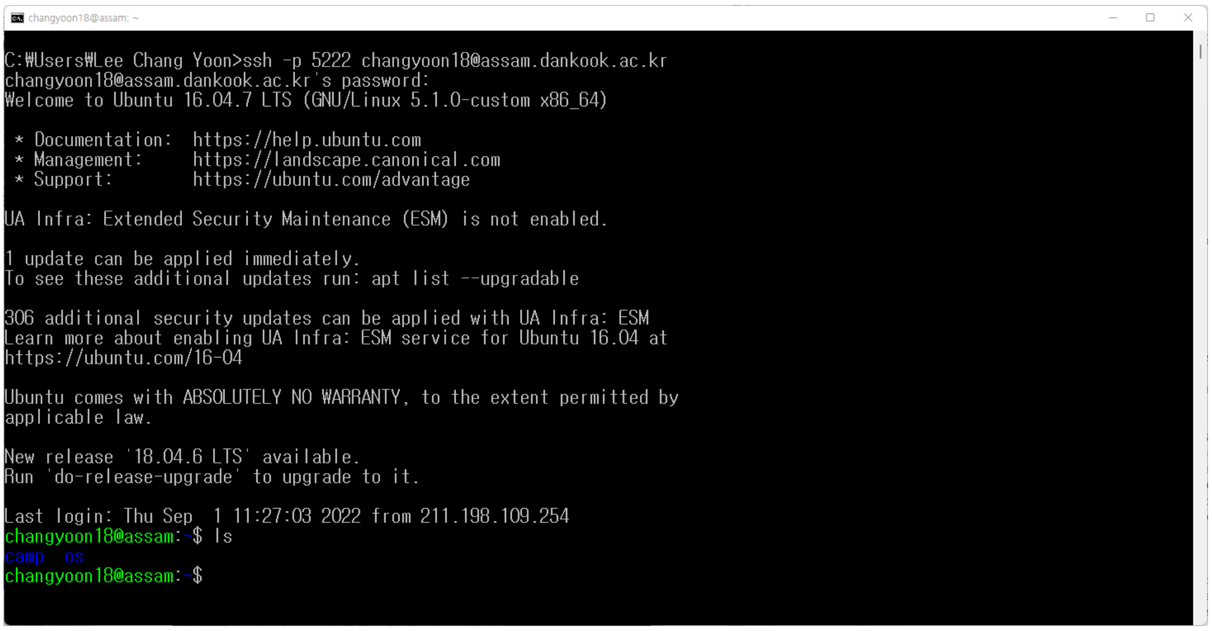
**Figure 2 - Requirement Specification**

Figure 2 shows the requirements for a simple shell. The implementations for these requirements will be described in detail afterwards.

## 3 Concepts

In this section, we will discuss the concepts of the shell, which is the command interpreter implemented in operating systems, such as LINUX, UNIX, and Windows.

### 3.1 What is Shell



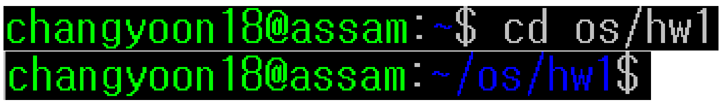
```
changyoon18@assam: ~

C:₩Users₩Lee Chang Yoon>ssh -p 5222 changyoon18@assam.dankook.ac.kr
changyoon18@assam.dankook.ac.kr's password:
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 5.1.0-custom x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

UA Infra: Extended Security Maintenance (ESM) is not enabled.

1 update can be applied immediately.
To see these additional updates run: apt list --upgradable

306 additional security updates can be applied with UA Infra: ESM
Learn more about enabling UA Infra: ESM service for Ubuntu 16.04 at
https://ubuntu.com/16-04

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

New release '18.04.6 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Thu Sep  1 11:27:03 2022 from 211.198.109.254
changyoon18@assam:~$ ls
camp  os
changyoon18@assam:~$
```

**Figure 3 - The Bash Shell Command Interpreter in Linux (Assam Server)**

As mentioned previously in the introduction, the shell is the command interpreter that runs when the process is initiated or when a user first logs in to the interactive systems. Figure 3 shows the shell program on the Assam server, which is the main server of our department. The main function of the shell is to get the input commands from the user and execute it. These commands contain the operation of file manipulations, such as create, delete, copy, execute, etc. The following commands, which are used in the shell, can be implemented in two general ways.

First, the command interpreter itself can contain the code to execute the command. To delete a file, for example, the shell may jump to a section of its code that sets up the parameters and makes the appropriate system call. Considering that each command requires its own implementation code, the size of the shell is determined by the number of commands that can be given.

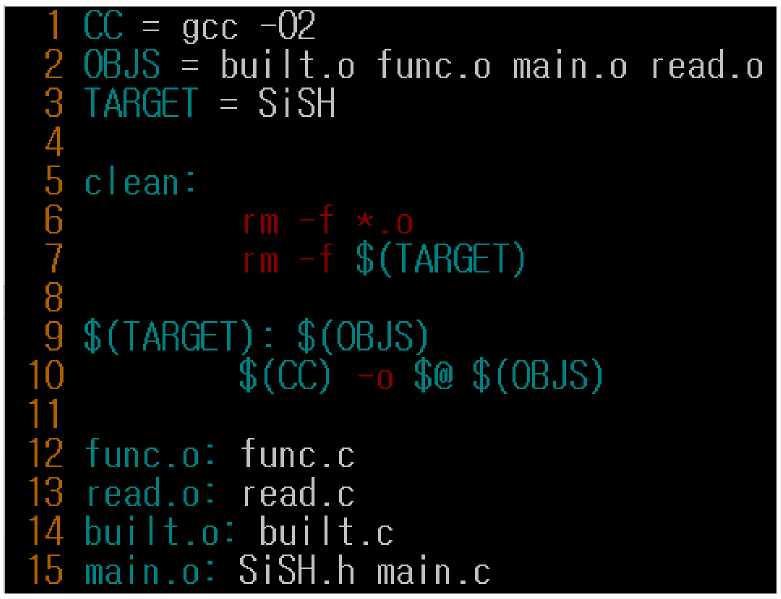
Next, in UNIX, most commands are implemented through system programs. In this case, the shell does not understand the command in any way. The command merely identifies the file to be loaded into memory and executed.



```
changyoon18@assam:~$ cd os/hw1
changyoon18@assam:~/os/hw1$
```

**Figure 4 - Execution and Result of the command 'cd'**

Figure 4 shows the execution and result of the command ‘cd’, which changes direction to the following parameter. The logic associated with the ‘cd’ command would be defined completely by the code in the file ‘cd’. In this way, a programmer can add new commands to the system easily by creating new files with the proper program logic. The command interpreter program, as known as the shell program, does not have to be changed for new commands to be added.



```
CC = gcc -O2
OBJS = built.o func.o main.o read.o
TARGET = SiSH

clean:
        rm -f *.o
        rm -f $(TARGET)

$(TARGET): $(OBJS)
        $(CC) -o $@ $(OBJS)

func.o: func.c
read.o: read.c
built.o: built.c
main.o: SiSH.h main.c
```

**Figure 5 - Make File of Simple Shell (SiSH)**

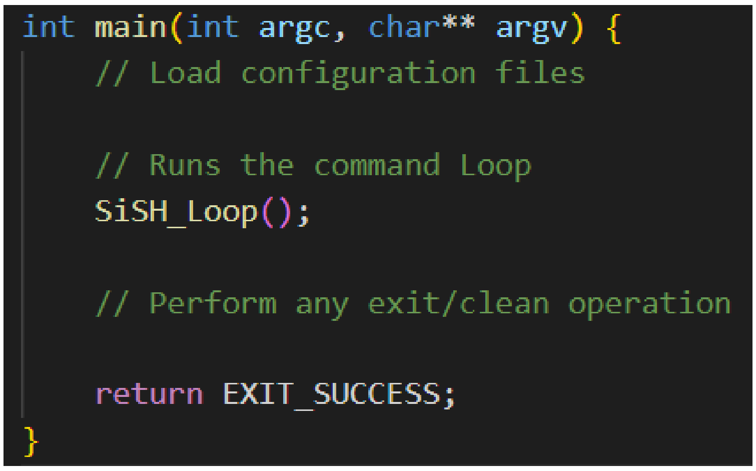
This shell, which is the command line interface (CLI), usually makes repetitive tasks easier, in part the reason that they have their programmability. For instance, when you frequently perform a task that requires a set of command line steps, you can record those steps into a file, and run the file like a program. Figure 5 shows the set of commands to build the executable file for the Simple Shell program. In the case of the Make file, we can simply build the executable file by just inputting the command of 'make ${program}'. These shell scripts, such as UNIX and LINUX, are very common on systems that are command-line oriented.

## 3.2 Basic Lifetime of a Shell

Shell does three main operations in its lifetime: Initialize, Interpret, and Terminate.

- Interpret: In this step, a typical shell would read and execute its configuration files. These changes the aspects of the shell's behavior.
- Interpret: Following step, the shell peruses commands from standard in (which may be intelligently, or a record) and executes them.
- Terminate: As soon as the shell has executed its commands, it executes any shutdown commands, frees up any memory, and terminates.

These steps are so general that they could apply to many programs, but we are going to use them for the basics of our Simple Shell. This shell will call the looping function and then terminate. However, in terms of the architecture, it is important to keep the lifetime of the program more than just looping.



```c
int main(int argc, char** argv) {
    // Load configuration files

    // Runs the command Loop
    SiSH_Loop();

    // Perform any exit/clean operation

    return EXIT_SUCCESS;
}
```

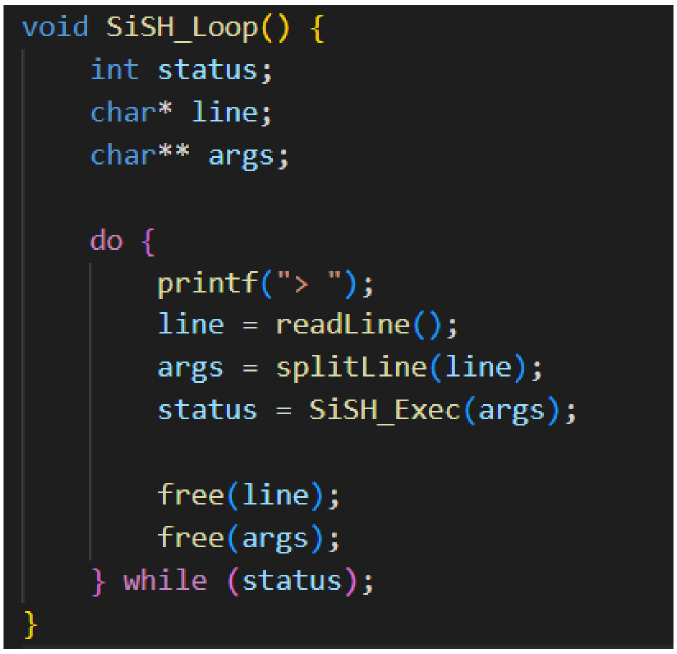
**Figure 6 - Loop of Simple Shell**

Figure 6 shows the loop of the Simple Shell. This loop will load the configuration files, interpret the input commands, and perform any other additional operations. The details of the following loop will be presented in the later sections.

### 3.3 Basic Loop of a Shell

Now, let’s get into the function that are used in the Simple Shell loop. Following steps are the simple way to handle the command during the loop of the shell: Read, Parse, and Execute.

- Read: Read the input command from the standard input.
- Parse: Separate the command string into the program and arguments.
- Execute: Run the parsed command.



```c
void SiSH_Loop() {
    int status;
    char* line;
    char** args;

    do {
        printf("> ");
        line = readLine();
        args = splitLine(line);
        status = SiSH_Exec(args);

        free(line);
        free(args);
    } while (status);
}
```

**Figure 7 - Loop Function of Simple Shell**

Figure 7 shows the structure of the loop function that is used in the loop of the shell. In the do-while loop, it first reads the line of the command. Second, it parses the input command and returns it into the array of the string. Third, it executes the parsed command and returns the status of the execution. At the end, it frees the string variables. The do-while loop continues its operation while the status is normal.

The details of the functions for read, parse, and execution of the program will be discussed in the later sections.

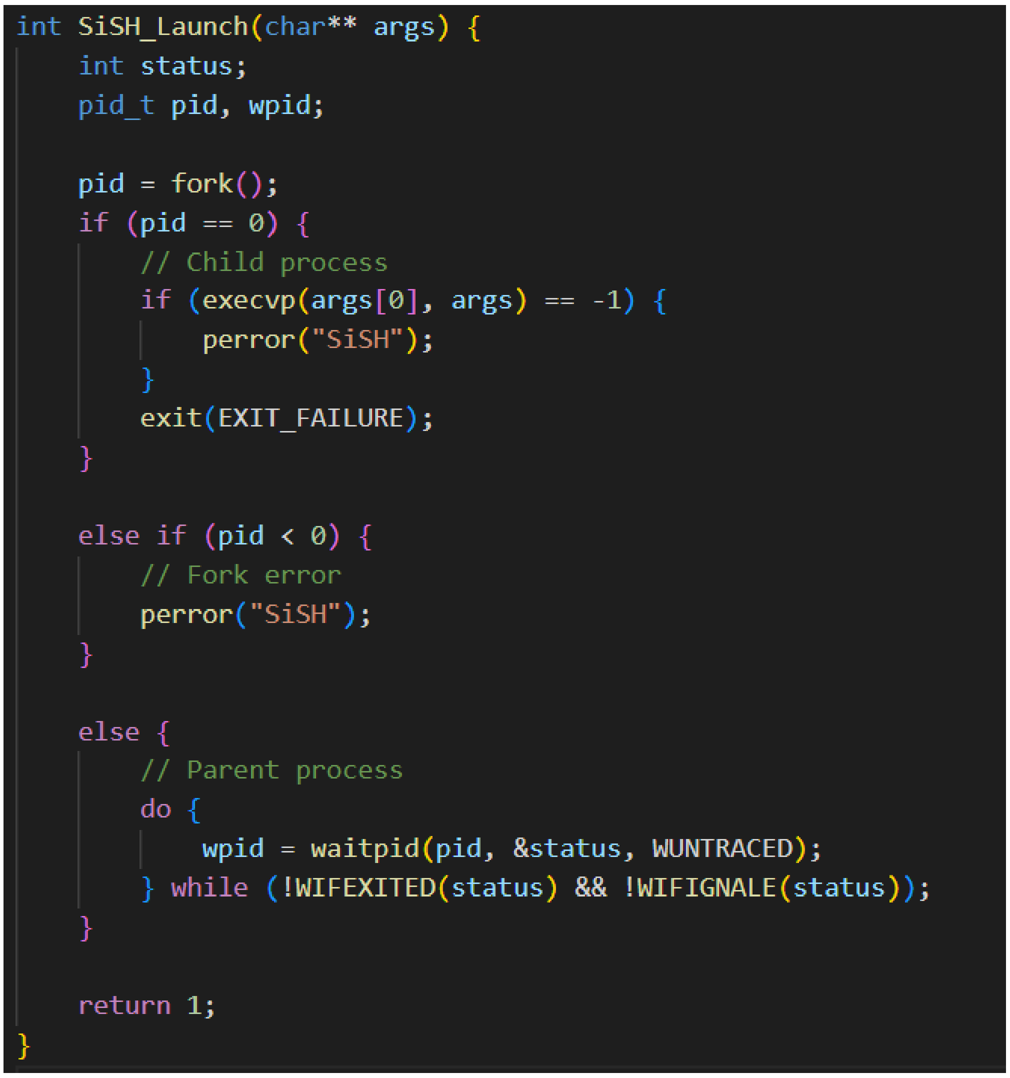
### 3.4 How Shell Start the Processes

Starting processes is the main function of the shell. Therefore, writing shell means that we need to know exactly what is going on with the processes and how they are executed. In the case of Simple Shell, it is implemented in the environment of UNIX-like operating systems. As a result, we will discuss how the process is created, executed, and terminated in the Simple Shell with UNIX functions. Shell usually works with low-level operating system functions such as system calls: fork, exec, and wait.

The fork is a system call that creates another user process, which is usually a copy of the caller process. Then, the original and copied processes become two different processes, returning with different values. For the parent process, which creates the child process, the fork function call returns with some number, that is the process ID (PID) of the child process. For the child process, the fork function call returns zero.

Exec is another operating system function that changes the process into another process. The following all takes the filename as an argument, which is the name of the program that the user wants to execute. The call discards the current program context and loads the specified program into the memory. Then, it begins execution from the very beginning point of the program. As a result, the parent process can run another program that it wants to execute in the child process.

The wait is a system call that stops the execution of the parent process. The parent process must wait until the child process completes its execution to gather and clean up the resources which are used by the children.



```c
int SiSH_Launch(char** args) {
    int status;
    pid_t pid, wpid;

    pid = fork();
    if (pid == 0) {
        // Child process
        if (execvp(args[0], args) == -1) {
            perror("SiSH");
        }
        exit(EXIT_FAILURE);
    }

    else if (pid < 0) {
        // Fork error
        perror("SiSH");
    }

    else {
        // Parent process
        do {
            wpid = waitpid(pid, &status, WUNTRACED);
        } while (!WIFEXITED(status) && !WIFIGNALE(status));
    }

    return 1;
}
```

**Figure 8 - Launch Function of Simple Shell**

Figure 8 shows the launch function of the Simple Shell. The following function takes the list of the string, which is the arguments of the user command. Then, it forks the process and saves its return value of it. When the fork function returns, two processes run concurrently. The child process will take the first if the condition of the process ID is 0.

In the child process, it wants to run the program which is given by the user as a command. In the following figure, it uses the execvp function, which is one of the variants of the exec system call. This variant, which is execvp, expects a program name and an array of string arguments (vector), which has the program name on the first element of the array. Also, the following variant lets the operating system search for the program in the current path instead of providing the full path of the running program.

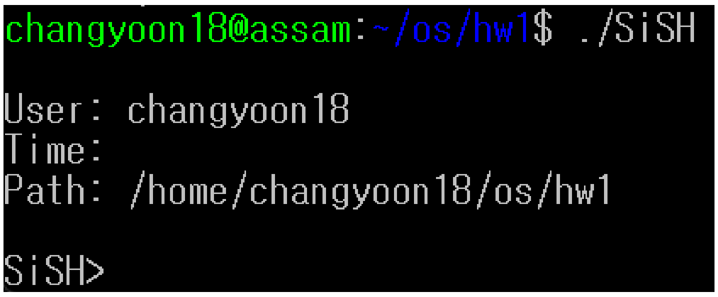
If the exec command returns -1, the following function recognizes it as an error. Therefore, by using perror function, it returns the error of the program.

The second condition checks whether the fork function has an error or not. If so, it returns the error by using the perror function.

The third condition means that the fork function has been executed successfully. The parent process will land in the following code area. Then, while the child process is going to be executed, the parent process will wait for the command to finish running. By using the system call of wait, which is waitpid function in the following function, the parent process waits for the child process to be done with its execution. Processes can change their state in lots of ways, and not all of them mean that the process has ended its execution. A process can either exit with an error, or it can be killed by a signal which is generated by a different process or system. As a result, the following function uses the macros provided with the waitpid function to wait until the child process is exited or killed. Then, if the function returns 1 as a return value, the Simple Shell prompt will wait for the user input command again.

The details of the implementation and meaning of the launch function for the Simple Shell will be discussed in the later section.

## 4 Simple Shell (SiSH)



```
changyoon18@assam:~/os/hw1$ ./SiSH

User: changyoon18
Time:
Path: /home/changyoon18/os/hw1

SiSH>
```

**Figure 9 - Simple Shell (SiSH)**

In this section, we will present the program definition for Simple Shell implementation

### 4.1 Program Definition

Before implementing the Simple Shell (SiSH), we will state the additional program definition that will be used in the real implementation.

- Global Variables

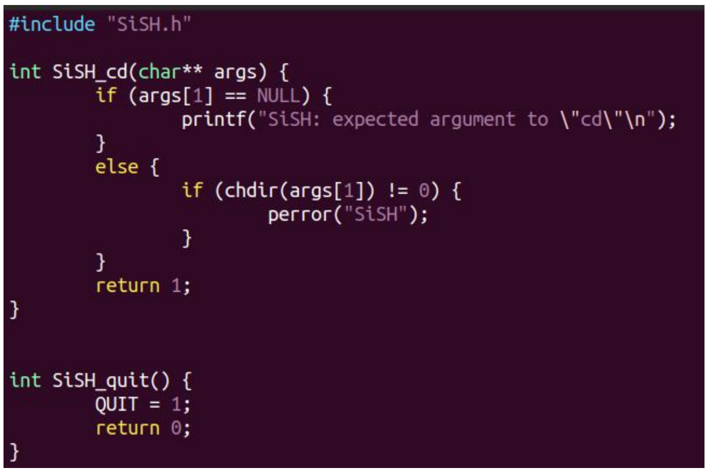
| Variables | Data Type | Definition |
|---|---|---|
| QUIT | Integer | Variable that is used to check the terminate condition |
| SHELL_NAME | String | Specifies the name of the shell |
| builtin_cmd | Array of String | Array that stores the built-in commands that needs additional parameters to execute. |

- Modules and Functions

| Modules | Functions | Definition |
|---|---|---|
| Read | readLine | Function to read a line from the command into the buffer. |
| | splitLine | Function to split a line into the consistent commands. |
| Func | numBuiltin | Function that returns the number of implemented built-in commands. |
| | SiSH_Launch | Functions to create the child process and run the commands. |
| | SiSH_Exec | Function to execute the command from the terminal. |
| | readConfig | Function that read and parse the context from the configuration file. |
| | SiSH_Interact | Function that become active when the Simple Shell is called interactively. |
| | SiSH_Script | Function that become active when the Simple Shell is called with a script as an argument. |
| Built | SiSH_cd | Function to operate the change directory command. |
| | SiSH_exit | Function to operate the quit command. |
| Main | HU_Init | Main function of the Simple Shell program. |

# 5 Implementation

### 5.1 Built In Function (built.c)



```c
#include "SiSH.h"

int SiSH_cd(char** args) {
        if (args[1] == NULL) {
                printf("SiSH: expected argument to \"cd\"\n");
        }
        else {
                if (chdir(args[1]) != 0) {
                        perror("SiSH");
                }
        }
        return 1;
}


int SiSH_quit() {
        QUIT = 1;
        return 0;
}
```

**Figure 10 - Built-in Functions**

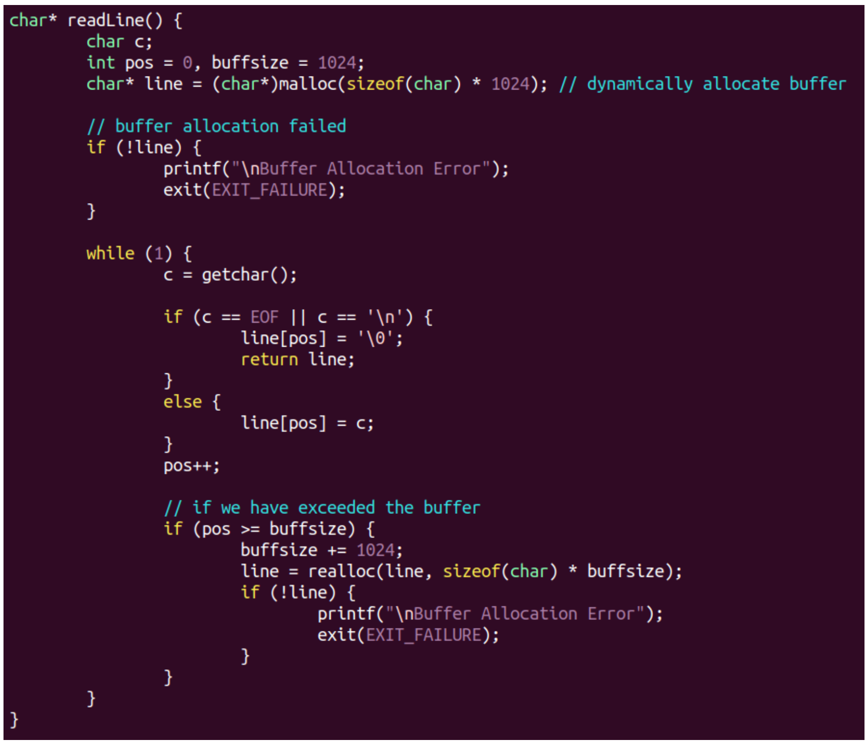
For most of the commands, a shell executes a program. However, not all of them are executing the program, and some of them are built right into the shell.

For example, if we want to change the current directory, we need to use the chdir function. The current directory of the shell is the property of the process. Therefore, if we execute a program that calls for a change directory, the shell just changes its current directory and then terminates. The current directory of the parent process in the shell will not be changed. However, the shell process itself needs to execute the chdir function, so that its current directory is updated. Then, when the shell launches the child processes, they will inherit the following directory, too.

Similarly, if there is a program called quit, it would not be able to exit the shell that is called it. The following command also needs to be built into the shell. Also, most shells are configured by running configuration scripts. These scripts use commands that change the operation of the shell.

Figure 10 shows the implemented code of the built-in functions for changing direction and quit commands. First is SiSH_cd function. This function first checks whether its second argument exists or not and prints an error message if there is no second argument. If there is a second argument, it calls the chdir function, checks for error, and returns. The second is the SiSH_quit function. The following function exits the shell.

## 5.2 Command Read and Parsing (read.c)

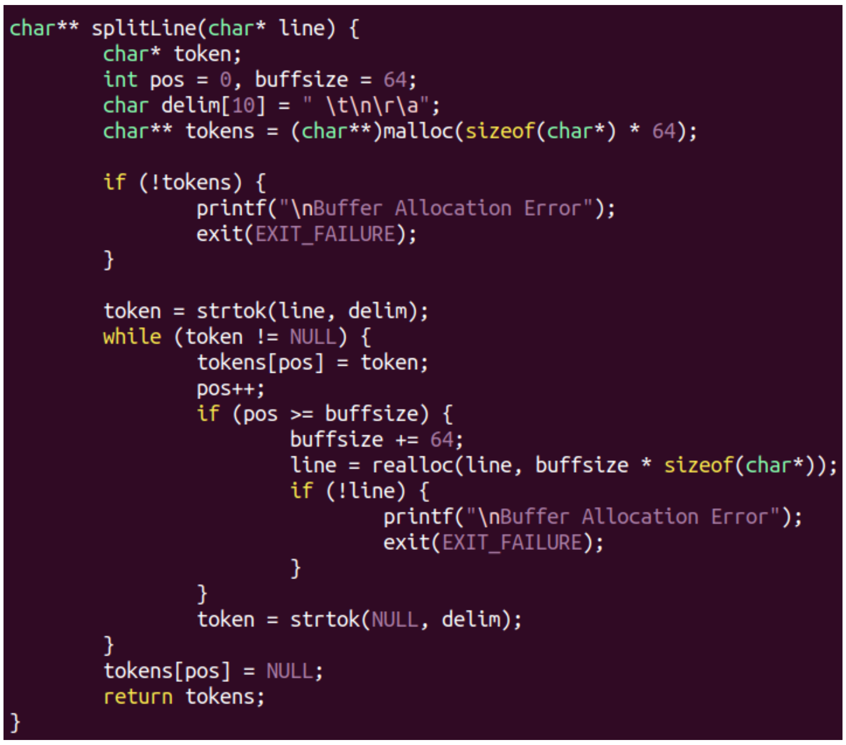


```
char* readLine() {
        char c;
        int pos = 0, buffsize = 1024;
        char* line = (char*)malloc(sizeof(char) * 1024); // dynamically allocate buffer

        // buffer allocation failed
        if (!line) {
                printf("\nBuffer Allocation Error");
                exit(EXIT_FAILURE);
        }

        while (1) {
                c = getchar();

                if (c == EOF || c == '\n') {
                        line[pos] = '\0';
                        return line;
                }
                else {
                        line[pos] = c;
                }
                pos++;

                // if we have exceeded the buffer
                if (pos >= buffsize) {
                        buffsize += 1024;
                        line = realloc(line, sizeof(char) * buffsize);
                        if (!line) {
                                printf("\nBuffer Allocation Error");
                                exit(EXIT_FAILURE);
                        }
                }
        }
}
```

**Figure 11 - Simple Shell Read Line Function**

Reading lines in C is a hassle. We do not know which command will be typed by the shell user. We cannot just allocate a block of the string to gather the input command. It can exceed the size of the block and return an error. Therefore, we must reallocate more space for the input command if it exceeds the size of the block. This concept is the main point of the readLine function.

In the while loop, the function reads a character. If it is a new line or end of file (EOF), the shell will terminate its current string and return it. Otherwise, it adds the character to the existing string. Then, it checks whether the next character will exceed the current buffer size of the string. If so, the shell reallocates the buffer before continuing.



```
char** splitLine(char* line) {
        char* token;
        int pos = 0, buffsize = 64;
        char delim[10] = " \t\n\r\a";
        char** tokens = (char**)malloc(sizeof(char*) * 64);

        if (!tokens) {
                printf("\nBuffer Allocation Error");
                exit(EXIT_FAILURE);
        }

        token = strtok(line, delim);
        while (token != NULL) {
                tokens[pos] = token;
                pos++;
                if (pos >= buffsize) {
                        buffsize += 64;
                        line = realloc(line, buffsize * sizeof(char*));
                        if (!line) {
                                printf("\nBuffer Allocation Error");
                                exit(EXIT_FAILURE);
                        }
                }
                token = strtok(NULL, delim);
        }
        tokens[pos] = NULL;
        return tokens;
}
```

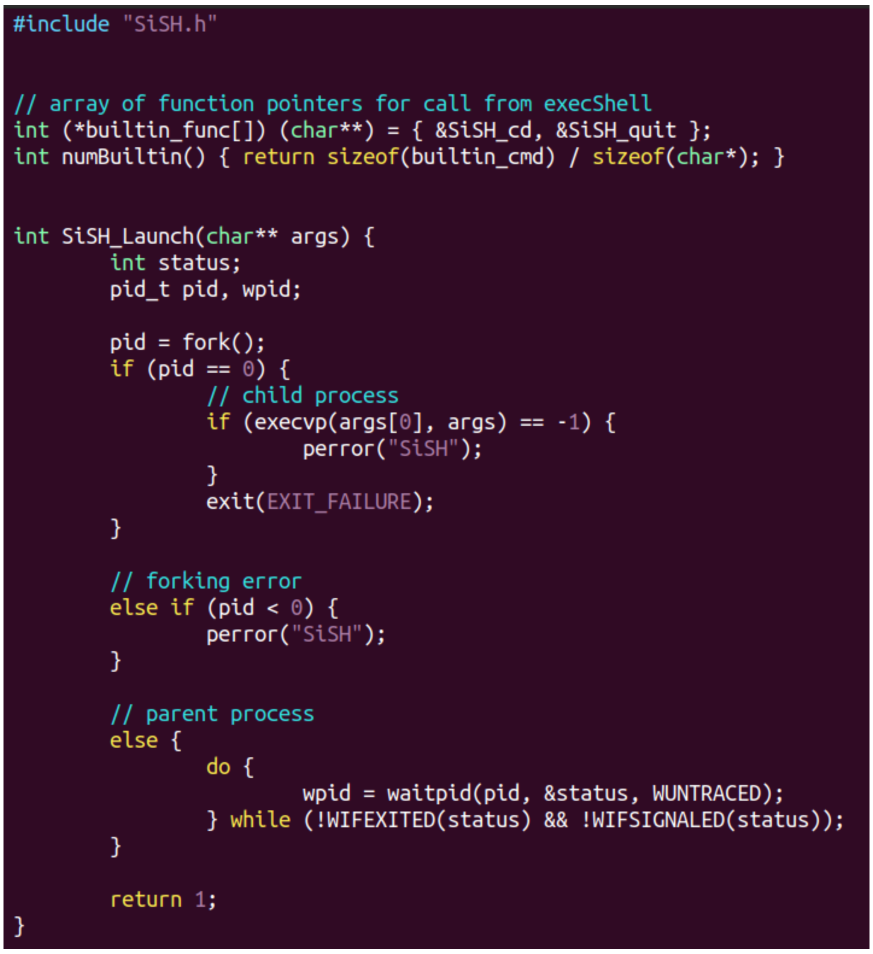
**Figure 12 - Simple Shell Split Line Function**

After we get the input command by using the readLine function, the shell must parse the following command and arguments properly. To parse the readLine command, we must determine the delimiters, which are the criterion of arguments.

In the splitLine function, the shell will not allow quoting or back-slash escaping characters in the given command or argument. Instead, the shell will use whitespace to separate the arguments from each other.

At the start of the following function, it begins tokenizing by calling the strtok function, which stands for string tokenize. This function returns a pointer to the first token. While executing while loop, the shell stores each pointer in an array of the character pointer. The shell reallocates if it is necessary to do it. The process repeats the following operation until there is no token left and returns a null pointer at the end of the operation.

## 5.3 Shell Operation (func.c)



```c
#include "SiSH.h"


// array of function pointers for call from execShell
int (*builtin_func[]) (char**) = { &SiSH_cd, &SiSH_quit };
int numBuiltin() { return sizeof(builtin_cmd) / sizeof(char*); }


int SiSH_Launch(char** args) {
        int status;
        pid_t pid, wpid;

        pid = fork();
        if (pid == 0) {
                // child process
                if (execvp(args[0], args) == -1) {
                        perror("SiSH");
                }
                exit(EXIT_FAILURE);
        }

        // forking error
        else if (pid < 0) {
                perror("SiSH");
        }

        // parent process
        else {
                do {
                        wpid = waitpid(pid, &status, WUNTRACED);
                } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        }

        return 1;
}
```

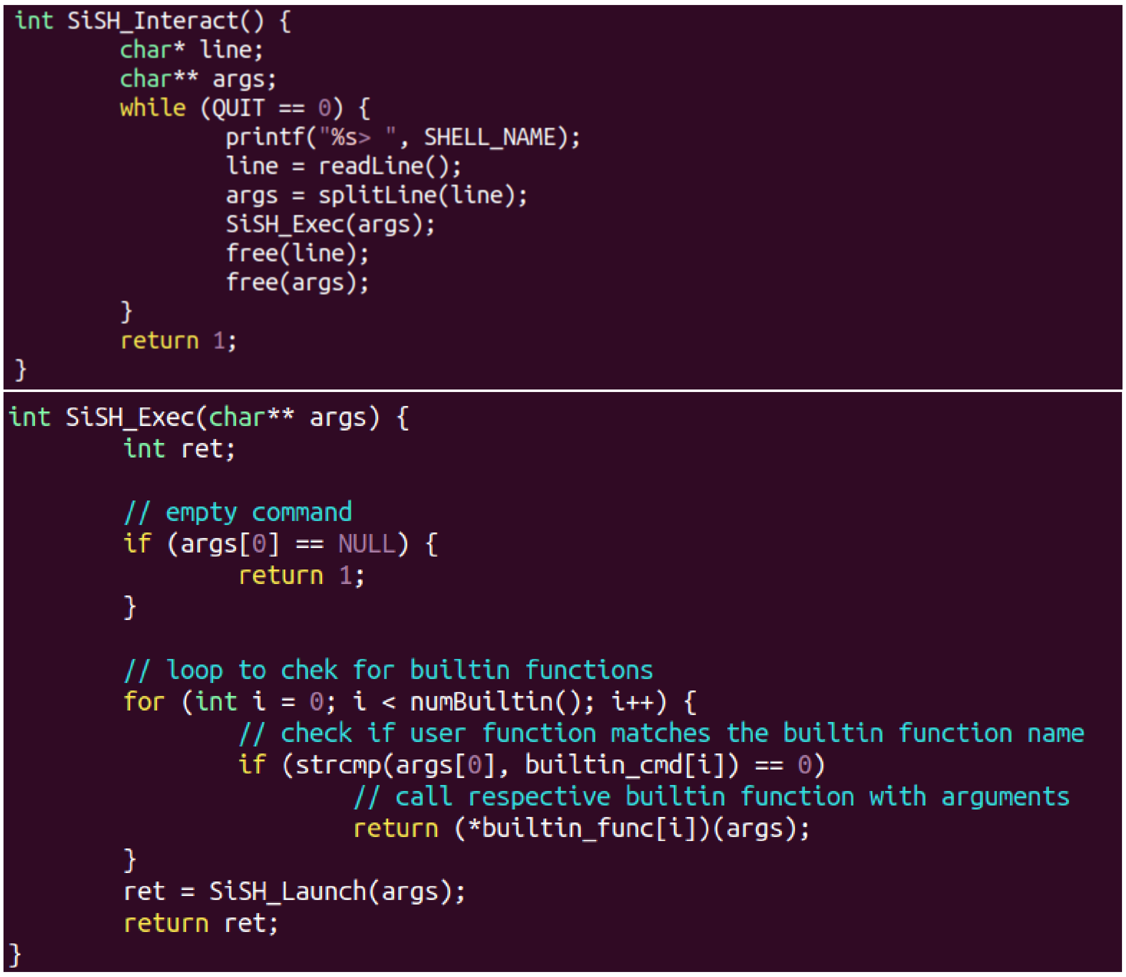
**Figure 13 - Simple Shell Launch Function**

Starting the processes is the main function of the shells. In the case of UNIX, there are only two ways of starting the processes. The first method is to execute the process in the initialization stage. When a UNIX computer boots, its kernel is loaded. Once the kernel is loaded and initialized, the kernel starts only one process, which is known as Init. The following process is run for the entire length of the time that the computer is operating and manages to load up the rest of the processes that the user needs.

Since most of the programs are not Init, there is only one practical way for processes to get started, which is known as the fork function. When the following function is called, the operating system makes duplication the process and starts them to be both running. The original one is called the parent process, and the other is called the child process. The fork function returns zero for the child process and returns process ID (PID) for the parent process.

Then, we use the exec system call. This function replaces the current running program with an entirely new one. This means that when we call the exec function, the operating system

stops the current process, loads up the new program, and starts the program in its place. The process never returns from an exec function call unless there is an error on execution.
After these two system calls, we can build the block for the new process on the UNIX system computer. First, an existing process, which is the Simple Shell, forks itself into two separate processes. Then, the child process uses the exec function to replace itself with a new program. The parent process can continue doing other operations, and it can keep tabs on the child process, by using the wait system call. Figure 13 shows the Simple Shell Launch function. The following function operates in the way of operation that is mentioned in section 3.4 of this paper.



```
int SiSH_Interact() {
        char* line;
        char** args;
        while (QUIT == 0) {
                printf("%s> ", SHELL_NAME);
                line = readLine();
                args = splitLine(line);
                SiSH_Exec(args);
                free(line);
                free(args);
        }
        return 1;
}
```

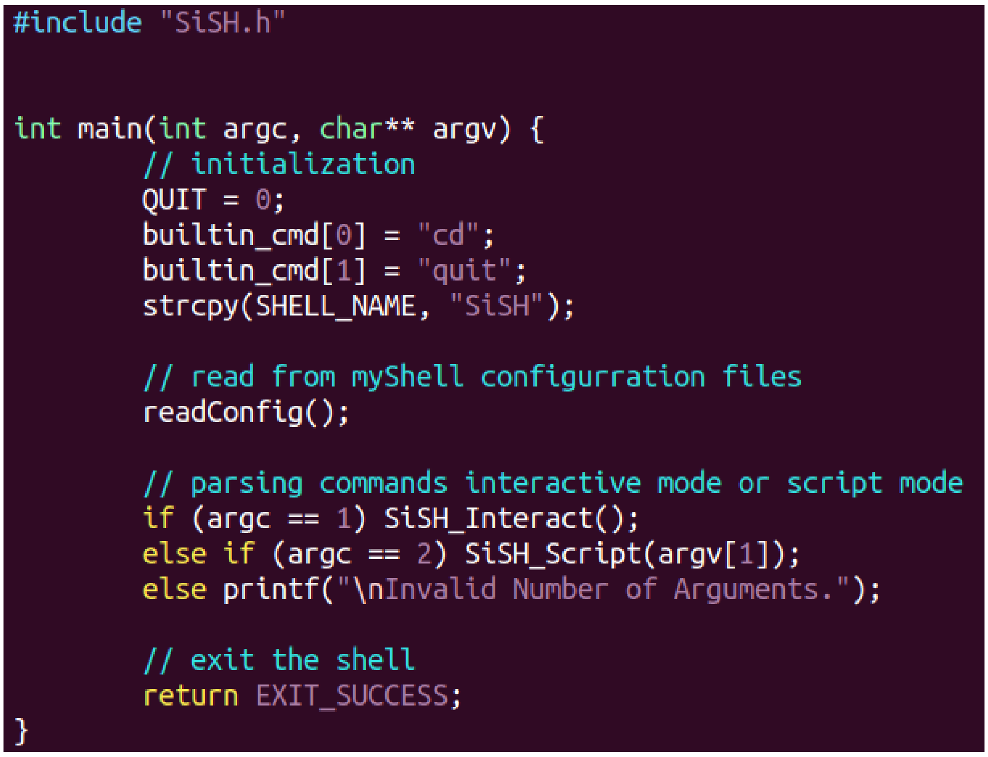
```
int SiSH_Exec(char** args) {
        int ret;

        // empty command
        if (args[0] == NULL) {
                return 1;
        }

        // loop to chek for builtin functions
        for (int i = 0; i < numBuiltin(); i++) {
                // check if user function matches the builtin function name
                if (strcmp(args[0], builtin_cmd[i]) == 0)
                        // call respective builtin function with arguments
                        return (*builtin_func[i])(args);
        }
        ret = SiSH_Launch(args);
        return ret;
}
```

**Figure 14 - Simple Shell Interact and Execution Function**

The last thing that we must do for implementing the Simple Shell is to build the function that determines whether the shell must launch a built-in function or a process.

Figure 14 shows the Simple Shell Execution Function. The following function checks if the command is equal to the one in the built-in functions, and if it is, the shell runs it. If it does not match a built-in function, it calls the Simple Shell Launch function to launch the process. One consideration is that the argument might just contain NULL if the user inputs an empty string or just white space. Then, the shell must check for that case at the beginning.

### 5.4 Main (main.c)



```
#include "SiSH.h"


int main(int argc, char** argv) {
        // initialization
        QUIT = 0;
        builtin_cmd[0] = "cd";
        builtin_cmd[1] = "quit";
        strcpy(SHELL_NAME, "SiSH");

        // read from myShell configurration files
        readConfig();

        // parsing commands interactive mode or script mode
        if (argc == 1) SiSH_Interact();
        else if (argc == 2) SiSH_Script(argv[1]);
        else printf("\nInvalid Number of Arguments.");

        // exit the shell
        return EXIT_SUCCESS;
}
```

**Figure 15 - Simple Shell Main Function**

As we mentioned in section 3.2, the basic life of the shell follows three main steps, which are Initialize, Interpret, and Terminate. Figure 16 shows the main function of the Simple Shell. If there is a single argument for executing the shell, it starts interacting with the user with the Simple Shell Interact function. Else if there are two arguments with the file name, it starts to read the configuration file that consists of the Simple Shell. Else, the program prints out the execution error.

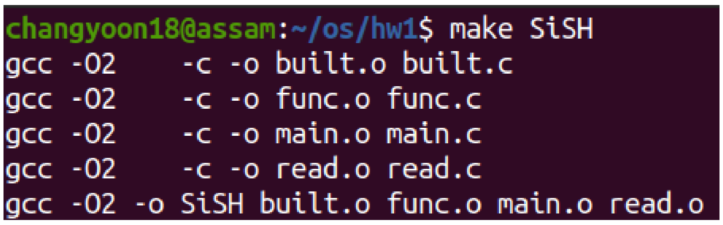
## 6 Build Environment

Following build environments are required to execute the Simple Shell.

- Build Environment:
  1. Linux Environment -> Vi editor, GCC Complier
  2. Program is built by using Makefile.

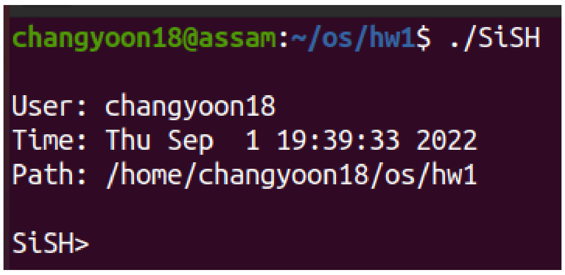
- Make Command:
  1. $make SiSH -> build the execution program of Simple Shell
  2. $make clean -> clean all of the object files that consists of the main function

## 7 Results



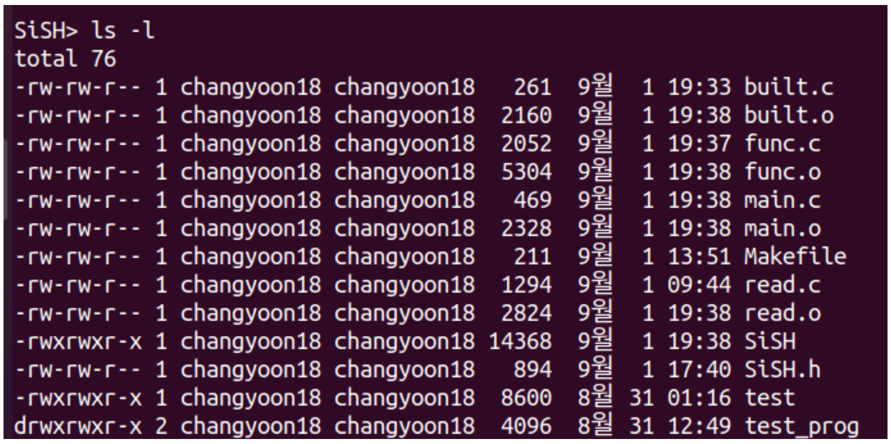
```
changyoon18@assam:~/os/hw1$ make SiSH
gcc -O2    -c -o built.o built.c
gcc -O2    -c -o func.o func.c
gcc -O2    -c -o main.o main.c
gcc -O2    -c -o read.o read.c
gcc -O2 -o SiSH built.o func.o main.o read.o
```

Figure 16 - make SiSH Command



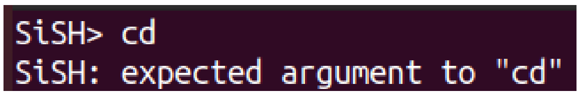
```
changyoon18@assam:~/os/hw1$ ./SiSH

User: changyoon18
Time: Thu Sep  1 19:39:33 2022
Path: /home/changyoon18/os/hw1

SiSH>
```

Figure 17 - SiSH Executions



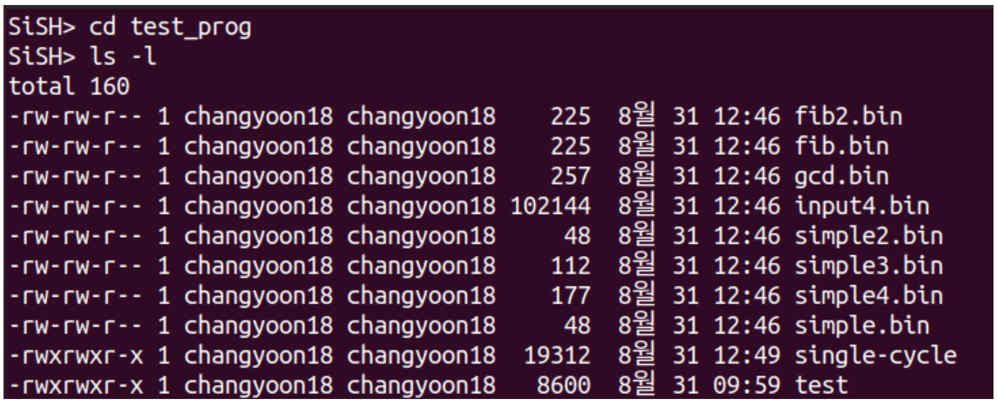
```
SiSH> ls -l
total 76
-rw-rw-r-- 1 changyoon18 changyoon18   261  9월  1 19:33 built.c
-rw-rw-r-- 1 changyoon18 changyoon18  2160  9월  1 19:38 built.o
-rw-rw-r-- 1 changyoon18 changyoon18  2052  9월  1 19:37 func.c
-rw-rw-r-- 1 changyoon18 changyoon18  5304  9월  1 19:38 func.o
-rw-rw-r-- 1 changyoon18 changyoon18   469  9월  1 19:38 main.c
-rw-rw-r-- 1 changyoon18 changyoon18  2328  9월  1 19:38 main.o
-rw-rw-r-- 1 changyoon18 changyoon18   211  9월  1 13:51 Makefile
-rw-rw-r-- 1 changyoon18 changyoon18  1294  9월  1 09:44 read.c
-rw-rw-r-- 1 changyoon18 changyoon18  2824  9월  1 19:38 read.o
-rwxrwxr-x 1 changyoon18 changyoon18 14368  9월  1 19:38 SiSH
-rw-rw-r-- 1 changyoon18 changyoon18   894  9월  1 17:40 SiSH.h
-rwxrwxr-x 1 changyoon18 changyoon18  8600  8월 31 01:16 test
drwxrwxr-x 2 changyoon18 changyoon18  4096  8월 31 12:49 test_prog
```

Figure 18 - SiSH ls -l Command



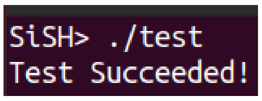
```
SiSH> cd
SiSH: expected argument to "cd"
```

Figure 19 - SiSH cd Command without any Arguments



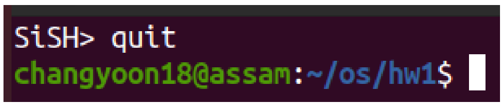
```
SiSH> cd test_prog
SiSH> ls -l
total 160
-rw-rw-r-- 1 changyoon18 changyoon18    225  8월 31 12:46 fib2.bin
-rw-rw-r-- 1 changyoon18 changyoon18    225  8월 31 12:46 fib.bin
-rw-rw-r-- 1 changyoon18 changyoon18    257  8월 31 12:46 gcd.bin
-rw-rw-r-- 1 changyoon18 changyoon18 102144  8월 31 12:46 input4.bin
-rw-rw-r-- 1 changyoon18 changyoon18     48  8월 31 12:46 simple2.bin
-rw-rw-r-- 1 changyoon18 changyoon18    112  8월 31 12:46 simple3.bin
-rw-rw-r-- 1 changyoon18 changyoon18    177  8월 31 12:46 simple4.bin
-rw-rw-r-- 1 changyoon18 changyoon18     48  8월 31 12:46 simple.bin
-rwxrwxr-x 1 changyoon18 changyoon18  19312  8월 31 12:49 single-cycle
-rwxrwxr-x 1 changyoon18 changyoon18   8600  8월 31 09:59 test
```

Figure 20 - SiSH cd Command with the Argument



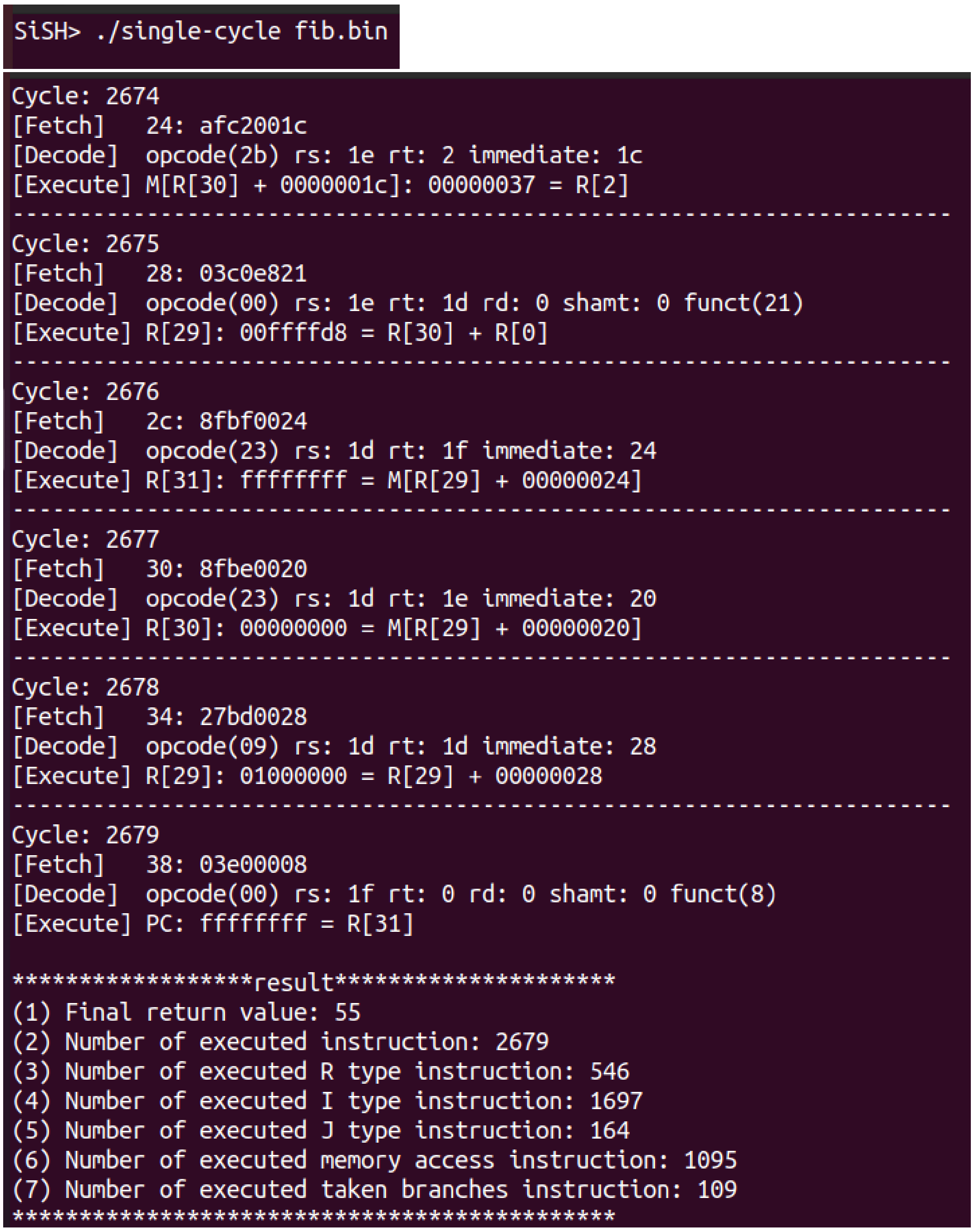
```
SiSH> ./test
Test Succeeded!
```

**Figure 21 - SiSH Program Execution Command**



```
SiSH> quit
changyoon18@assam:~/os/hw1$
```

**Figure 22 - SiSH quit Command**



```
SiSH> ./single-cycle fib.bin
```

```
Cycle: 2674
[Fetch]   24: afc2001c
[Decode]  opcode(2b) rs: 1e rt: 2 immediate: 1c
[Execute] M[R[30] + 0000001c]: 00000037 = R[2]
----------------------------------------------------------------------
Cycle: 2675
[Fetch]   28: 03c0e821
[Decode]  opcode(00) rs: 1e rt: 1d rd: 0 shamt: 0 funct(21)
[Execute] R[29]: 00ffffd8 = R[30] + R[0]
----------------------------------------------------------------------
Cycle: 2676
[Fetch]   2c: 8fbf0024
[Decode]  opcode(23) rs: 1d rt: 1f immediate: 24
[Execute] R[31]: ffffffff = M[R[29] + 00000024]
----------------------------------------------------------------------
Cycle: 2677
[Fetch]   30: 8fbe0020
[Decode]  opcode(23) rs: 1d rt: 1e immediate: 20
[Execute] R[30]: 00000000 = M[R[29] + 00000020]
----------------------------------------------------------------------
Cycle: 2678
[Fetch]   34: 27bd0028
[Decode]  opcode(09) rs: 1d rt: 1d immediate: 28
[Execute] R[29]: 01000000 = R[29] + 00000028
----------------------------------------------------------------------
Cycle: 2679
[Fetch]   38: 03e00008
[Decode]  opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[Execute] PC: ffffffff = R[31]

******************result*********************
(1) Final return value: 55
(2) Number of executed instruction: 2679
(3) Number of executed R type instruction: 546
(4) Number of executed I type instruction: 1697
(5) Number of executed J type instruction: 164
(6) Number of executed memory access instruction: 1095
(7) Number of executed taken branches instruction: 109
*********************************************
```

**Figure 23 - SiSH Program Execution Command for Single Cycle MIPS Simulator**

## 8 Conclusion

Shell is a small program that allows the user to directly interact with the operating system. By typing the commands into the shell, we can create, delete, and copy the file, or run the program. After the input commands are processed by the operating system, the shell waits for the next input command. In this paper, we discussed the Shell that we implemented, the Simple Shell. We first introduced the concepts that are mainly used in the shell program: what is the shell, the basic lifetime of the shell, and the basic loop of the shell. Then, we presented the program definition before the real implementation of the Simple Shell. According to the concepts and program definition that we previously mentioned, we explained the code of the Simple Shell, its operation, and its result. By understanding this paper, we can understand the basic operation of the shell, also known as a command line interpreter, on various operation systems, such as LINUX, UNIX, and Windows.

## Citations

[1] Mobile-Os-Dku-Cis-Mse. (n.d.). *Mobile-os-DKU-cis-MSE/MSE-os-HW1*. GitHub. Retrieved September 12, 2022, from https://github.com/mobile-os-dku-cis-mse/mse-os-hw1

[2] Silberschatz, A., Galvin, P. B., &amp; Gagne, G. (2014). 2.2.1 Command Interpreters. In Operating Systems Concepts. essay, Wiley.