OPERATING SYSTEMS
HW1 :: MyShell

By using C prigramming

모바일시스템공학과 손보경 (32212190)
ssg020910@naver.com

2024.09.29.

목 차

1. Introduction

쉘(Shell)은 작업 제어 시스템(job control system)으로, 프로그래머가 프로그램을 생성하고 관리하여 특정 작업을 수행할 수 있게 해준다. 즉, 사용자가 운영체제와 직접 상호작용할 수 있게 해주는 명령어 해석기(command interpreter)이다. 사용자가 파일 조작과 같은 작업(파일 생성, 삭제, 복사, 실행 등)을 명령어로 입력하면 이를 처리하고 실행한다. 대부분 운영체제, 예를 들어 LINUX, UNIX, Windows에는 기본적으로 명령어 해석기가 포함되어 있으며, 프로세스를 시작할 때 실행된다.

이 보고서에서는 쉘 프로그램의 작동, 구현, 그리고 프로젝트의 평가가 담겨있다. 간단한 쉘을 프로그램을 만들어 사용자가 입력한 문자열을 받아 명령을 실행하는 프로그램을 작성해보겠다.

2. Requirement

Category	Requirement
과제 설명	- 쉘 프로그램 작성 (Simple Shell. SiSH)
쉘의 기능	- 입력 : 사용자가 입력한 문자열을 받아서 해당 프로그램 실행 - 프로세스 생성 : fork()을 사용하여 새로운 프로세스 생성 - 프로그램 실행 : execve()을 사용하여 입력된 프로그램 실행 - 대기 : wait()을 사용하여 부모 프로세스가 자식 프로세스의 실행이 완료될 때까지 대기
요구 사항	- 시작 : 실행 가능한 파일 이름을 입력하면 쉘 시작 - 종료 : 사용자로부터 ‘quit’문자열을 받으면 쉘 종료 - 입력 경로 : PATH 환경 변수를 사용하여 파일 시스템에서 프로그램 검색 - 입력 반복 : 프로그램 실행이 완료되면 다음 입력을 받을 수 있어야 함

선택적 기능	- Makefile : 코드를 컴파일할 수 있는 Makefile 작성 - 프롬프트 : getenv() 함수를 사용하여 다양한 쉘 프롬프트를 지정 - 추가 인자 : 실행할 프로그램에 추가 인자를 입력받아 전달.
문서화	- 프로그램에 대한 간단한 소개 - 프로그램을 만드는 방법에 대한 구체적인 설명 - 작동 예시 - 개인적인 아이디어나 피드백 포함.

3. Concepts

프로젝트 설명에 앞서, 주로 구현에 사용되는 개념인 LINUX, UNIX, Windows와 같은 운영체제에 구현된 명령어 해석기, 즉 쉘의 개념에 대해 알아보도록 하겠다. 또한, 시스템 호출의 의미와 구현할 각각의 시스템 호출 명령어들에 대해 살펴보겠다.

3-1. What is Shell

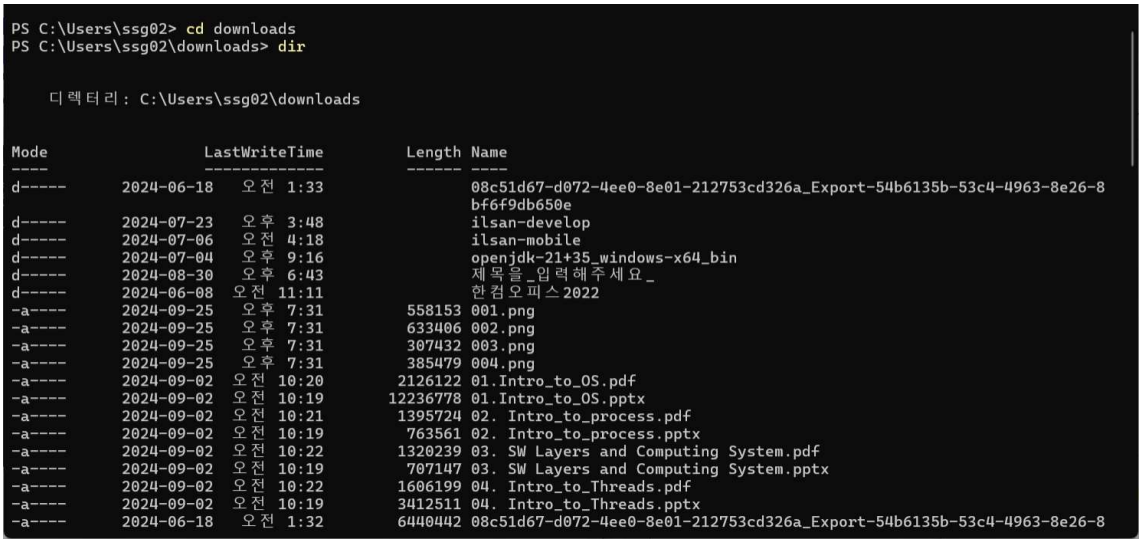


그림 1 Windows Command Shell

쉘은 운영체제에서 제공하는 명령을 실행하는 프로그램으로, 파일, 프린터, 하드웨어 장치 및 애플리케이션과의 인터페이스를 제공한다. 사용자가 쉘을 통해 명령(생성, 삭제, 복사, 실행 등의 파일 조작)을 입력하면 운영체제가 해당 작업을 수행한다. 쉘의 명령은 두 가지 방식으로 구현될 수 있다.

첫 번째 방법은 쉘 내장 구현이다. 쉘이 특정 명령을 처리하는 코드를 포함하고 있어, 입력된 명령어를 해석한 후 직접 실행 코드를 호출한다. 예를 들어, rm file.txt를 입력하면 쉘은 해당 명령을 해석하고 필요한 매개변수를 설정하여 시스템 호출을 통해 파일을 삭제한다.

두 번째 방법은 외부 프로그램 호출 방식이다. 쉘이 명령어 처리 코드를 포함하지 않고, 별도의 프로그램을 호출한다. 사용자가 입력한 명령어는 파일 시스템에서 실행할 프로그램의 경로로 해석된다. 예를 들어, ls 명령을 입력하면 쉘은 경로를 찾고 fork()를 통해 새로운 프로세스를 생성한 후, 자식 프로세스에서 ls 프로그램을 실행하고 부모 프로세스는 자식 프로세스의 종료를 기다린다.

쉘은 시작할 때 환경을 설정하고, 사용자의 환경 변수 및 구성 파일을 로드합니다. 그 후 명령어를 해석하여 동작을 결정하고, 모든 명령어 실행 후 자원을 정리한 뒤 종료된다.

명령어 처리 단계는 다음과 같이 3가지로 나뉜다. 사용자가 입력한 명령어를 읽어들이는 단계, 어들인 명령어 문자열을 해석하여 필요한 요소로 분리하는 단계, 구문 분석된 명령어를 실제로 실행하는 단계로 나뉜다.

3-2. System Calls

시스템 호출(System Call)은 사용자 프로그램이 운영체제의 서비스나 기능을 요청할 수 있도록 하는 인터페이스이다. 운영체제는 하드웨어와 사용자 프로그램 간의 중재자 역할을 하며, 시스템 호츨을 통해 프로그램이 파일 입출력, 프로세스 생성, 메모리 관리 등 다양한 기능을 사용할 수 있게 한다. 시스템 호출은 일반적으로 사용자 모드에서 커널 모드로 전환하고, 호출할 시스템 호출의 번호와 필요한 매개변수를 커널에 전달한 후, 운영체제가 요청된 작업을 수행하고 결과를 사용자 프로그램에 반환하는 과정을 포함한다.

주요 시스템 호출로는 fork(), execve(), wait()가 있다. fork()는 새로운 프로세스를 생성하며, 부모 프로세스의 복사본인 자식 프로세스가 독립적으

로 실행된다. 이 호출은 부모 프로세스에 자식 프로세스인 PID를 반환하고, 자식 프로세스에는 0을 반환한다. execve()는 현재 프로세스를 완전히 새로운 프로그램으로 대체하여, 프로세스의 메모리 공간을 새로운 프로그램으로 덮어씌우고 그 프로그램을 실행한다. 이 호출은 실행할 프로그램의 경로와 인자 목록, 환경 변수 목록을 매개변수로 받는다. wait()는 부모 프로세스가 자식 프로세스가 종료될 때까지 대기하게 하며, 자식 프로세스가 종료되면 부모는 종료 상태를 받아 처리하고 자원을 해제한다.

또한, C 언어에서 제공하는 라이브러리 함수로는 strtok_r()와 getenv()가 있다. strtok_r()는 문자열을 구분자로 분리하여 토큰으로 나누는 함수로, thread-safe하여 여러 스레드가 동시에 사용할 수 있다. 이 함수는 분리할 문자열과 구분자, 상태 정보를 유지할 포인터를 매개변수로 받으며, 쉘에서 입력된 명령어와 인자를 구분할 때 사용된다. getenv()는 환경 변수를 조회하여 그 값을 반환하며, 쉘에서 PATH 환경 변수를 통해 명령어가 위치한 디렉토리를 찾는 데 사용된다.

따라서 시스템 호출은 운영체제의 기능을 요청하는 인터페이스로, fork(), execve(), wait()는 프로세스와 메모리 관리와 관련된 시스템 호출이며, strtok_r()와 getenv()는 문자열 처리와 환경 변수 조회에 사용되는 C 언어의 라이브러리 함수이다. 이러한 시스템 호출과 라이브러리 함수는 운영체제와 프로그램 간의 효율적인 상호작용을 가능하게 하여, 프로그래머가 원하는 기능을 쉽게 구현할 수 있도록 돕는다.

4. Implementation

4-1. File Structure

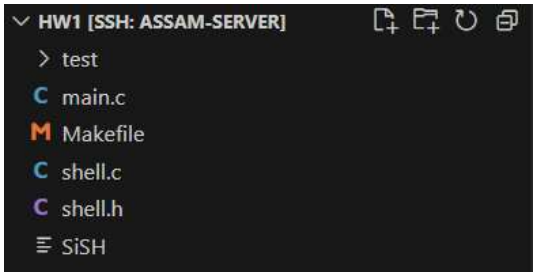


그림 2 파일 구조 (실제 파일 디렉토리)

그림 3 파일 구조 (도식화)

4-2. Makefile()

그림 4 Makefile의 전문

이 Makefile은 SiSH 프로그램의 빌드 과정을 자동화하기 위한 설정 파일이다. gcc 컴파일러를 사용하며, -Wall 플래그를 통해 모든 경고를 표시하고, -g 플래그로 디버깅 정보를 포함하도록 설정했다. 프로그램은 main.c와 shell.c 소스 파일을 컴파일하여 SiSH라는 실행 파일을 생성한다. make all 명령어를 통해 빌드를 수행하며, make clean 명령어는 빌드 후 생성된 파일을 삭제하는 역할을 한다.

4-3. main.c 파일

그림 5 main.c 파일의 헤더 파일과 main() 함수의 변수 선언 부분

<stdio.h>, <stdlib.h>, <string.h>, <unistd.h>는 기본적인 입출력, 문자열 처리, 환경 변수 접근 및 기타 유틸리티 기능을 사용하기 위해 선언하였으며, shell.h는 쉘 관련 함수의 선언이 포함된 헤더 파일이다.

char cmd[MAX_CMD_LEN]는 사용자가 입력한 명령을 저장할 문자열 버퍼이며, char *args[MAX_ARGS]는 명령어와 인자를 파싱한 결과를 저장할 배열이다. char *user와 char *pwd는 각각 현재 사용자 이름과 현재 작업 경로를 저장할 포인터이다.

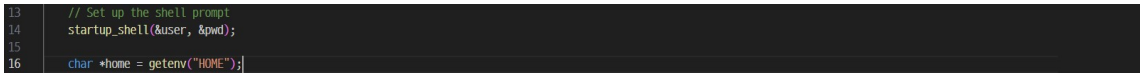


그림 6 쉘 프롬프트 설정 및 홈 디렉터리 처리

startup_shell(&user, &pwd)는 쉘이 시작될 때 사용자 이름과 현재 작업 경로를 설정하며, 여러 출력값을 표시한다. 이 함수는 추후 shell.c 파일 파트에서 자세히 다루도록 하겠다.

char *home = getenv("HOME")는 홈 디렉터리 경로를 가져오는 부분이다. 이는 프롬프트가 출력될 때 조금 더 깔끔하게 출력하기 위해 작성하였다.

```
18      while (1) {
19          // Print the prompt
20          if (strncmp(pwd, home, strlen(home)) == 0) {
21              // Replace home path with '~'
22              printf("%s@SiSH: ~%s$ ", user, pwd + strlen(home));
23          } else {
24              printf("%s@SiSH: %s$ ", user, pwd);
25          }
26
27          // Get user input
28          if (fgets(cmd, sizeof(cmd), stdin) == NULL) {
29              break; // Exit on EOF
30          }
31
32          // Check for exit command
33          if (strcmp(cmd, "quit\n") == 0) {
34              break;
35          }
36
37          // Parse the command into arguments
38          parse_command(cmd, args);
39
40          // Execute the command
41          execute_command(args, &pwd);
42      }
```

그림 7 메인 루프 전문

해당 부분은 main()에서 중요한 부분이며 사용자의 명령을 반복해서 입력받고 처리하는 부분이며, 종료 명령이나 EOF가 입력될 때까지 실행된다.

20~25줄은 프롬프트를 출력하는 코드이며, 현재 작업 경로가 홈 디렉터리와 동일할 경우, 홈 디렉터리를 ~로 대체하여 출력하고 그렇지 않으면 전체 경로를 그대로 출력할 수 있게 해주는 구문이다. 덕분에 프롬프트 형식이 조금 더 깔끔하고 간결하게 변경되었다.

fgets() 함수를 사용해 사용자로부터 명령을 입력받고, 사용자가 quit를 입력하면 루프가 종료되고, 프로그램이 종료된다. parse_command(cmd, args) 함수를 통해 입력된 명령을 인자로 파싱하며, execute_command(args, &pwd) 함수를 이용해 명령어를 실행하고, 필요한 경우 pwd를 업데이트한다.

```
44      printf("Goodbye! MyShell!\n");
45      return 0;
46  }
```

그림 8 프로그램 종료 부분

루프가 종료되면, 메시지를 출력하고 프로그램이 종료된다.

4-4. shell.c 파일

shell.c 파일은 사용자 명령을 처리하고 실행하는 쉘의 핵심 기능을 구현하고 있다. 쉘의 주요 작업에 따라 함수를 나눴으며, 쉘을 시작하는 부분, 명령어를 해석하는 부분, 명령을 실행하는 부분으로 나누었다.

```c
#ifndef SHELL_H
#define SHELL_H

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

// Function prototypes
void startup_shell(char **user, char **pwd);
int parse_command(char *cmd, char **args);
void execute_command(char **args, char **pwd);

#endif // SHELL_H
```

그림 9 shell.h 파일의 전문

shell.h 파일은 매크로와 함수 프로토타입이 작성되어 있는데, #define MAX_CMD_LEN 1024는 사용자가 입력할 수 있는 최대 명령어 길이를 1024로 정의한 것이며, #define MAX_ARGS 100는 명령어에 포함될 수 있는 최대 인자 수를 100으로 정의한 것이다.

함수 프로토타입은 총 3가지로 이루어져 있으며, void startup_shell(char **user, char **pwd)는 쉘 시작 시 사용자 이름과 경로를 설정하고 출력하는 함수이며, int parse_command(char *cmd, char **args)는 명령어를 인자 리스트로 파싱하는 함수, void execute_command(char **args, char **pwd)는 명령어를 실행하고, 필요할 경우 현재 작업 디렉터리를 업데이트하는 함수이다. 지금부터 각각의 함수의 세부 내용에 대해서 알아보겠다.

```c
// Function to startup information
void startup_shell(char **user, char **pwd) {
    printf("\n\n");

    // user name
    *user = getenv("USER");
    if (*user == NULL) {
        *user = "unknown";
    }

    printf("User: %s\n", *user);

    // real time
    time_t current_time = time(NULL);
    struct tm *local_time = localtime(&current_time);

    printf("Time: %02d:%02d:%02d\n", local_time->tm_hour, local_time->tm_min, local_time->tm_sec);

    // Path
    *pwd = getenv("PWD");
    if (*pwd == NULL) {
        printf("Unable to get PWD environment variable.\n");
    } else {
        printf("Path: %s\n\n", *pwd);
    }

    printf("Welcome MyShell!\n\n\n");
}
```

그림 10 startup_shell(char **user, char **pwd)

이 함수는 쉘이 시작할 때 호출되며, 사용자 이름, 현재 시간, 현재 작업 디렉터리 등을 설정하고 출력한다.

getenv("USER")를 통해 환경 변수 USER에서 사용자 이름을 가져오며, 만일 가져오지 못한다면 unknown으로 기본값을 설정한다. 다음으론 time() 과 localtime() 함수를 사용해 현재 시스템 시간을 가져와서 출력한다. 이후 getenv("PWD")를 통해 현재 작업 디렉터리 경로를 설정하고 출력한다.

```c
// Function to remove quotes from a string
void remove_quotes(char *arg) {
    char *src = arg, *dst = arg;
    while (*src) {
        if (*src != '"' && *src != '\'') {
            *dst++ = *src;
        }
        src++;
    }
    *dst = '\0'; // Null-terminate the modified string
}
```

그림 11 remove_quotes(char *arg)

이 함수는 명령어 인자에서 따옴표를 제거하는 함수이다. 이를 통해 ehco를 사용할 때, 따옴표를 사용 여부에 상관없이 문자열을 깔끔하게 출력할 수 있다.

```c
// Function to split the command into arguments
int parse_command(char *cmd, char **args) {
    int count = 0;
    char *token = strtok(cmd, " \n");
    while (token != NULL && count < MAX_ARGS - 1) {
        remove_quotes(token);
        args[count++] = token;
        token = strtok(NULL, " \n");
    }
    args[count] = NULL; // Null-terminate the argument list
    return count;
}
```

그림 12 parse_command(char *cmd, char **args)

parse_command 함수는 사용자가 입력한 명령어를 공백을 기준으로 분리하여 인자 리스트로 파싱한다. strtok() 함수를 사용해 명령어를 공백 또는 개행 문자로 구분된 토큰으로 나누고, 인자 배열에 저장한다. remove_quotes()를 사용해 각 토큰에서 따옴표를 제거하며 마지막에 NULL을 추가형 인자 리스트를 종료할 수 있게 하였다.

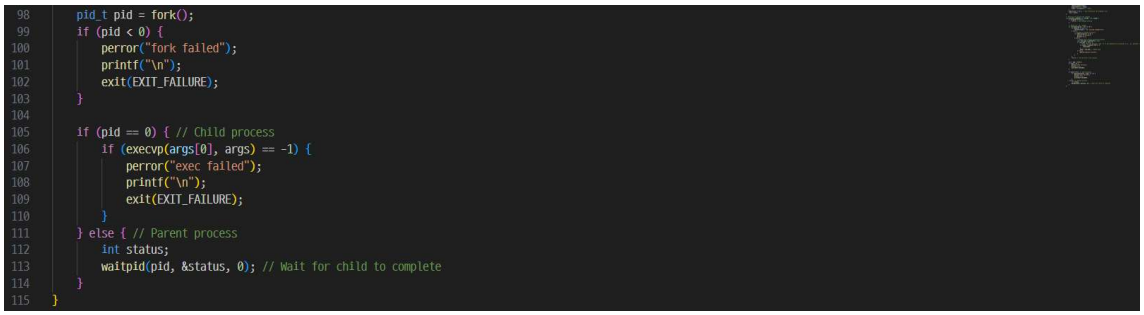
```c
// Function to execute the command
void execute_command(char **args, char **pwd) {
    if (args[0] == NULL) {
        return; // No command entered
    }

    // Check for 'cd' command
    if (strcmp(args[0], "cd") == 0) {
        if (args[1] == NULL) {
            fprintf(stderr, "cd: missing argument\n");
        } else {
            // Attempt to change directory
            if (chdir(args[1]) != 0) {
                perror("cd failed");
                printf("\n");
```

```
            } else {
                // Update the current working directory
                char *new_pwd = getcwd(NULL, 0);
                if (new_pwd != NULL) {
                    // Only free the previous `pwd` if it was dynamically allocated (i.e., by `getcwd()`)
                    if (*pwd != getenv("PWD")) {
                        free(*pwd);
                    }
                    *pwd = new_pwd; // Update pwd
                } else {
                    perror("getcwd failed");
                }
            }
        }
        return; // Do not fork a new process
    }
```

그림 14 execute_command(char **args, char **pwd) 중 빈 명령어 처리 구문과 cd 명령어 처리 구문

이 함수는 파싱된 명령어를 실제로 실행하는 기능을 한다. 만약 인자가 없으면 아무 작업도 하지 않고 함수를 종료하며, 사용자가 cd 명령어를 입력하면, 해당 디렉터리로 이동을 시도한다. 이동에 성공하면 getcwd()로 현재 작업 디렉터리를 업데이트하고, 필요시 메모리 해제를 하고, 실패 시 오류를 출력한다. cd 명령어는 따로 자식 프로세스가 필요하지 않기 때문에 그대로 함수를 종료한다.



```
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        printf("\n");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) { // Child process
        if (execvp(args[0], args) == -1) {
            perror("exec failed");
            printf("\n");
            exit(EXIT_FAILURE);
        }
    } else { // Parent process
        int status;
        waitpid(pid, &status, 0); // Wait for child to complete
    }
}
```

그림 15 자식 프로세스 생성 부분

cd 명령어 외 다른 명령어는 fork()를 통해 자식 프로세스를 생성한 후, 자식 프로세스에서 execvp() 함수를 사용해 명령어를 실행한다. 만약, fork()가 실패하면 오류를 출력하고 프로그램을 종료하고, fork() 성공시 자식 프로세스를 실행한다. 자식 프로세스에서 execvp()로 명령어를 실행하고, 실패 시 오류 메시지를 출력하고 종료한다. 부모 프로세스는 자식 프로세스가 끝날 때까지 waitpid()로 대기한다.

5. Result

이렇게 제작된 MyShell을 실행한 결과를 보여주도록 하겠다. 컴파일부

터, 여러 기본 명령어, 파일 실행 등을 보여주도록 하겠다.

```
assam_bokyeong21@assam:~/hw1$ ls
main.c  Makefile  shell.c  shell.h  test
assam_bokyeong21@assam:~/hw1$ make
gcc -Wall -g -o SiSH main.c shell.c
assam_bokyeong21@assam:~/hw1$ ls
main.c  Makefile  shell.c  shell.h  SiSH  test
assam_bokyeong21@assam:~/hw1$ ./SiSH


  User: assam_bokyeong21
  Time: 17:30:49
  Path: /home/assam_bokyeong21/hw1

  Welcome MyShell!


  assam_bokyeong21@SiSH: ~/hw1$
```

그림 16 Makefile을 통해 컴파일 후, SiSH 프로그램을 실행한 모습

```
assam_bokyeong21@SiSH: ~/hw1$ ls -l
total 44
-rw-rw-r-- 1 assam_bokyeong21 assam_bokyeong21  1064  9월 27 21:41 main.c
-rw-rw-r-- 1 assam_bokyeong21 assam_bokyeong21   159  9월 27 19:56 Makefile
-rw-rw-r-- 1 assam_bokyeong21 assam_bokyeong21  3031  9월 28 16:18 shell.c
-rw-rw-r-- 1 assam_bokyeong21 assam_bokyeong21   257  9월 27 21:41 shell.h
-rwxrwxr-x 1 assam_bokyeong21 assam_bokyeong21 23200  9월 28 17:30 SiSH
drwxrwxr-x 2 assam_bokyeong21 assam_bokyeong21  4096  9월 27 20:42 test
assam_bokyeong21@SiSH: ~/hw1$
```

그림 17 ls -l 명령어를 사용한 모습

```
assam_bokyeong21@SiSH: ~/hw1$ echo hello
hello
assam_bokyeong21@SiSH: ~/hw1$ echo "hello"
hello
assam_bokyeong21@SiSH: ~/hw1$ echo "hello'
hello
assam_bokyeong21@SiSH: ~/hw1$ echo "he"llo'
hello
assam_bokyeong21@SiSH: ~/hw1$
```

그림 18 echo 명령어를 실행한 모습

```
assam_bokyeong21@SiSH: ~/hw1$ ls
main.c  Makefile  shell.c  shell.h  SiSH  test
assam_bokyeong21@SiSH: ~/hw1$ cd a
cd failed: No such file or directory
```

```
assam_bokyeong21@SiSH: ~/hw1$ cd ./a
cd failed: No such file or directory

assam_bokyeong21@SiSH: ~/hw1$ cd test
assam_bokyeong21@SiSH: ~/hw1/test$ cd ..
assam_bokyeong21@SiSH: ~/hw1$
```

그림 20 cd 명령어 사용 모습

```
assam_bokyeong21@SiSH: ~/hw1$ ls
main.c  Makefile  shell.c  shell.h  SiSH  test
assam_bokyeong21@SiSH: ~/hw1$ mkdir a
assam_bokyeong21@SiSH: ~/hw1$ ls
a  main.c  Makefile  shell.c  shell.h  SiSH  test
assam_bokyeong21@SiSH: ~/hw1$ rmdir a
assam_bokyeong21@SiSH: ~/hw1$ ls
main.c  Makefile  shell.c  shell.h  SiSH  test
assam_bokyeong21@SiSH: ~/hw1$
```

그림 21 디렉토리를 생성하고 제거한 모습

```
assam_bokyeong21@SiSH: ~/hw1/test$ ./hello
Hello. This is my first HW!
assam_bokyeong21@SiSH: ~/hw1/test$ ./single_cycle
Cycle[001](PC : 0x00000000)======================================================

[Fetch Inxtruction] 27BDFFD8
[Decode Instruction] type : I
        OPCODE : 0x9, RS : 0x1D (R[29] = 0x10000000), RT : 0x1D (R[29] = 0x10000000), IMM : 0xFFD8
[Write Back] R[29] <- 0xFFFFFD8[PC Update] PC <- 0x00000004 = 0x00000000 + 4


Cycle[002](PC : 0x00000004)======================================================

[Fetch Inxtruction] AFBF0024
[Decode Instruction] type : I
        OPCODE : 0x2B, RS : 0x1D (R[29] = 0xFFFFFD8), RT : 0x1F (R[31] = 0xFFFFFFFF), IMM : 0x24
[Store] Mem[0x3FFFFF] <- R[31] = 0xFFFFFFFF[PC Update] PC <- 0x00000008 = 0x00000004 + 4
```

그림 22 여러 파일을 실행한 모습

```
assam_bokyeong21@SiSH: ~/hw1/test$
assam_bokyeong21@SiSH: ~/hw1/test$ quit
Goodbye! MyShell!
assam_bokyeong21@assam:~/hw1$
```

그림 23 문자열이 NULL일 때와 quit를 입력한 모습

이렇듯 대부분의 명령어가 잘 작동하는 모습을 볼 수 있다.

6. Build Environment

MyShell을 실행하기 위해서는 GCC 컴파일러가 필요하며, 해당 프로그램은 Makefile을 사용하여 빌드되기 때문에 프로그램이 위치한 디렉토리 내에서 다음과 같은 명령어를 사용하면 프로그램이 실행된다.

$make / $make all / $make SiSH : MyShell의 실행 파일을 빌드
$make clean : 모든 오브젝트 파일 및 실행 파일 정리

7. Lesson

7-1. Conclusion

이렇게 C 언어를 이용하여 간단하게 MyShell 프로그램을 만들어봤다. 특히 쉘의 생애 주기를 모티브로 함수를 구현하였으며, 프로그램 실행 중 가시성이 좋게 프롬프트 수정 및 echo 응답 문자열 처리 등을 추가하여 제작해 보았다. 결과물에서 알 수 있듯이, ls, cd, mkdir, 프로그램 실행 등의 명령어가 잘 처리되는 것을 볼 수 있었다.

7-2. Review

오랜만에 C 언어를 이용하여 코딩하느라 조금 애를 쓴 듯하다. 알고리즘 구현에서도 어떻게 하면 깔끔하게 작성할 수 있을지 많이 고민했다. 특히 기존 Shell에서 명령어가 어떻게 작동하는지 비교해가며 코드를 계속 수정하며 완성도를 높인 것 같다. 처음 작성한 코드에서 cd 명령어를 입력한 후 뒤에 경로의 시작이 ./ 이면 오류가 발생하는 버그가 있었는데, 이를 해결하기 위해 많은 시간을 들였다. 이번 과제를 수행하며 C 언어 포인터에 대해 다시금 복습할 수 있었으며 리눅스 환경에 대해 조금 더 알아갈 수 있었던 것 같다.

7-3. Using ChatGPT

· 과제의 요구 사항을 정리하는 부분에서 ChatGPT의 도움을 받았다.
· assam 서버의 파일을 VSCode에서 수정하고 만들기 위한 방식을 찾을 때 사용하였다. 그 결과 Remote-SSH 확장 프로그램을 사용하여 로컬에서 편집하는 것처럼 편리하게 작업할 수 있었다.
· 기존 코드에서 cd 명령어의 인자가 ./로 시작될 때 경로를 찾지 못하고 오류 코드를 전송한 후, 프로그램이 종료되는 오류가 있었다. 이 오류의 이유를 찾기 위해 chatGPT에게 물어봤고 getenv("PWD")로 받은 pwd의 메모리를 해제하여 생긴 오류임을 알게 되어 코드를 수정했다.