
REPORT



HW#01 – Simple Shell

Freedays: 5

과목명	운영체제(MS)	담당교수	유시환 교수님
학 번	32204292	전 공	모바일시스템공학과
이 름	조민혁	제 출 일	2024/09/28

목 차

1. Introduction	1
2. Requirements	2
3. Concepts	2
3-1. Shell	2
3-2. Fork Function	4
3-3. Exec Function	4
4. Implements	5
4-1. Idea for Implementation Simple Shell	5
4-2. Specifying for Shell Implementation	6
4-3. Program Design for Shell Implementation	7
4-4. Implementation to Shell Program	7
4-4-1. Build Environments	7
4-4-2. MyShell.h	9
4-4-3. parser.cpp	10
4-4-4. shellFunc.cpp	10
4-4-5. main.cpp	15
4-4-6. config.cpp	16
5. Results	17
6. Conclusion	20

1. Introduction



Fig 1. Operating System



Fig 2. Shell Logo

초창기 컴퓨터 시스템부터 현대 컴퓨터 시스템에 이르기까지 운영체제는 컴퓨터 시스템을 작동하는데 있어서 필수적이다. 운영체제는 컴퓨터 하드웨어(HW)와 소프트웨어(SW) 사이에 위치한 소프트웨어(SW)이다. HW 와 SW 사이에 위치함으로써 컴퓨터 자원을 관리하고, 프로그램들이 효과적으로 HW 자원을 관리할 수 있게 해준다. 자세하게는 복잡한 HW 의 추상화, 공유 자원에 대한 보호된 접근, 보안과 인증, 프로그램 통신 관리 등과 같이 SW 가 HW 자원을 효율적으로 사용하게 해준다. Fig 1 과 같이 현대 운영체제는 MacOS, Linux, Windows, Android 등과 같이 여러 종류의 운영체제가 존재한다. 이렇게 수많은 운영체제들은 각각의 구별되는 특징이 있지만 공통된 특징이 존재한다. ‘셸(Shell)’은 모든 운영체제가 가지고 있는 인터페이스로, 프로그램 또는 사용자가 운영체제를 사용하고 접근하기 위한 인터페이스이다. 운영체제마다 GUI(Graphic User Interface)로 셸을 제공하거나 CLI(Command Line Interface)로 제공한다. 본 레포트에서는 ‘Simple Shell’이라는 주제로 System-Level Programming 을 할 것이다. Simple Shell 을 구현하기 위해 먼저, 요구사항을 살펴본다. 또한 Shell 을 구현하기 위한 관련 개념을 소개한다. 이후, Simple Shell 을 구현하기 위한 아이디어를 소개하고, 구현된 코드를 소개하며, Simple Shell 프로그램을 실행함으로써 결과를 소개한다. 마지막으로, Simple Shell Programming 을 하면서 느낀 점을 서술하며 마무리한다.

2. Requirements

Table 1. Requirements Table

Index	Requirements
1	Makefile & Compile
2	Start of SiSH: by entering the executable filename, my shell starts. End of SiSH: SiSH finishes execution when it gets 'quit' string from the user.
3	Operation of SiSH: 3-1. Input: takes program name as input string 3-2. Execution: it has to execute every single executable program in the filesystem, if it has proper privilege 3-3. Execution path: to simplify (contract) the filename (full path beginning with '/'), SiSH should look into directories, in PATH environment variable. PATH environment variable holds the ':'-separated string, that specifies multiple locations in the filesystem. 3-4. During the execution of the user-input program, shell should not be active. 3-5.Repetition: When the given program completes its execution, it receives the next input string, to run another program.
4	I can specify different shell prompt using getenv function. (e.g. current working directory (PWD, TIME, USER, etc.)
5	I can take additional input parameters for the executing program, and pass them to the created process.

Shell Programming 에서 요구되는 요구사항들이 Table1 에 제시 되어있다.

3. Concepts

본 장에서는 Simple Shell 을 구현하는데 있어서 필요한 관련 개념들을 소개한다.

3-1. Shell

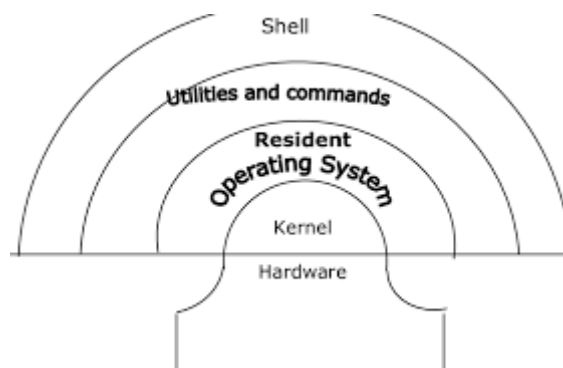


Fig 3. Shell-OS Structure

Introduction 에서도 언급했듯이 셸(Shell)은 GUI 또는 CLI 형태로 Fig3 과 같이 운영체제와 프로그램 또는 사용자 사이에 존재한다. 셸은 사용자가 명령어를 입력하면 이를 운영체제의 커널로 전달하여 명령을 실행하는 프로그램이다. 이를 통해 사용자가 시스템과 상호작용이 가능해지며, 다양한 명령어를 통해 프로그램 실행, 파일 및 디렉터리 관리가 가능해진다. 셸의 종류로는 Bourne Shell, Bash, C Shell, Korn Shell, Zsh 등이 존재하며 각기 다른 기능과 문법이 존재한다. 이 중 Bash Shell 이 리눅스 시스템에서 가장 많이 사용되는 셸이다. 따라서 본 레포트에서는 Bash Shell 의 문법을 기준으로 셸 프로그램을 구현하도록 한다.

```
minhyuk@DESKTOP-9KUSLQP:~$ pwd
/home/minhyuk
minhyuk@DESKTOP-9KUSLQP:~$ whoami
minhyuk
minhyuk@DESKTOP-9KUSLQP:~$ ls -al
total 196
drwxr-x--- 27 minhyuk minhyuk 4096 Sep 28 17:36 .
drwxr-xr-x  3 root    root    4096 May 29 00:07 ..
-rw-----  1 minhyuk minhyuk 26677 Sep 28 17:36 .bash_history
-rw-r--r--  1 minhyuk minhyuk  220 May 29 00:07 .bash_logout
-rw-r--r--  1 minhyuk minhyuk 3925 May 29 01:18 .bashrc
drwxr-xr-x  9 minhyuk minhyuk 4096 Aug 17 18:41 .cache
drwxr-x--x  4 minhyuk minhyuk 4096 Sep 26 11:36 .config
```

Fig 4. Bash Shell Example1

Fig 4 는 Bash Shell 의 화면이다. 화면과 같이 셸은 명령어를 사용자로부터 입력 받는다. 입력 받은 명령어는 파싱되어, 프로그램을 실행하거나 특정 시스템을 호출한다. 또한 명령어에 옵션을 추가하여 해당 명령어가 부가 기능을 수행하도록 할 수 있다.

```
minhyuk@DESKTOP-9KUSLQP:~$ ls -al > testMinhyuk.txt
minhyuk@DESKTOP-9KUSLQP:~$
minhyuk@DESKTOP-9KUSLQP:~$ cat testMinhyuk.txt
total 196
drwxr-x--- 27 minhyuk minhyuk 4096 Sep 28 17:40 .
drwxr-xr-x  3 root    root    4096 May 29 00:07 ..
-rw-----  1 minhyuk minhyuk 26677 Sep 28 17:36 .bash_history
-rw-r--r--  1 minhyuk minhyuk  220 May 29 00:07 .bash_logout
-rw-r--r--  1 minhyuk minhyuk 3925 May 29 01:18 .bashrc
drwxr-xr-x  9 minhyuk minhyuk 4096 Aug 17 18:41 .cache
drwxr-x--x  4 minhyuk minhyuk 4096 Sep 26 11:36 .config
drwxr-xr-x  3 minhyuk minhyuk 4096 May 29 00:25 .dotnet
-rw-----  1 minhyuk minhyuk   31 Aug 17 18:28 .gdb_history
```

Fig 5. Bash Shell Example 2

이뿐만 아니라 연속되어 나타나는 첫 번째 명령어와 두 번째 명령어를 통해 작업을 Fig 5 와 같이 수행할 수 있다. 이를 ‘Redirection’이라 하며 명령어 한 줄을 통하여 수행할 수 있는 작업의 범위를 확대시켜준다.

3-2. Fork Function

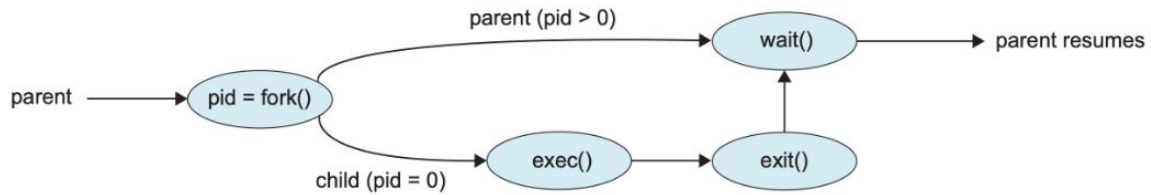


Fig 6. Fork Function

앞서 언급한 셸의 개념과 기능은 `fork()` 함수에 의해 구현된다. `fork()` 함수는 셸을 구현하는 데 있어 핵심적인 기능을 수행하는 함수이다. 이 함수는 현재 실행 중인 프로세스(부모 프로세스)를 복사하여 새로운 프로세스(자식 프로세스)를 생성하는 역할을 한다. 셸이 새로운 명령어를 실행할 때마다 새로운 프로세스를 만들어 명령을 처리하게 되는데, 이 과정에서 `fork()` 함수가 중요한 역할을 한다. Fig 6 은 `fork()` 함수의 동작 원리를 보여준다. `fork()` 함수는 2 가지의 값을 반환한다. 먼저, 부모 프로세스에게 자식 프로세스의 PID 를 반환하며, 자식 프로세스에게는 0 의 값을 반환한다. 이를 통해 `if~else` 문을 결합하여 프로세스의 흐름을 제어할 수 있다. `exec()` 함수를 자식 프로세스에서 호출하면 호출된 함수의 값이 자식 프로세스의 메모리를 덮어쓰는데, 이는 3-3 에서 자세히 서술하겠다. 부모 프로세스는 `wait()` 함수를 통해 자식 프로세스의 동작이 끝날 때까지 기다리고 있다가, 동작이 끝나면 부모 프로세스의 작업을 계속 하도록한다.

3-3. Exec Function

`exec()` 함수는 유닉스 및 리눅스 운영체제에서 새로운 프로그램을 실행하기 위한 시스템 콜 함수이다. 3-2 에서 서술한 `fork()` 함수와 보통 사용되며 `fork()` 함수를 통해 자식 프로세스가 생성되면, 자식 프로세스에 `exec()` 함수 호출을 통해 자식 프로세스의 메모리 공간을 덮어쓴다. 이를 셸을 구현하는 관점으로 바라보면 부모 프로세스는 사용자로부터 명령어를 입력 받는 프로그램을 실행하고 있고, 사용자로부터 명령어를 입력받으면 `fork()` 함수 호출을 통해 입력받은 명령어를 `exec()` 함수와 결합하여 처리가 가능해진다. `exec()` 함수는 여러 파생 함수가 존재한다. 대표적으로 `execl()`, `execvp()`, `execlp()`, `execvp()` 등이 있다. 본 레포트에서는 셸을 구현하는데 있어 `execvp()` 함수를 이용하였다. `execvp()` 함수의 프로토타입을 살펴보면 'int execvp(char * prog, char * argv[])' 이다. 실행할 프로그램을 입력 받고, 옵션을 입력을 받음으로써 명령어를 효율적으로 처리가 가능해진다. 또한 해당 함수 실행 시 환경변수 'PATH'를 참고하여 인자를 처리하기에 훨씬 간편하게 사용이 가능하다.

4. Implements

본 장에서는 Simple Shell 을 구현하기 위한 아이디어 및 설계를 소개하고, Simple Shell 의 소스 코드에 대한 설명을 서술한다.

4-1. Idea for Implementation Simple Shell

3 장에서는 관련 개념을 소개하며 Simple Shell 을 구현하는 관점으로 바라보았다. 3 장의 내용을 정리하면 다음과 같다. 셸을 사용자로부터 명령어를 입력 받는다. 입력 받은 명령어는 파싱 되어 처리된다. 파싱된 명령어를 처리하기 위해 `fork()` 함수를 호출하고, 이를 통해 자식 프로세스를 생성하며, 해당 프로세스에서 `exec()` 함수를 호출하여 명령어를 처리한다. 명령어 처리가 완료된 후, 다시 사용자로부터 명령어를 입력 받는다. 이를 나타내는 순서도가 Fig 7 에 나타나 있다.

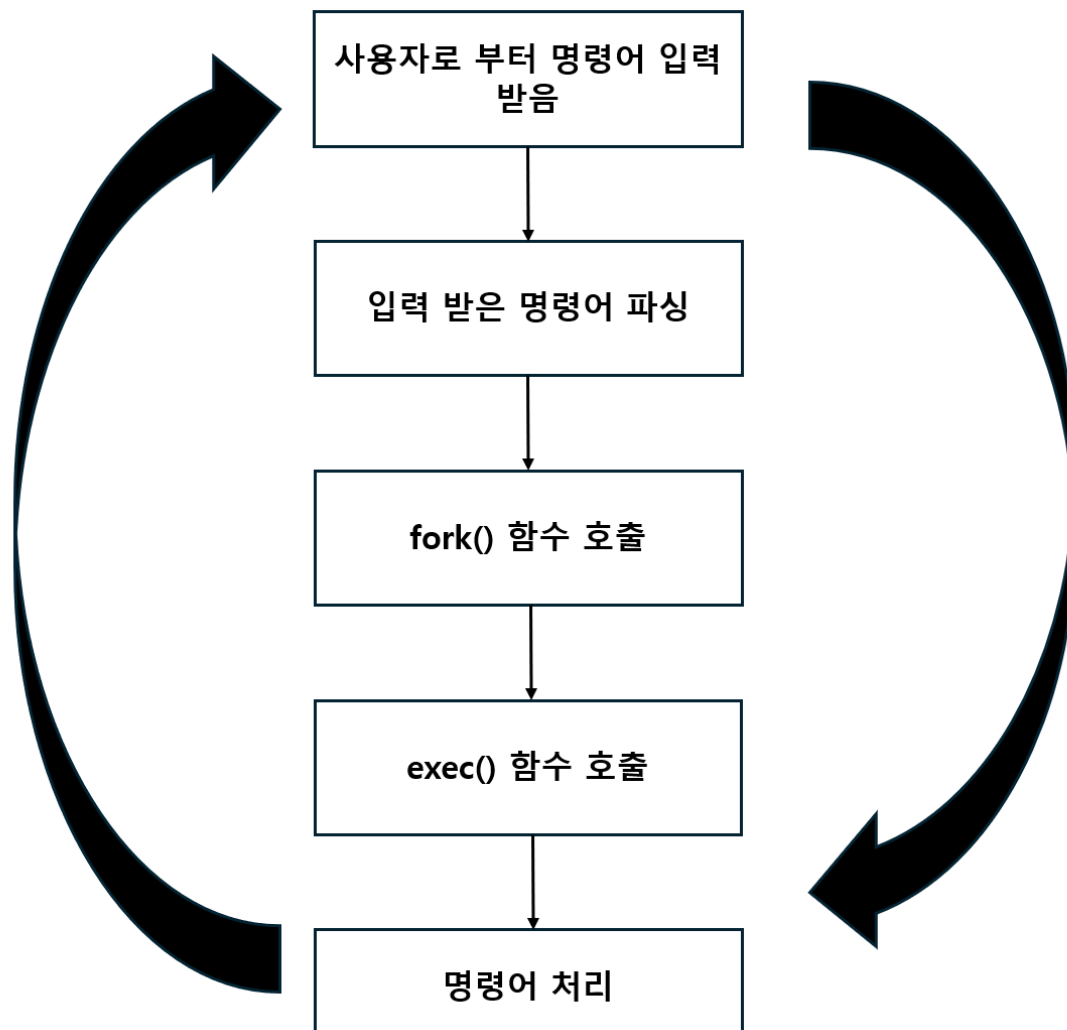


Fig 7. Shell Operation Process

Fig 7 에서 나타나는 프로세스는 fork() 함수, exec() 함수를 통해 셸을 동작 시키고 있다. 그러나 해당 함수들을 통한 셸의 동작은 'ls', 'cat', 'grep' 등과 같이 셸에서 제공되는 외부 명령어만 처리 가능하다. 따라서, 'cd', 'pwd', 'quit' 등과 같이 셸에서 제공되지 않는 명령어를 처리해야 한다. 또한, 이러한 명령어들의 조합에서 리다이렉션을 통한 명령어 처리도 필요하다. 명령어들의 종류가 증가했기에 이러한 명령어들의 종류를 구별하고 선택하는 과정도 필요하다.

4-2. Specifying for Shell Implementation

본 절에서는 4-1 절에서 제시한 아이디어 및 문제점을 바탕으로 실제적인 셸 구현을 위한 구체화를 한다. 본 레포트에서 제시하는 설계는 다음과 같다. 먼저, 셸 프로그램을 실행하면 사용자로부터 명령어를 입력 받는다. 입력 받은 명령어를 파싱을 하며, 명령어의 종류를 선택하고 해당 명령어로 분기한다. 각 명령어에 맞게 처리를 해준다. 이후 명령어를 수행해주고, 다시 사용자로부터 명령어를 입력 받도록 한다. 이를 반영한 순서도가 Fig 8 에 제시되어 있다.

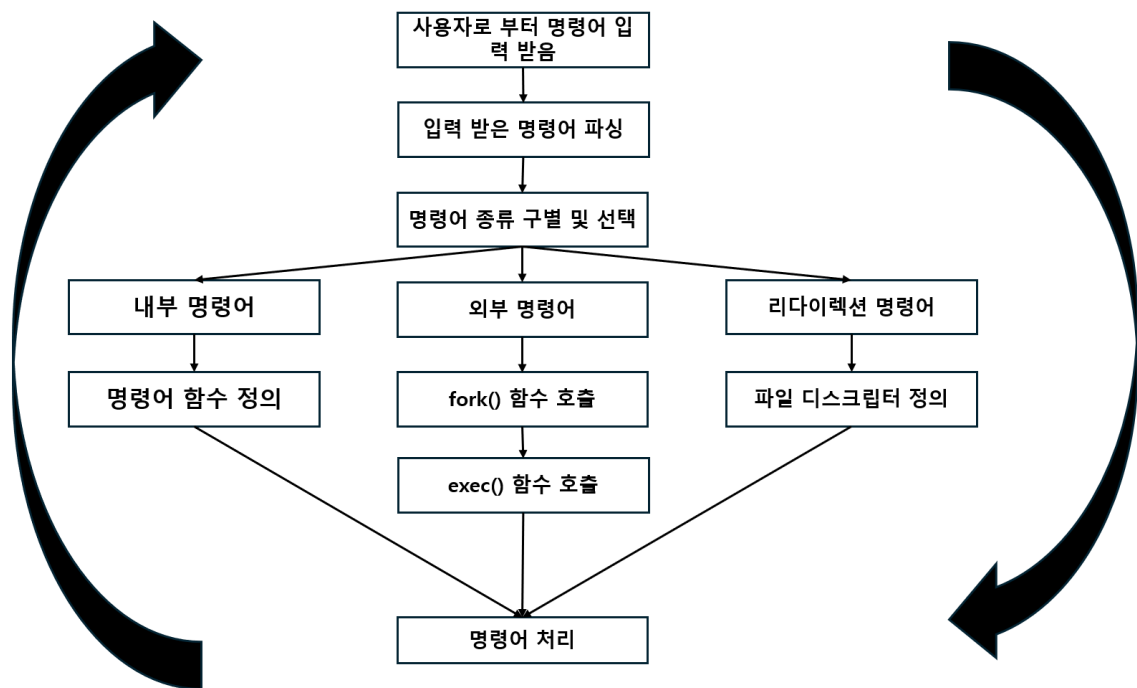


Fig 8. Detail of Shell Operation Processing

4-3. Program Design for Shell Implementation

본 절에서는 4-2 절에서 제시된 순서도를 바탕으로 코드를 설계하도록 한다. 설계된 코드는 Table 2에 나타나 있다.

Table 2. Program Design

File	Kinds of Function	Function
MyShell.h	-	헤더파일 포함 및 매크로, 전역 변수, 필요한 함수 선언
parser.cpp	vector<string> parseArg(string arg)	문자열 토큰
shellFunc.cpp	void initShell()	셸 초기화 진행.
	void selectShell(vector<string> args)	명령어 종류 선택
	void internalCmd(vector<string> args)	내부 명령어 처리
	void externalCmd(vector<string> args)	외부 명령어 처리
	void redirectCmd(vector<string> args)	리다이렉션 처리
main.cpp	int main(int argc, char * argv[])	셸 프로그램 작동
config.cpp	void configEnv()	셸 프로그램 시작 시 최초 화면 출력

Table 2에 나타나있는 파일과 함수를 살펴보면 MyShell.h에 헤더파일 포함 및 매크로, 전역 변수, 필요한 함수 선언을 통하여 하위 파일들이 해당 헤더파일을 통해 정보를 얻도록 한다. parser.cpp에서는 문자열을 토큰하는 작업을 수행하고, shellFunc.cpp에서는 셸을 초기화하고, 명령어를 처리 및 수행하는 기능을 하도록 한다. 이후, main.cpp에서는 셸 프로그램을 실질적으로 작동시키고, config.cpp에서 셸 프로그램 시작 시 최초 화면을 출력하도록 한다.

4-4. Implementation to Shell Program

본 절에서는 4-3 절에서 설계된 프로그램을 바탕으로 구현하도록 한다.

4-4-1. Build Environments

먼저 구현한 코드 설명에 앞서 해당 프로그램을 개발한 IDE, 프로그램 실행하기 위한 환경은 다음과 같다.

- ✓ 개발 환경: Ubuntu 22.04, VSCode
- ✓ 프로그래밍 언어: C++
- ✓ 프로그램 실행 방식: Makefile 이용

```
CC = g++ -O2
OBJS = config.o main.o parser.o shellFunc.o
TARGET = MyShell

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

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

config.o: config.cpp MyShell.h
    $(CC) -c config.cpp

Parser.o: parser.cpp MyShell.h
    $(CC) -c Parser.cpp

shell_func.o: shellFunc.cpp MyShell.h
    $(CC) -c shellFunc.cpp

main.o: MyShell.h main.cpp
    $(CC) -c main.cpp
```

```
G+ config.cpp
G+ main.cpp
M Makefile
C MyShell.h
G+ parser.cpp
G+ shellFunc.cpp
```

```
minhyuk@DESKTOP-9KUSLQP:~/OperatingSystem/HW#01-SimpleMyShell$ make MyShell
g++ -O2 -c config.cpp
g++ -O2 -c main.cpp
g++ -c -o parser.o parser.cpp
g++ -c -o shellFunc.o shellFunc.cpp
g++ -O2 -o MyShell config.o main.o parser.o shellFunc.o
```

Fig 9. Makefile, Files and Way to Build

4-4-2. MyShell.h

```
#pragma once

#include "bits/stdc++.h"

#include <filesystem>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

using namespace std;

/* Global Variables */
extern bool isEXIT;
extern string SHELL_NAME;
extern time_t SHELL_TIME;
extern string SHELL_USER;
extern string delim;
extern vector<string> internalCommand;

/* Functions */
void configEnv();
void initShell();
void selectShell(vector<string> args);
void internalCmd(vector<string> args);
void externalCmd(vector<string> args);
void redirectCmd(vector<string> args);
vector<string> parseArg(string arg);
```

Fig 10. MyShell.h

Fig 10 에 제시된 코드를 살펴보면 Table 2 에 제시된 것과 같이 헤더파일, 매크로, 전역 변수, 함수가 선언되어 있다. 특히, '#pragma once'를 통해 헤더파일의 중복 포함을 방지하였고, '#include "bits/stdc++"'을 통해 하나의 include 선언으로 표준 C++ 헤더파일들을 포함하고 이외 포함되지 않은 헤더파일은 따로 포함하였다. 또한, 내부 명령어를 정의를 위한 'internalCommand' 벡터를 선언하였다.

4-4-3. parser.cpp

```
#include "MyShell.h"

vector<string> parseArg(string arg){
    vector<string> args;
    size_t start = 0; size_t end = 0;

    while((end = arg.find_first_of(delim, start)) != string::npos){
        if(end != start) args.push_back(arg.substr(start, end-start));
        start = end+1;
    }

    if(start < arg.length()) args.push_back(arg.substr(start));

    return args;
}
```

Fig 11. parser.cpp

parser.cpp 에서는 명령어를 전달받아서, 사전에 정의한 구분자를 통해 parsing 하였다. 여기서 find_first_of() 함수를 통해 delim 문자열 중 하나라도 존재할 경우 parsing 을 하도록 하였다. 이후 반복문을 빠져나오면 남은 문자열을 string vector 에 push 한 후 반환하였다.

4-4-4. shellFunc.cpp

```
void initShell(){
    string cmd;
    vector<string> tokCmd;

    while(!isEXIT){
        cout<<"\033[32m\033[1m"<<SHELL_NAME<<"\033[0m"<<"\033[34m\033[1m"<<getcwd(NULL, 0)<<"\033[0m"<<"$ ";

        getline(cin, cmd);
        tokCmd = parseArg(cmd);
        selectShell(tokCmd);
    }
}
```

Fig 12. initShell()

initShell() 함수에서 입력된 명령어를 한 줄로 받기위해 getline() 함수를 사용했고, 이를 cmd에 저장한 후, parseArg() 함수를 호출하고 반환받은 값은 selectShell()로 전달하였다.

```

void selectShell(vector<string> args){
    if(args.empty()) perror("Empty Command");

    if(args.size() >= 3) redirectCmd(args);
    else{
        for(int idx = 0; idx < internalCommand.size(); ++idx){
            if(args[0] == internalCommand[idx]) {
                internalCmd(args);
                return;
            }
        }
        externalCmd(args);
    }
    return;
}

```

Fig 13. selectShell()

해당 함수에서는 문자열이 3이상이면 redirection 명령어라 판단하였고, 사전에 정의한 내부 명령어를 통해 입력받은 명령어와 일치하면 내부 명령어로 판단하고, internalCmd()를 호출하였고, 이외에는 외부 명령어로 판단하여 externalCmd() 함수를 호출하였다.

```

if(args[0] == "cd"){
    if(args.size() == 1) {
        perror("No Argument, So ChangeDir to $HOME\n");
        const char * home_dir = getenv("HOME");
        if(home_dir != NULL){
            if (chdir(home_dir) != 0) {
                perror("Failed to change directory");
            }
        }
        else{
            perror("There is no set Home Dir");
        }
    }
    else if(!args[1].empty()){
        if(chdir(args[1].c_str()) != 0) perror("No such file or directory\n");
    }
}

```

Fig 14. internalCmd()

해당 함수에서는 여러 내부 명령어에 대한 로직을 구현하였지만 대표적으로 cd(Change Directory)

에 대해 살펴보겠다. 먼저 cd인지를 판단한 후, 전달받은 인자의 길이가 1이라면 따로 경로 지정을 안했다고 판단하여 getenv() 함수를 통해 환경 변수 HOME에 등록된 경로로 변경해주었다. 만약 경로가 존재하지 않다면 에러를 발생시켰고, 이외에는 경로가 존재한다 판단하여 chdir() 함수로 입력한 경로로 변경시켜주었다.

```
void externalCmd(vector<string> args){
    pid_t pid;
    char* argv[args.size() + 1];

    for (int i = 0; i < args.size(); ++i) {
        argv[i] = const_cast<char*>(args[i].c_str());
    }
    argv[args.size()] = NULL;

    pid = fork();

    if(pid < 0) perror("Fork Error\n");
    else if(pid == 0){
        if(execvp(args[0].c_str(), argv) == -1) perror("execvp failed");
        exit(EXIT_FAILURE);
    }
    else{
        int status;
        pid_t wait_pid = waitpid(pid, &status, 0);

        if(wait_pid == -1) perror("Wait PID Failed");
    }
}
```

Fig 15. externalCmd()

해당 함수에서는 입력 받은 인자를 C Style 문자열로 변경시켜주는 전처리 작업을 해주고, fork() 함수를 호출하여 자식 프로세스를 생성한다. execvp() 함수를 호출함으로써 자식 프로세스에서 해당 명령어를 처리하도록 구현하였다. 이후 처리가 완료된 후 부모 프로세스에서의 작업을 이어가도록 하였다.

```

void redirectCmd(vector<string> args){
    int fd;
    int saved_fd;
    string redirectChar;

    for(int i = 0; i < args.size(); ++i){
        if(args[i] == ">"){
            fd = open(args[i + 1].c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if(fd < 0){
                perror("open");
                return;
            }

            redirectChar = ">";
            saved_fd = dup(STDOUT_FILENO);
            dup2(fd, STDOUT_FILENO);
            close(fd);

            args.erase(args.begin() + i, args.begin() + i + 2);
            break;
        }
    }
}

```

Fig 16. redirectCmd()'s '>' logic

해당 함수는 리다이렉션을 처리하도록 구현하였다. 먼저, '>'의 리다이렉션이 인식된 경우 File Descriptor를 선언하여 해당 파일을 표현하도록하고 dup2() 함수를 통해 해당 파일을 표준 출력으로 지정하였고, 해당 프로세스가 처리된 후 복귀하기 위해 dup() 함수를 통해 저장 해두었다. 이후 Fig 15와 같이 fork() 함수를 호출하고 execvp() 함수 호출을 통해 해당 리다이렉션 및 명령어를 처리하였다.

```

else if(args[i] == "|"){
    int pipefd[2];
    if(pipe(pipefd) == -1){
        perror("Pipe Failed");
        return;
    }

    pid_t pid = fork();

    if(pid == -1){
        perror("Fork Failed");
        return;
    }

    if(pid == 0){
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[0]); close(pipefd[1]);

        vector<string> leftCmd(args.begin(), args.begin() + i);
        char * argv[leftCmd.size() + 1];

        for(int i = 0; i < leftCmd.size(); ++i){
            argv[i] = const_cast<char*>(args[i].c_str());
        }
        argv[leftCmd.size()] = NULL;
        execvp(argv[0], argv);

        perror("execvp Failed");
        exit(EXIT_FAILURE);
    }
}

```

```

else{
    wait(NULL);

    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]); close(pipefd[1]);

    vector<string> rightCmd(args.begin() + i + 1, args.end());
    char* argv[rightCmd.size() + 1];

    for(int i = 0; i < rightCmd.size(); ++i){
        argv[i] = const_cast<char*>(rightCmd[i].c_str());
    }
    argv[rightCmd.size()] = NULL;
    execvp(argv[0], argv);

    perror("execvp failed");
    exit(EXIT_FAILURE);
}

```

Fig 17. redirectCmd()'s Pipe

'|(pipe)'의 경우 두 가지의 명령어 모두 실행이 필요했기에 File Descriptor를 2개 선언하였고, 먼저 왼쪽 명령어를 처리하도록 하였다. 왼쪽 명령어의 결과가 오른쪽 명령어로 출력되어야 하기 때문에 pipefd[1]을 표준 출력으로 지정하였고, 왼쪽 명령어를 수행하였다. 이후 오른쪽 명령어를 처리하기 위해 pipefd[0]을 표준 입력을 뒀으로써 pipe()에 저장되어있는 출력 결과를 표준 입력으로 받아 해당 명령어를 처리하였다.

4-4-5. main.cpp

```
#include "MyShell.h"

bool isEXIT;

time_t SHELL_TIME;
string SHELL_NAME;
string SHELL_USER;

string delim;
vector<string> internalCommand;

int main(int argc, char * argv[]){
    SHELL_TIME = time(NULL);
    SHELL_NAME = "minhyuk:";
    SHELL_USER = getenv("USER");

    delim = " \n\t\r\a";
    internalCommand = {"cd", "exit", "quit", "q", "pwd", "help"};

    configEnv();
    initShell();

    cout<<"Finish SiSH Shell !!!"<<endl;
    return 0;
}
```

Fig 18. main()

main() 함수에서는 전역 변수를 초기화해줌으로써 셸이 실행될 때 출력 화면을 구성하기 위한 데이터를 설정하였고, 내부 명령어를 사전에 정의하였다. 이후 셸이 종료되면 종료 메시지를 두어 해당 셸이 종료되었음을 표시하였다.

4-4-6. config.cpp

```
3-11
#include "MyShell.h"
using namespace std;

void configEnv() {
    cout<<"\033[34m\033[1m";
    cout<<"*****"<<endl;
    cout<<"*****  Welcome to MinHyuk Shell - SiSH  *****"<<endl;
    cout<<"*****"<<endl;
    cout<<"\033[0m";

    cout<<"\033[35m\033[1m";
    cout<<"Current Time: "<<ctime(&SHELL_TIME);
    cout<<"User: "<<SHELL_USER<<endl;
    cout<<"\033[0m";

    cout<<"\033[36m\033[1m";
    cout<<"-----"<<endl;
    cout<<"Type 'help' to get started!"<<endl;
    cout<<"-----"<<endl;
    cout<<"\033[0m";
    cout<<endl;
}
```

Fig 19. configEnv()

configEnv() 함수에서는 셸이 실행될 때 최초 화면을 출력하기 위한 구성을 하였다. 여러 색상을 출력문에 설정하였다.

5. Results

본 장에서는 4장에서 설명한 프로그램을 바탕으로 결과를 확인한다.

```

minhyuk@DESKTOP-9KUSLQP:~/OperatingSystem/HW#01-SimpleMyShell$ ./MyShell
*****
***** Welcome to MinHyuk Shell - SiSH *****
*****
Current Time: Sat Sep 28 20:43:37 2024
User: minhyuk
-----
Type 'help' to get started!
-----

minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$

```

Fig 20. Running Shell Program

Fig 20과 같이 셸 프로그램을 실행하면 config.cpp에서 정의한 출력문이 색상과 함께 출력된다. 이후 셸이 변경됨을 통해 Simple Shell이 구동된 것을 알 수 있다.

```

minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ help
=====
This shell provides external and internal commands, as well as redirection.
Although it's a simple shell and does not offer advanced features,
it supports essential commands. Please use the shell with commands such as "ls -al", "cd", "cat *.c | grep z" !
Thanks!!!
=====

minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ pwd
Current Working Directory: "/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell"
minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ cd ..
minhyuk:/home/minhyuk/OperatingSystem$ cd ./HW#01-SimpleMyShell
minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$

```

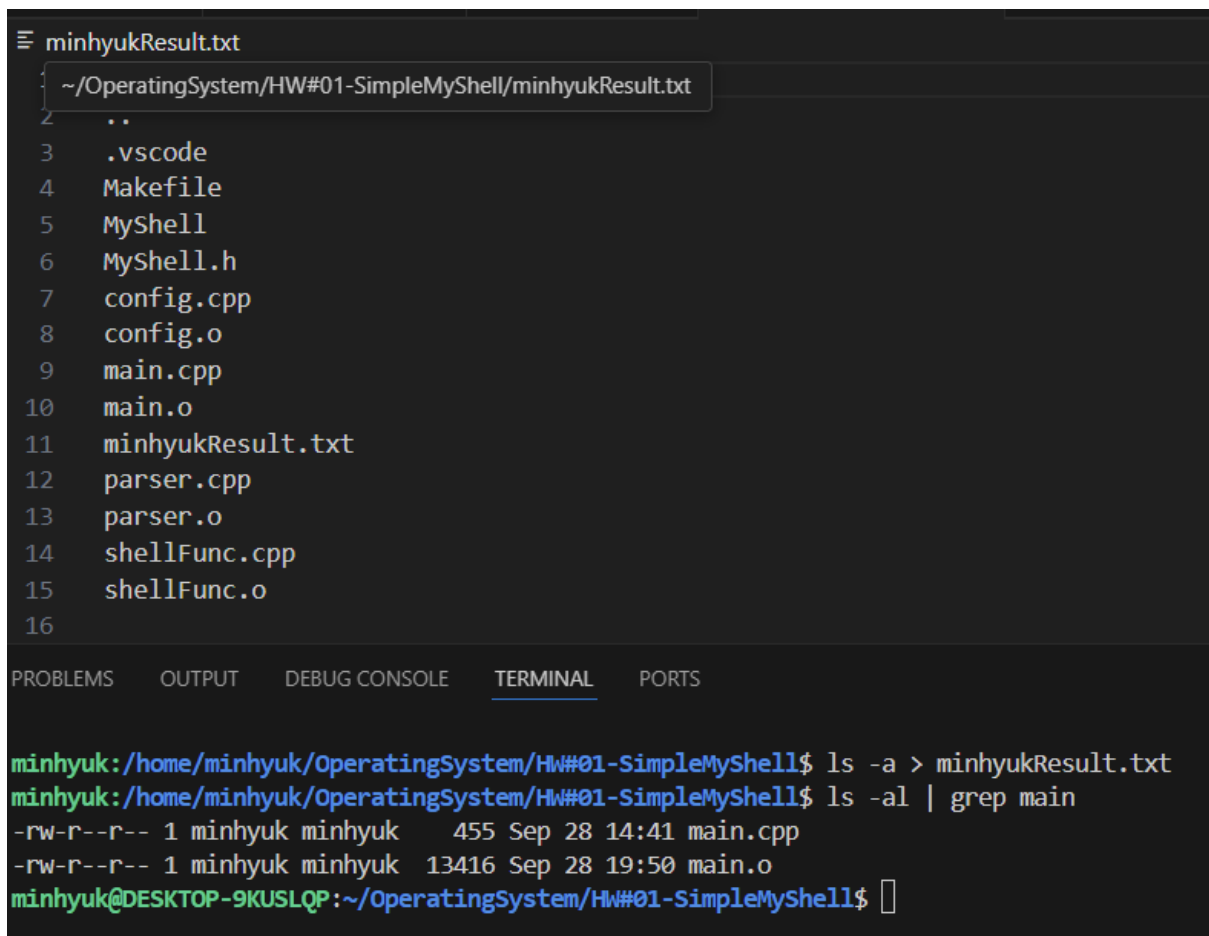
Fig 21. Internal Command Result

Fig 21과 같이 정의된 내부 명령어가 정상적으로 동작하는 것을 확인할 수 있다.

```
minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ ls -al
total 300
drwxr-xr-x 3 minhyuk minhyuk 4096 Sep 28 20:43 .
drwxr-xr-x 6 minhyuk minhyuk 4096 Sep 18 18:54 ..
drwxr-xr-x 2 minhyuk minhyuk 4096 Sep 28 14:45 .vscode
-rw-r--r-- 1 minhyuk minhyuk 373 Sep 28 14:39 Makefile
-rwxr-xr-x 1 minhyuk minhyuk 83792 Sep 28 20:43 MyShell
-rw-r--r-- 1 minhyuk minhyuk 572 Sep 28 16:23 MyShell.h
-rw-r--r-- 1 minhyuk minhyuk 674 Sep 28 16:22 config.cpp
-rw-r--r-- 1 minhyuk minhyuk 6880 Sep 28 19:50 config.o
-rw-r--r-- 1 minhyuk minhyuk 455 Sep 28 14:41 main.cpp
-rw-r--r-- 1 minhyuk minhyuk 13416 Sep 28 19:50 main.o
-rw-r--r-- 1 minhyuk minhyuk 381 Sep 28 15:16 parser.cpp
-rw-r--r-- 1 minhyuk minhyuk 41928 Sep 28 19:50 parser.o
-rw-r--r-- 1 minhyuk minhyuk 5956 Sep 28 20:32 shellFunc.cpp
-rw-r--r-- 1 minhyuk minhyuk 107528 Sep 28 20:43 shellFunc.o
minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ cat main.cpp
#include "MyShell.h"
```

Fig 22. External Command Result

Fig 22에서 외부 명령어 역시 정상적으로 작동하는 것을 알 수 있다.



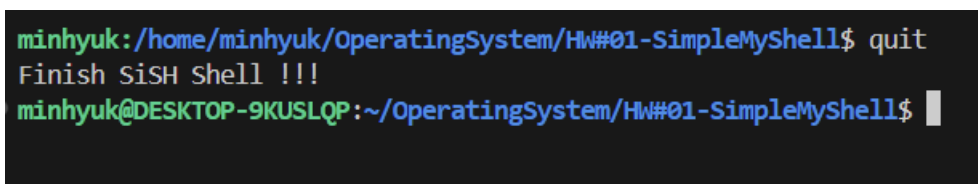
```
minhyukResult.txt
{
  ~/OperatingSystem/HW#01-SimpleMyShell/minhyukResult.txt
}
2  ..
3  .vscode
4  Makefile
5  MyShell
6  MyShell.h
7  config.cpp
8  config.o
9  main.cpp
10 main.o
11 minhyukResult.txt
12 parser.cpp
13 parser.o
14 shellFunc.cpp
15 shellFunc.o
16

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ ls -a > minhyukResult.txt
minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ ls -al | grep main
-rw-r--r-- 1 minhyuk minhyuk  455 Sep 28 14:41 main.cpp
-rw-r--r-- 1 minhyuk minhyuk 13416 Sep 28 19:50 main.o
minhyuk@DESKTOP-9KUSLQP:~/OperatingSystem/HW#01-SimpleMyShell$
```

Fig 23. Redirection Result

Fig 23과 같이 Redirection인 '>', '|'이 정상적으로 작동하는 것을 확인할 수 있다.



```
minhyuk:/home/minhyuk/OperatingSystem/HW#01-SimpleMyShell$ quit
Finish SiSH Shell !!!
minhyuk@DESKTOP-9KUSLQP:~/OperatingSystem/HW#01-SimpleMyShell$
```

Fig 24. Finish Shell Result

Fig 24와 같이 quit 명령어를 입력함으로써 Simple Shell을 종료하고 bash Shell로 돌아오는 것을 확인할 수 있다.

6. Conclusion

본 레포트에서는 서론에서 운영체제와 셸에 대한 개념을 언급하고, 본 레포트의 구성에 대해 언급하였다. 이후 2장에서 Simple Shell을 완성하기 위한 요구사항을 검토하고, 3장에서 Simple Shell을 구현하기 위한 관련 개념을 자세하게 살펴보았다. 4장에서는 2장과 3장을 구현하기 위한 아이디어를 제시하고 설계를 하였고, 더 자세하게 설계 또한 진행했다. 이후 코드 레벨에 대해 설계를 하고, 구현한 코드를 설명하였다. 마지막으로 5장에서 Simple Shell 프로그램을 작동시킨 결과를 확인함으로써 정상적으로 프로그램이 실행된다는 것을 확인할 수 있었다. 시스템 프로그래밍이란 low-level 수준의 프로그래밍이기에 관련 개념에 대한 이해가 부족하면 구현하기 쉽지 않다. 본 레포트를 통해 시스템 프로그래밍의 핵심 요소 중 하나인 셸 프로그램을 설계하고 구현하는 과정을 체계적으로 다룰 수 있었다. 특히, 셸에서 중요한 프로세스 생성과 명령어 실행의 기초가 되는 `fork()`와 `exec()` 함수의 동작 원리와 그 사용법을 깊이 있게 이해하고 적용하는 데 중점을 두었다. 이러한 과정에서, 운영 체제의 프로세스 관리 및 시스템 자원 제어에 대한 중요한 개념들을 배우고 이를 실제로 구현함으로써 시스템 프로그래밍의 기본 원리를 익힐 수 있었다. Simple Shell 프로그램의 성공적인 구현은 단순한 명령어 해석을 넘어서, 리다이렉션, 파이프, 프로세스 관리와 같은 고급 기능들을 통해 셸이 실질적으로 어떻게 동작하는지를 탐구하는 기회가 되었다. 이를 통해 시스템 프로그래밍이 컴퓨터 시스템 내부에서 어떻게 동작하는지에 대한 이해를 더욱 심화할 수 있었다. 결론적으로, 본 프로젝트는 시스템 프로그래밍에서 셸의 역할과 기능을 직접 구현해보는 좋은 기회였으며, 이 과정에서 다양한 도전과 해결 방안을 경험할 수 있었다. 시스템의 동작 원리에 대한 깊은 이해와 실질적인 문제 해결 능력을 키우는 데 기여했으며, 이를 바탕으로 향후 더 복잡한 시스템 프로그램 구현에도 도전할 수 있는 자신감을 얻게 되었다.