쉘 프로그램 구현 write-up

현재 디렉토리 관련 함수 작성

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "pwd.h"

void initialize_cwd(void) { // 현재 디렉토리를 사용자의 홈 디렉토리로 초기화
    chdir(getenv("HOME")); // 현재 디렉토리 변경
    setenv("PWD", getenv("HOME"), 1); // PWD 환경변수 설정
}

int pwd(void) { // 현재 작업 디렉토리 출력
    printf("%s\n", getenv("PWD"));
    return 0;
}
```

기본적으로 쉘을 실행시키면 초기 디렉토리는 사용자의 홈 디렉토리이다. 이를 구현하기 위해 먼저 환경변수를 이용해 현재 디렉토리를 홈 디렉토리로 바꿔주는 기능을 작성하였고, 다른 기능들에서도 활용할 PWD 환경변수도 현재 디렉토리의 위치와 일치하도록 업데이트 해줬다. 다음으론 사용자가 호출하여 사용할 수 있는 pwd 명령어를 구현했고 앞서 얘기한 PWD 환경변수를 활용하였다.

프롬프트 출력 함수 작성

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "print_prompt.h"

void print_prompt(void) {

    // 현재 디렉토리명 추출
    char *directory_name;
    int slashCount = 0;
    const char *p = getenv("PWD");

    while ((p = strchr(p, '/')) != NULL) {
        slashCount++; // '/'를 찾을 때마다 카운트 증가
        p++;      // 포인터를 다음 문자로 이동
    }

    if (slashCount == 1) { // 현재 디렉토리가 루트 혹은 루트 하위 디렉토리인 경우
        directory_name = getenv("PWD");
    } else if (strcmp(getenv("PWD"), getenv("HOME")) == 0) { // 현재 디렉토리가 홈 디렉토리인 경우
        directory_name = "~";
    } else {
        directory_name = strrchr(getenv("PWD"), '/') + 1; // 마지막 '/' 이후의 문자열
    }

    // 유저 이름 추출
    char *user = getenv("USER");

    // 호스트 이름 추출
    char *hostname = getenv("HOSTNAME");

    // 프롬프트 출력
    printf("[%s@%s:%s]%c ", user, hostname, directory_name, strcmp(user, "root") ? '$' : '#');
    fflush(stdout);
}
```

bash 쉘에서는 루트 하위 디렉토리의 경우 프롬프트 상에서 절대경로를, 홈 디렉토리의 경우 ~을, 그 외의 경우 절대경로상의 마지막 디렉토리명만 나타낸다. 이를 구현하기 위해 위와 같이 절대경로상에 포함되어있는 '/'의 개수를 파악, 각 케이스별로 그에 맞는 경로를 directory_name에 저장한다. 그 후엔 환경변수에 담겨있는 유저네임, 호스트네임을 담아 bash쉘의 기본 프롬프트 형식에 맞게 출력하는데, 사용자가 루트인 경우 프롬프트상 마지막 문자가 다르게 출력되므로 이를 같이 구현해주었다.

cd 명령어 구현

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "cd.h"

int cd(const char *path) {
    int exit_code = 0; // 종료 코드 초기화
    const char *oldpwd = getenv("PWD"); // 현재 디렉토리를 OLDPWD로 저장

    if (path == NULL || strcmp(path, "") == 0) { // 입력된 경로가 없는 경우
        path = getenv("HOME"); // 홈 디렉토리로 이동
    } else if (strcmp(path, "-") == 0) { // cd - 처리
        path = getenv("OLDPWD"); // 이전 디렉토리로 이동
        if (path == NULL) {
            fprintf(stderr, "cd: OLDPWD not set\n");
            exit_code = 1; // 오류 코드 설정
            return exit_code;
        }
        printf("%s\n", path); // 이동할 디렉토리 출력
    }

    // chdir로 디렉토리 변경
    if (chdir(path) != 0) {
        perror("cd"); // 오류 메시지 출력
        exit_code = 1; // 오류 코드 설정
        return exit_code; // 오류 발생 시 종료
    }

    // PWD와 OLDPWD 환경변수 업데이트
    setenv("OLDPWD", oldpwd, 1); // 이전 디렉토리를 OLDPWD로 설정
    char *cwd = getcwd(NULL, 0); // 현재 작업 디렉토리 가져오기
    setenv("PWD", cwd, 1); // PWD 환경변수 설정
    free(cwd); // 메모리 해제
    return exit_code; // 종료 코드 반환
}
```

우선 명령어 종료 시 반환할 exit_code를 선언한 뒤 0으로 초기화하고, 현재 디렉토리를 바꾸기 전 현재 디렉토리를 변수에 저장해둔다. 만약 사용자가 입력한 경로가 없다면 홈 디렉토리로, 경로로 '-' 를 입력한 경우 bash 쉘에서의 동작과 같이 이전 디렉토리로 이동할 수 있도록 예외처리를 한뒤에 chdir로 현재 디렉토리를 변경한 뒤 OLDPWD 환경변수를 미리 저장해 둔 이전 디렉토리로, PWD 환경변수를 현재 디렉토리로 변경한다.

process_command 함수 작성

process_command 함수는 입력받은 명령어를 실행하기 전 && || ; 등의 명령어 연산자를 처리하고 그에 맞게 단일 명령어들을 실행하도록 하는 함수로 그 로직은 아래와 같다.

```c
void process_command(const char *input) { // 입력문 처리 함수
    char *cmd_copy = strdup(input); // 입력 문자열 복사
    char *commands[256]; // 명령어를 저장할 배열
    char *operators[256]; // 연산자를 저장할 배열
    int cmd_index = 0, op_index = 0;

    for (int i = 0; cmd_copy[i] != '\0';) {

        // 명령어 추출
        commands[cmd_index++] = &cmd_copy[i];
        while (cmd_copy[i] != '\0' && cmd_copy[i] != ';' && cmd_copy[i] != '|' && cmd_copy[i] != '&') {
            i++;
        }
```

1) 명령어를 저장할 commands 배열의 0번 인덱스에 입력된 문자열의 0번째 문자 주소를 저장하고 입력된 문자열에서 명령어 연산자, 혹은 문자열의 끝에 닿을 때까지 문자열의 문자를 하나씩 검사한다.

```c
// 연산자 추출 및 저장
if ((cmd_copy[i] == ';' || cmd_copy[i] == '|' || cmd_copy[i] == '&') && commands[cmd_index - 1] != &cmd_copy[i]) {
    if (cmd_copy[i] == '|' && cmd_copy[i + 1] == '|') { // || 처리
        operators[op_index++] = "||"; // 연산자 저장
        cmd_copy[i] = '\0'; // 명령어 끝에 NULL 추가
        i += 2;
    } else if (cmd_copy[i] == '&' && cmd_copy[i + 1] == '&') { // && 처리
        operators[op_index++] = "&&";
        cmd_copy[i] = '\0';
        i += 2;
    } else if (cmd_copy[i] == '&' && cmd_copy[i + 1] == ';') { // 백그라운드(&;) 처리
        operators[op_index++] = ";";
        cmd_copy[i + 1] = '\0';
        i += 2;
    } else if (cmd_copy[i] == ';') { // ; 처리
        operators[op_index++] = ";";
        cmd_copy[i] = '\0';
        i++;
    } else if (cmd_copy[i] == '|') { // | 처리
        operators[op_index++] = "|";
        cmd_copy[i] = '\0';
        i++;
    } else { // 백그라운드(&) 처리
        i++;
    }
} else if (cmd_copy[i] == '\0') {
    break;
} else {
    fprintf(stderr, "Error: Invalid command\n");
    free(cmd_copy); // 복사한 문자열 메모리 해제
    return; // 오류 발생 시 종료
}
```

2) 특정된 문자는 operators에 순차적으로 저장하고, 연산자가 있던 위치에 '\0' 을 대입하며 commands의 0번째 명령어를 확정하고 끝마친다. 이때 만약 문자열 끝에 도달하여 특정된 문자가 없이 '\0'을 가리킨다면 루프를 종료한다.

3) i를 증가시켜 다음 문자의 주소를 commands에 뒤이어 저장하고 위 과정을 반복한다.

```
// 배열 종료
commands[cmd_index] = NULL;
operators[op_index] = NULL;
```

4) 모든 과정이 끝난 뒤에는 위와 같이 각 배열의 마지막 인덱스에 null을 입력하며 배열을 종료한다.

이렇게되면 commands에는 단일 명령어들이, operators에는 명령어 연산자들이 저장되고 이젠 이에 맞춰 명령어들을 실행해야 한다.

```c
// 마지막 종료코드를 저장할 변수
int exit_code = 0;

// 명령어 실행
for (int i = 0; commands[i] != NULL;) {
    // 파이프라인 처리
    if (operators[i] != NULL && strcmp(operators[i], "|") == 0) {
        char *pipeline_cmds[256];
        int pipeline_cmd_count = 0; // 파이프라인으로 연결된 명령어 수

        // 파이프라인 명령어 추출
        char *cmd_copy = strdup(commands[i]);
        pipeline_cmds[pipeline_cmd_count++] = cmd_copy;

        while (operators[i + pipeline_cmd_count - 1] != NULL && strcmp(operators[i + pipeline_cmd_count - 1], "|") == 0) {
            if (commands[i + pipeline_cmd_count] == NULL) {
                fprintf(stderr, "Error: Invalid command\n");
                for(int k=0; k<pipeline_cmd_count; k++) {
                    free(pipeline_cmds[k]);
                }
                free(cmd_copy);
                return;
            }
            cmd_copy = strdup(commands[i + pipeline_cmd_count]);
            pipeline_cmds[pipeline_cmd_count++] = cmd_copy;
        }
        pipeline_cmds[pipeline_cmd_count] = NULL; // 리스트 종료

        // 파이프라인 실행
        exit_code = execute_pipeline(pipeline_cmds, 0, STDIN_FILENO);

        // 파이프라인 실행 후 strdup된 메모리 해제
        for(int j = 0; j < pipeline_cmd_count; j++) {
            free(pipeline_cmds[j]);
        }

        // 메인 루프 인덱스 건너뛰기
        i += pipeline_cmd_count;
        continue;
```

우선 이전 명령어의 종료코드를 저장할 변수를 선언하고, 파이프라인을 처리할 로직을 구성해주었다. 만약 i번째 연산자가 '|' 라면..

1) 파이프라인으로 연결된 명령어들을 저장할 배열, 명령어의 수를 저장할 변수를 선언
2) i번째 명령어를 명령어 저장 배열에 저장
3) while 문을 돌며 다음 operators에 저장된 값이 '|' 이 아닐 때까지 해당 명령어에 대해 2) 반복. 이때 입력된 문자열이 (cmd1 | cmd2 |) 와 같은 형태라면 오류메세지 출력 후 strdup으로 할당된 메모리 free 및 종료
4) while 문이 끝나면 명령어들이 저장된 배열을 종료해주고 그 배열을 인자로 담아 후술할 파이프라인 실행 함수 호출
5) strdup된 메모리 해제 후 다음 명령어 실행을 위해 i += pipeline_cmd_count 후 for문 복귀

```
    } else if (i == 0 || operators[i-1] == NULL || strcmp(operators[i-1], ";") == 0) {
        exit_code = execute_command(commands[i], 0);

    } else if (strcmp(operators[i-1], "&&") == 0) {
        exit_code = (exit_code == 0) ? execute_command(commands[i], 0) : -1;

    } else if (strcmp(operators[i-1], "||") == 0) {
        exit_code = (exit_code == 0) ? -1 : execute_command(commands[i], 0);
    }
    i++;
}
free(cmd_copy); // 복사한 문자열 메모리 해제
```

남은 명령어들의 경우 operators[i - 1]과 exit_code에 따라 실행 여부가 달라지므로 위와 같이 그 케이스를 분류해 각각의 연산자에 맞는 조건을 걸어 후술할 단일 명령어 실행 함수를 통해 실행해주고, 메모리를 free 하며 함수를 끝마친다.

execute_command 함수 작성

```c
int execute_command(const char *command, const int is_forked) { // 명령어 실행 함수
    char *args[256]; // 명령어와 옵션을 저장할 배열
    char *cmd_copy = strdup(command); // 입력 문자열 복사
    cmd_copy[strcspn(cmd_copy, "\n")] = '\0'; // 개행 문자 제거
    int i = 0, exit_code = 1;

    // 백그라운드 실행 여부 확인
    int is_background = 0;
    if (cmd_copy[strlen(cmd_copy) - 1] == '&') { // 명령어가 &로 끝나는 경우
        is_background = 1;
        cmd_copy[strlen(cmd_copy) - 1] = '\0'; // & 제거
    }

    char *token = strtok(cmd_copy, " "); // 공백 기준으로 첫 번째 토큰 추출

    // 명령어와 옵션 분리 저장
    while (token != NULL) {
        args[i++] = token;
        token = strtok(NULL, " "); // 다음 토큰 추출
    }
    args[i] = NULL; // 마지막 인자는 NULL로 설정

    if (args[0] == NULL) {
        return exit_code;
    }

    pid_t pid = 0;

    if (is_background && !is_forked) {
        pid = fork(); // 자식 프로세스 생성
        if (pid > 0) { // 부모 프로세스인 경우
            fflush(stdout);
            printf("+ %s %d\n", args[0], pid); // 백그라운드 프로세스 PID 출력
            add_background_pid(pid);
            goto SKIP_CHILD;
        } else if (pid < 0) { // fork 실패
            perror("fork failed");
            free(cmd_copy);
            return exit_code;
        }
    }
```

execute_command 함수는 입력된 단일 명령어를 실행하는 함수로 먼저 명령어의 끝에 &가 붙어있는지를 검사함으로써 백그라운드 실행여부를 판단한 뒤 이를 변수에 저장해둔다. 이후 입력된 명령어에서 명령어 원문과 옵션을 분리해 저장하고, 입력된 명령어가 없는 경우 초기 1로 설정된 exit_code를 반환하며 종료한다. 그렇게 파싱이 끝나면 백그라운드 명령어이면서 이미 fork가 된 상태가 아닌 경우 fork를 통해 자식 프로세스를 생성하고, 자식 프로세스는 후술할 명령어 실행 코드를 순차적으로 실행하며, 부모 프로세스는 자식

프로세스의 PID를 출력하고 후술할 백그라운드 명령어 리스트에 자식 프로세스의 pid를 추가한 뒤 리턴한다.

```c
    // 쉘 내부 명령어 처리
    if (strcmp(args[0], "pwd") == 0) { // pwd 처리
        exit_code = pwd(); // pwd 명령어 실행
        free(cmd_copy);
        if (is_background) {
            exit(exit_code);
        } else {
            return exit_code;
        }
    } else if (strcmp(args[0], "cd") == 0) { // cd 처리
        exit_code = cd(args[1]);
        free(cmd_copy);
        if (is_background) {
            exit(exit_code);
        } else {
            return exit_code;
        }
    }

    if (!is_background && !is_forked) {
        pid = fork(); // 자식 프로세스 생성
    }

    if (pid == 0) { // 자식 프로세스
        execvp(args[0], args);
        perror("exec failed");
        free(cmd_copy);
        exit(1);
    } else if (pid > 0) { // 부모 프로세스
        int status; // 자식 프로세스 종료 상태를 저장할 변수
        waitpid(pid, &status, 0); // 자식 프로세스 종료 대기
        if (WIFEXITED(status)) { // 자식 프로세스가 정상 종료되었는지 확인
            exit_code = WEXITSTATUS(status); // 종료 코드 저장
            if (exit_code != 0) { // 종료 코드가 0이 아닌 경우
                printf("Command failed with exit code: %d\n", exit_code);
            }
        }
    } else { // fork 실패
        perror("fork failed");
    }

    SKIP_CHILD:
    free(cmd_copy);
    return exit_code; // 종료 코드 반환
}
```

백그라운드 프로세스가 아닌이상 쉘 내부 명령어는 fork, exec의 과정을 거치지 않으므로 우선적으로 검사, 실행해주고 외부 명령어의 경우 fork, exec의 과정을 거치므로 백그라운드 프로세스가 아니면서 이미 fork된 상태가 아닌 경우 fork를 통해 자식 프로세스 생성, 자식 프로세스는 명령어 실행, 부모 프로세스는 자식 프로세스가 종료될 때까지 기다렸다가 종료 코드를 받아 return한다.

execute_pipeline 함수 작성

```c
int execute_pipeline(char *commands[], int index, int input_fd) {
    int pipefd[2];
    if (commands[index + 1] != NULL) { // 다음 명령어가 있다면 파이프 생성
        if (pipe(pipefd) == -1) {
            perror("pipe failed");
            exit(1);
        }
    }

    pid_t pid = fork();
    if (pid == 0) { // 자식 프로세스
        if (input_fd != STDIN_FILENO) {
            dup2(input_fd, STDIN_FILENO); // 이전 명령어의 출력 연결
            close(input_fd); // 이전 파이프의 읽기 끝 닫기
        }
        if (commands[index + 1] != NULL) {
            dup2(pipefd[1], STDOUT_FILENO); // 현재 명령어의 출력 연결
            close(pipefd[0]);
            close(pipefd[1]);
        }
        execute_command(commands[index], 1); // 명령어 실행
    } else if (pid > 0) { // 부모 프로세스
        wait(NULL); // 자식 프로세스 종료 대기
        if (commands[index + 1] != NULL) {
            close(pipefd[1]); // 현재 파이프의 쓰기 끝 닫기
            execute_pipeline(commands, index + 1, pipefd[0]); // 다음 명령어 처리
        }
    } else {
        perror("fork failed");
    }
    return 0;
}
```

process_command에서 배열의 형태로 전달해준 명령어들을 pipe를 통해 이어 실행하는 함수이다. 파이프라인의 실행은 재귀 호출로 이루어지는데 첫 호출은 execute_pipeline(commands, 0, STDIN_FILENO); 로 이뤄진다. 이후엔 위에서 볼 수 있듯 마지막 인자에 현재 파이프의 읽기 끝을 담아 다시 호출을 하는 방식이다. 우선 함수가 실행되면 다음 명령어가 있는지를 따져보고 다음 명령어가 있는경우 파이프를 생성한다. 그후 자식 프로세스를 생성해 만약 이 명령어가 파이프라인의 첫 명령어가 아닌 경우 이전 명령어의 출력을 연결하고 읽기 끝을 닫아준다. 그리고 다음 명령어가 있는 경우 현재 명령어의 출력을 읽을 수 있도록 연결해주고, 자식 프로세스의 읽기 끝과 쓰기 끝을 닫은 뒤

execute_command 함수를 호출해 명령어를 실행한다. 부모 프로세스의 경우 자식 프로세스의 종료를 기다렸다가 다음 명령어가 있는 경우 쓰기 끝을 닫고, 다음 명령어로 넘어갈 수 있도록 현재 파이프의 읽기 끝을 담아 재귀호출한다.

백그라운드 프로세스 관리 함수 작성

```c
#define MAX_BACKGROUND_PROCESSES 256

// 백그라운드 프로세스 리스트
pid_t BACKGROUND_PIDS[MAX_BACKGROUND_PROCESSES];
int BACKGROUND_COUNT = 0;

// 백그라운드 PID 추가 함수
void add_background_pid(pid_t pid) {
    if (BACKGROUND_COUNT < MAX_BACKGROUND_PROCESSES) {
        BACKGROUND_PIDS[BACKGROUND_COUNT++] = pid;
    }
}

// 백그라운드 PID 제거 함수
void remove_background_pid(pid_t pid) {
    for (int i = 0; i < BACKGROUND_COUNT; i++) {
        if (BACKGROUND_PIDS[i] == pid) {
            // PID를 리스트에서 제거
            for (int j = i; j < BACKGROUND_COUNT - 1; j++) {
                BACKGROUND_PIDS[j] = BACKGROUND_PIDS[j + 1];
            }
            BACKGROUND_COUNT--;
            break;
        }
    }
}
```

위 함수들은 백그라운드 프로세스를 리스트화하여 관리할 수 있도록 해준다. 이때 그 리스트는 전역변수로 후술할 시그널 핸들러에서 활용될 것이다. 먼저 백그라운드 pid들을 담을 배열을 선언했고, pid를 인자로 담아 add_background_pid 함수를 호출하면 백그라운드 pid 배열에 순차적으로 추가되도록 하였다. 그리고 pid를 인자로 담아 remove_background_pid 함수를 호출하면 pid 리스트에서 타겟 pid를 찾은 뒤 그 뒤의 요소들을 한 칸 앞으로 당겨오는 방식으로 pid 리스트 내의 타겟 pid를 제거하는 기능을 구현하였다.

```c
// SIGCHLD 시그널 핸들러
void sigchld_handler(int signo) {
    (void)signo; // 시그널 번호 사용하지 않음
    pid_t pid;
    int status;

    // 백그라운드 프로세스 중 종료된 자식 프로세스 처리
    for (int i = 0; i < BACKGROUND_COUNT; i++) {
        pid = waitpid(BACKGROUND_PIDS[i], &status, WNOHANG);
        if (pid > 0) { // 종료된 프로세스가 있는 경우
            printf("- done %d\n", pid);
            remove_background_pid(pid); // 리스트에서 제거
        }
    }
}

// SIGCHLD 핸들러 설정 함수
void setup_signal_handler(void) {
    struct sigaction sa;
    sa.sa_handler = sigchld_handler;
    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP; // 시그널 처리 중 중단된 시스템 호출 재시작
    sigaction(SIGCHLD, &sa, NULL);
}
```

백그라운드 pid 리스트를 토대로 위와 같이 코드를 구성해 좀비 프로세스가 발생되지 않도록 계속해서 백그라운드 프로세스의 종료 여부를 확인해 종료 상태가 회수되도록 하였다.

main 함수 작성

```c
#include <stdio.h>
#include <string.h>
#include "pwd.h"
#include "exec_cmd.h"
#include "print_prompt.h"
#include "handle_sig.h"

int main(void) {
    char command[256]; // 명령어를 저장할 배열

    setup_signal_handler(); // SIGCHLD 핸들러 설정
    initialize_cwd(); // 현재 디렉토리를 사용자의 홈 디렉토리로 초기화

    while (1) {
        print_prompt(); // 프롬프트 출력

        fgets(command, sizeof(command), stdin); // 명령어 입력
        command[strcspn(command, "\n")] = '\0'; // 개행 문자 제거

        if (strcmp(command, "exit") == 0) { // exit 명령어 처리
            break; // 루프 종료
        } else if (strcmp(command, "") == 0) {
            continue;
        }
        process_command(command); // 명령어 처리
    }
    return 0;
}
```

앞서 작성한 함수들을 바탕으로 main을 작성했다. 무한 루프를 돌며 프롬프트를 출력하고 fgets를 이용해 명령어를 입력받는다. 만약 입력값이 exit이라면 break 하고 입력값이 없는 경우 continue하며 그렇지 않은 경우 입력값을 process_command에 전달해 명령을 수행할 수 있도록 하였다.