Cykor 2주차
리눅스 환경에서의 Shell 구현 보고서

학번 : 2025350213
이름 : 승창민

1. 과제 목적

본 과제는 C언어를 사용하여 리눅스 환경에서 동작하는 Shell을 구현하고, 기본 명령어 처리, 파이프라인, 조건 실행 등을 지원하는 프로그램을 제작하는 것을 목표로 한다.

2. 프로그램 발전 단계

[1]

1) 기본적인 Shell을 형성하고, 그 구성요소를 공부함.

아래는 초기 shell을 간단하게 정리한 의사코드

```
main()                              # 종료할 때까지 작동하는 무한루프
print_prompt()                      # 프롬프트 메시지를 출력

execute_command()
        void parse_command()  # 입력문을 잘라서 인식
                              #인식된 결과에 따라 작업 실행
        if 입력값 = "cd"
                shell_cd ()       # 작업이 진행되는 디렉토리를 변경
        else if 입력값 = "pwd"
                shell_pwd ()      # 현재 작업 중인 디렉토리 표시
                                  # pwd는 Print Working Directory의 줄임말이었음.
        else if 입력값 = "exit"
                exit (0)          # shell 종료 트리거

        else fork() -> 자식 프로세스 생성     #외부 명령어에 대한 처리
```

과제 조건 중 달성한 것은 아래와 같음.

- 디렉토리 이름 표시
- bash 스타일 인터페이스
- cd, pwd 구현
- exit시 종료

2) 달성하지 못한 과제의 조건들을 다음 목표로 설정함

-파이프라인 지원
-다중 명령어 처리
-백그라운드 실행

[2]

1) 다중명령어의 예시를 찾아봄.
A ; B - 명령어 구분
A && B - A가 실패하는 경우(False인 경우)에만 B 실행
A || B - A가 성공하는 경우(True인 경우)에만 B 실행

2) A ; B
전체 문장을 입력받고 ';'를 기준으로 입력값을 구분하는 execute_clear() 함수 생성
작업을 실행하는 execute_command를 execute_clear 내부에 재사용함.
main 함수의 execute_command를 execute_clear로 바꿔줌.

main -> execute_clear -> execute_command -> execute_clear -> main

3) A && B, A || B
execute_clear를 실행하기 전에 '&&'나 '||'가 있는지 검사 후, 해당 문자 기준으로 함수를 구분하면 되겠다고 생각함.
새로운 함수 execute_conditional_line을 만듬.
변수 last_status에 이전 함수 실행 결과를 0이나 1 등으로 저장하는 방식을 사용하기로 함.
&&일 때와 ||일 때를 분리하고, 각각 last_status 값에 따라 다음 작업 실행 여부를 결정.

[3]

1) 백그라운드 설계
입력값을 쪼개는 단계에서 &를 감지하면 프로세스를 백그라운드 실행으로 넘기기로 함.
작업을 실행하는 execute_command 단계에 백그라운드 프로세스 분기를 추가함.

2) 좀비 프로세스 제거
과제 조건 중에 좀비 프로세스[1]에 대한 내용이 있었기 때문에 해당 내용에 대해 조사해봄.
좀비 프로세스로 인한 문제는 종료된 자식 프로세스를 부모 프로세스가 수거하지 않아서 발생하기 때문에, 자식 프로세스가 종료를 확인함과 동시에 정리해버리면 되지 않을까 생각.

[4]

1) 파이프 라인 지원 설계

---

1) 부모 프로세스가 wait() 등으로 종료된 자식 프로세스를 수거하지 않아서 생기는 문제로, PID 낭비를 발생시킬 수 있다고 함.

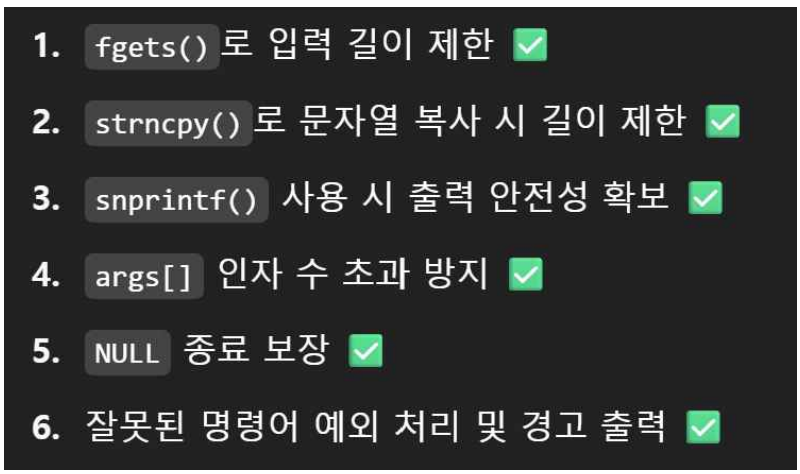
'|' 로 명령어를 구분하여 명령어마다 자식 프로세스를 생성하고, 파이프를 통해 한 자식 프로세스의 결괏값이 다른 자식 프로세스의 입력값으로 들어가는 게 목표.

2) 명령어 구분 & 파이프 준비
'|'로 명령어들을 구분하고, fork를 이용해서 명령어별로 자식 프로세스를 생성함.
자식 프로세스의 사이에 파이프가 하나씩 필요하니, '명령어(자식 프로세스 수) 개수 -1'만큼 파이프를 준비

3) 자식 프로세스의 연결, 명령어 실행
dup2 함수를 이용해서 자식 프로세스들의 입출력을 서로 연결.
이후 파이프를 닫아서, 하나의 프로세스 종료 -> 데이터 전달.
마지막으로 명령어 실행.

[5]
1) 오버플로우?
지금 만든 shell의 경우 반복적이거나, 재귀적인 구조가 존재하지는 않으니 별도의 무한루프에 빠지는 일은 없을 것 같음.
하지만 너무 많은 파이프라인을 연결하면 문제가 발생할 수도 있지 않을까 생각이 듬.
-> 파이프라인 생성 개수에 제한을 둔다면 해결되지 않을까?

2)
이전에 C언어에서 scanf() 등으로 입력을 받을 때 버퍼 오버로우의 위험이 있다는 내용을 본 적이 있어, shell에도 과도한 입력값이 들어오면 위험할 수도 있겠다는 생각이 들었다.
해당 내용을 ChatGPT에 물어봄.



등의 방법이 있다고 한다.
이번에는 시간이 부족해서 shell을 더 발전시키지는 못했지만, 다음에 기회가 된다면 이런 요소들을 참고하면 어떨까 싶다.

3. 구현 개요

<핵심 기능>
프롬프트 표시: 현재 디렉토리 경로를 포함한 프롬프트(mysh:/path>) 출력
내장 명령어 처리: cd, pwd, exit 명령어를 직접 구현
일반 명령어 실행: execvp()를 사용하여 외부 명령 실행
다중 명령어 처리 (;): 한 줄에 여러 명령어를 입력하여 순차 실행 가능
조건부 실행 (&&, ||): 앞 명령의 성공 여부에 따라 다음 명령 실행
백그라운드 실행 (&): &가 붙은 명령은 백그라운드에서 실행하며, 좀비 프로세스를 방지하기 위해 SIGCHLD 시그널 핸들링 포함
파이프라인 처리 (|): 명령어들을 파이프라인으로 연결하여 stdin/stdout을 통해 데이터 흐름 전달

4. 함수별 역할

- main()
사용자 입력 루프 / 시그널 핸들러 등록
- print_prompt()
현재 경로 기반 프롬프트 출력
- execute_line()
; 단위 명령어 분리 및 실행
- execute_conditional_line()
&&, || 조건 분기 실행
- execute_pipeline()
| 파이프라인 연결 및 다중 프로세스 처리
- parse_command()
공백 기준 인자 분해 및 백그라운드 여부 판별
- shell_cd(), shell_pwd()
내장 명령어 직접 구현
- handle_sigchld()
백그라운드 좀비 제거용 시그널 핸들러

5. 최종 완성 코드

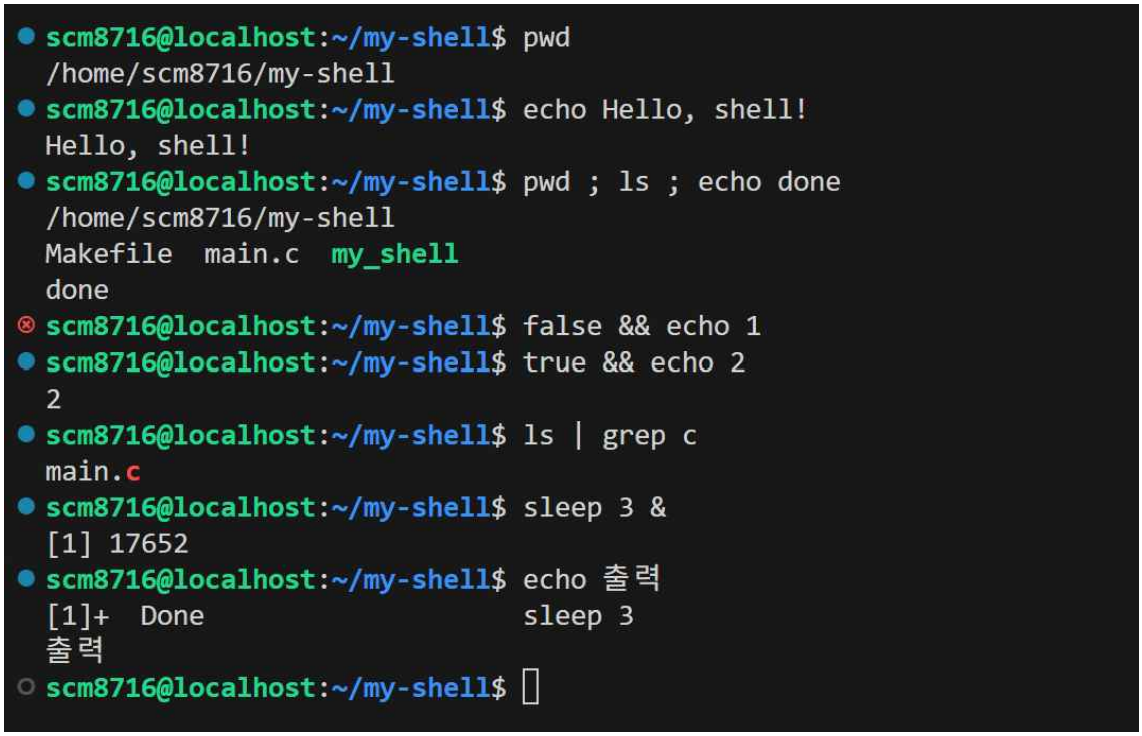
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <limits.h>
#include <signal.h>


#ifndef PATH_MAX
#define PATH_MAX 4096
#endif

#define MAX_LINE 1024
#define MAX_ARGS 64

void print_prompt() {
    char cwd[PATH_MAX];
    getcwd(cwd, sizeof(cwd));
    printf("mysh:%s> ", cwd);
}

int parse_command(char* line, char** args) {
    int i = 0;
    args[i] = strtok(line, " \t\r\n");
    while (args[i] != NULL && i < MAX_ARGS - 1) {
        i++;
        args[++i] = strtok(NULL, " \t\r\n");
    }
    args[i] = NULL;

    if (i > 0 && strcmp(args[i - 1], "&") == 0) {
        args[i - 1] = NULL;
        return 1;
    }
    return 0;
}

void shell_cd(char** args) {
    if (args[1] == NULL) {
        fprintf(stderr, "cd: 경로가 없습니다\n");
    }
    else {
        if (chdir(args[1]) != 0) {
            perror("cd");
        }
    }
}

void shell_pwd() {
    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
```

```
            printf("%s\n", cwd);
        }
        else {
            perror("pwd");
        }
}

void execute_command(char* line) {
    if (strchr(line, '|') != NULL) {
        execute_pipeline(line);
        return;
    }

    char* args[MAX_ARGS];
    int is_background = parse_command(line, args);
    if (args[0] == NULL) return;

    if (strcmp(args[0], "cd") == 0) {
        shell_cd(args);
    }
    else if (strcmp(args[0], "pwd") == 0) {
        shell_pwd();
    }
    else if (strcmp(args[0], "exit") == 0) {
        printf("Bye!\n");
        exit(0);
    }
    else {
        pid_t pid = fork();
        if (pid == 0) {
            execvp(args[0], args);
            perror("exec");
            exit(1);
        }
        else {
            if (!is_background) {
                waitpid(pid, NULL, 0);
            }
            else {
                printf("[백그라운드 pid: %d]\n", pid);
            }
        }
    }
}

void execute_line(char* line) {
    char* command = strtok(line, ";");
    while (command != NULL) {
        while (*command == ' ') {
            command++;
        }
        if (*command != '\0') {
            execute_command(command);
        }
        command = strtok(NULL, ";");
```

```
    }
}


void execute_pipeline(char* line) {
    char* commands[10];
    int cmd_count = 0;

    commands[cmd_count] = strtok(line, "|");
    while (commands[cmd_count] != NULL && cmd_count < 9) {
        cmd_count++;
        commands[cmd_count] = strtok(NULL, "|");
    }

    int pipes[2 * (cmd_count - 1)];

    for (int i = 0; i < cmd_count - 1; i++) {
        pipe(pipes + i * 2);
    }

    for (int i = 0; i < cmd_count; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            if (i > 0) {
                dup2(pipes[(i - 1) * 2], 0);
            }
            if (i < cmd_count - 1) {
                dup2(pipes[i * 2 + 1], 1);
            }

            for (int j = 0; j < 2 * (cmd_count - 1); j++) {
                close(pipes[j]);
            }

            char* args[MAX_ARGS];
            parse_command(commands[i], args);
            execvp(args[0], args);
            perror("exec");
            exit(1);
        }
    }

    for (int i = 0; i < 2 * (cmd_count - 1); i++) {
        close(pipes[i]);
    }

    for (int i = 0; i < cmd_count; i++) {
        wait(NULL);
    }
}

int execute_conditional_line(char* line) {
    char* token;
    char* rest = line;
    int last_status = 0;
```

```
    int run_next = 1;
    char* op = NULL;

    while ((token = strsep(&rest, "&&||")) != NULL) {
        while (*token == ' ') token++;
        if (*token == '\0') continue;

        if (run_next) {
            pid_t pid = fork();
            if (pid == 0) {
                char* args[MAX_ARGS];
                parse_command(token, args);
                execvp(args[0], args);
                perror("exec");
                exit(1);
            }
            else {
                int status;
                waitpid(pid, &status, 0);
                last_status = WIFEXITED(status) ? WEXITSTATUS(status) : 1;
            }
        }

        if (rest != NULL) {
            op = rest - 2;
            if (strncmp(op, "&&", 2) == 0)
                run_next = (last_status == 0);
            else if (strncmp(op, "||", 2) == 0)
                run_next = (last_status != 0);
        }
    }

    return last_status;
}

void handle_sigchld(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0);
}

int main() {
    signal(SIGCHLD, handle_sigchld);
    char line[MAX_LINE];
    while (1) {
        print_prompt();
        if (fgets(line, MAX_LINE, stdin) == NULL) break;
        execute_line(line);
    }
    return 0;
}
```

6. 실행 예시



7. 느낀 점

이번 과제를 통해 리눅스 환경에서 동작하는 shell을 C언어를 사용해서 구현할 수 있었다. shell이라는 개념 자체가 낯설고, 처음 보는 함수들이 대거 등장한 만큼 쉽지 않은 과제였지만, 그만큼 독학 과정에서 많은 것을 알아갈 수 있었다고 생각한다.