CyKor Week2

작성자: 사이버국방학과 2024350213 김진모

1. 대략적인 구현 과정

대략적인 구현 과정에 대해 먼저 설명하려고 한다.

-쉘 명령어와 사용자 인터페이스 및 exit 명령어 구현 (1, 2, 7번 조건)

우선 while(1)을 통해 쉘 명령어가 입력에 따라 계속해서 나타나도록 했다. 기본적으로 명령어가 입력으로 들어오게 되면 이를 처리할 함수가 따로 있어야 한다고 판단이 되어 execute_command라는 함수를 통해 명령어를 처리하도록 했다. 이때 exit 명령어가 입력으로 들어올 경우 효율성을 위해 따로 cmd가 매개변수로 넘어가지 않고, break를 통해 while 루프가 종료되도록 했다. while문 내부에서는 bash의 실제 사용자 인터페이스와 동일한 형태를 띄도록 하기 위해 bash에서 출력되는 형태와 같이 인터페이스가 출력되도록 했다. 이때 bash와 같이 getcwd 함수를 이용하여 현재 디렉토리를 drc에 저장하고, 이도 같이 출력되도록 했다.

execute_command 함수 내부에서는 입력 받은 명령어의 개행을 제거하고 공백을 기준으로 잘라 arg[] 배열에 저장했다. arg[0]의 값에 따라 각각의 명령어가 실행되고, 인수를 필요로 하는 명령어는 arg[1]을 인수로 사용하도록 했다.

-pwd, cd, ls 명령어 및 상대경로 구현 (3번 조건)

while문이 1회 실행될 때마다 getcwd 함수를 이용하여 현재 디렉토리를 drc에 저장하도록 했다. 이때 arg[0]이 각각 pwd, cd, ls와 같다면 cmd_pwd, cmd_cd, cmd_ls 함수를 실행하도록 한 뒤 각각의 함수에 pwd, cd, ls 함수의 동작을 구현했다.

cmd_pwd 함수는 drc의 값을 출력하도록 했다.

cmd_cd 함수는 cd 명령어 뒤에 입력되는 문자열을 dst라는 인수로 받아 chdir 함수를 통해 dst 디렉토리에 접근하도록 했다. 이 chdir 함수가 -1을 리턴하면 에러가 발생한 것인데, dst라는 디렉토리가 없을 경우에 해당하기 때문에 이 경우 bash와 같은 에러 메시지를 출력하도록 했다. dst 디렉토리가 존재한다면 dst로 이동하고, getcwd 함수를 통해 다시 전역변수인 drc에 현재 디렉토리의 주소를 저장하도록 했다.

cmd_ls 함수는 DIR 포인터 dp를 활용했다. opendir 함수를 사용하여 현재 디렉토리를 열고, dirent 구조체 포인터를 통해 현재 디렉토리의 파일들의 이름인 dirp→d_name을 디렉토리의 모든 파일에 대해 출력하도록 했다. 이후 closedir 함수를 통해 다시 DIR 포인터를 닫도록 했다.

-다중 명령어(; , &&, ||) 구현 (5번 조건)

다중 명령어는 ;, &&, ||가 있는데, 이들은 각각 한 줄에서 여러 명령어 실행, 앞의 명령어가 성공할 때만 뒤의 명령어 실행, 앞의 명령어가 실패할 때만 뒤의 명령어 실행이라는 동작을 담당한다. execute_command 함수는 pwd, cd, ls 등의 명령어를 직접 실행하는 함수이므로 seperate_command 함수를 새로 만들고, 이 함수에서 입력받은 명령어를 ;, &&, || 기준으로 파싱한 다음 다시 seperate_command 함수를 적절하게 사용하도록 하여 구현했다. 기존에는 void형이었던 execute command 함수와 각각의 cmd_ 함수들을 int형으로 바꾸고, 각각 1에 해당하는 CMD_PASS와 0에 해당하는 CMD_FAIL 중 하나를 return하게 하여 앞 명령어의 성공 여부를 알 수 있도록 했다.(과목29 수업에서 영감을 받았다.) seperate_command 함수는 결국 다중 명령어 연산자가 하나도 들어가 있지 않음이 보장되는 명령어에 대하여 execute_command 함수를 통해 명령어를 실행시키도록 했다.

이때 &&, ||, ;라는 각 연산자의 우선순위를 고려하여 if-elseif문이 작동하도록 하였다. 이때 if-elseif 문은 strstr함수를 사용하여 각각의 연산자가 포함되어 있는 지를 통해 나누었다. 각각의 연산자를 기준으로 파싱하고, 앞에서부터 CMD_PASS와 CMD_FAIL 여부를 고려하여 &&는 실패한 명령어가 나오면 이후 명령어를 실행하지 않고, ||는 성공한 명령어가 나오면 이후 명령어를 실행하지 않도록 했다. ;를 기준으로 파싱한 명령어들은 순서대로 실행되도록 했다.

-파이프라인 및 멀티 파이프라인(|) 구현 (4번 조건)

파이프라인 명령어는 다중 명령어와 비교했을 때에 우선순위가 가장 높기 때문에 이 점을 고려하여 seperate_command 함수에서 else if문을 통해 감지하도록 하였다. 파이프라인을 기준으로 파싱하는 과정까지 동일하게 구현했고, 이후 fork 함수를 통해 자식 프로세스를 만들어주었다. 자식 프로세스인 경우 별도의 pipeline 함수에 진입하도록 했고, 부모 프로세스인 경우 waitpid 함수를 사용하여 자식 프로세스에서 exit을 만나도 전체 프로세스가 종료되지 않고 자식 프로세스가 종료될 때까지 기다리도록 했다.

pipeline 함수에서는 재귀함수를 통해 앞에서부터 pipeline이 실행되도록 하였다. |를 기준으로 파싱한 문자열을 배열에 저장한 것, 현재 실행하고 있는 명령어의 위치, 전체 명령어의 개수를 입력으로 받았다. 이때 n번째 실행이라면 재귀가 종료되어야 하기 때문에 execute_command 함수를 통해 마지막 명령어가 실행되고, exit을 통해 프로세스가 종료되도록 하였다. 이 경우가 아니라면 자식 프로세스의 출력을 pipe의 출력과 연결하고, 부모 프로세스의 입력을 pipe의 입력과 연결하여 자식 프로세스에서는 i번째 명령어, 부모 프로세스에서는 i+1번째 명령어를 실행하도록 하였다.

-백그라운드 실행(&) 구현 (6번 조건)

백그라운드 실행은 다중 명령어와 비교했을 때에 우선순위가 가장 낮기 때문에 이 점을 고려하여 seperate_command 함수에서 가장 위의 if문을 통해 감지하도록 하였다. &를 기준으로 파싱했고, 이때 자식 프로세스를 만들고 자식 프로세스에서 다시 seperate_command 함수를 통해 명령어가 실행되도록 했다. 이때 명령어가 종료되면

exit을 통해 프로세스가 종료되도록 했으며, 부모 프로세스에서는 곧바로 CMD_PASS를 리턴하여 다음 while 시행이 돌아가도록 했다.

-추가적인 명령어 구현 (8번 조건)

execvp 시스템콜을 사용하여 기타 명령어가 입력되었을 때 이 명령어들이 제대로 실행되도록 했다. 이때, 띄어쓰기를 기준으로 파싱한 각각의 인자들을 배열로 만들어 execvp 명령어를 사용했다. 부모 프로세스에서 이를 사용하면 프로세스가 대체되며 쉘이 종료되므로 자식 프로세스를 만들고 자식 프로세스에서 실행되도록 했다. 부모 프로세스에서는 waitpid를 통해 프로세스가 종료되기 전까지 기다리고, 매크로를 통해 자식 프로세스가 제대로 종료되었다면 CMD_PASS를 리턴하도록 했다.

---

2. 쉘 명령어와 사용자 인터페이스 구현

```c
int main() {
    char cmd[CMD_MAX_LEN];

    while (1){
        getcwd(drc, sizeof(drc));   // 현재 디렉토리 값을 drc에 저장함.
        printf("hissen@HISSEN:%s$ ", drc);

        if (fgets(cmd, sizeof(cmd), stdin) == NULL) {   // 입력 오류가 발생한 경우우
            printf("Error detected.\n");
            break;
        }

        cmd[strcspn(cmd, "\n")] = 0;

        if (strcmp(cmd, "exit") == 0) break;  // exit 명령어가 들어온 경우우
        seperate_command(cmd);

    }

    return 0;
```

기본적으로 쉘 명령어가 출력되도록 하는 main함수이다. 우선 getcwd 함수를 통해 매 시행마다 drc라는 전역변수에 현재 디렉토리 값을 저장하도록 했다. 현재 디렉토리는 모든 함수에서 접근이 가능해야 모든 명령어가 현재 디렉토리에 반영이 될 수 있기 때문에 전역변수로 선언했다. 이후 실제 bash와 같이 사용자 인터페이스가 출력되도록 했고, drc가 뒤에 출력되도록 하여 현재 디렉토리가 같이 출력되도록 했다. 이후 fgets 함수를 통해 한 줄을 입력받도록 했다. 이때 입력 오류가 발생한 경우 break를 통해 쉘 스크립트가 종료되도록 했다. strcspn을 통해 개행문자의 위치를 찾고, cmd에서 해당 위치의 값을 0으로 바꿔주어 개행을 제거했다. 이후 cmd에 exit이 입력되었다면 break를 통해 쉘이 종료되도록 했다. 이후에는 입력된 명령어가 실행되어야 하므로, seperate_command 함수에 cmd를 매개변수로 넣어 실행되도록 했다.

---

3. pwd, cd, ls 및 상대경로 구현

seperate_command 함수의 동작에 대해서는 추후에 설명한다.

```c
int execute_command(char *cmd) {

    char *arg[ARG_MAX_LEN] = {0,};
    int argn = 0;
    char *token;
    token = strtok(cmd, " ");
    argn = 0;
    //sscanf(cmd, "%s", kw);
    while (token != NULL && argn < ARG_MAX_LEN){
        arg[argn++] = token;
        token = strtok(NULL, " ");
    }
    arg[argn] = NULL;

    if (strcmp(arg[0], "help") == 0) {
        printf("Available commands: help, exit\n");
    }
    else if (strcmp(arg[0], "pwd") == 0){
        return cmd_pwd();
    }
    else if (strcmp(arg[0], "cd") == 0){
        return cmd_cd(arg[1]);
    }
    else if (strcmp(arg[0], "ls") == 0){
        return cmd_ls();
    }
    else {
        char *argv[ARG_MAX_LEN];
        for (int i = 0; i < argn; i++){
            argv[i] = arg[i];
        }
        argv[argn] = NULL;

        pid_t pid = fork();
        if (pid == 0){
            execvp(arg[0], argv);
            exit(1);
        }
        else {
            int stat;
            waitpid(pid, &stat, 0);
            return (WIFEXITED(stat) && WEXITSTATUS(stat) == 0) ? CMD_PASS : CMD_FAIL;
        }
    }
}
```

execute_command 함수는 명령어를 입력으로 받고, 띄어쓰기를 기준으로 파싱한 결과를 arg에 저장한다. argn은 파싱한 인자들의 개수에 해당하는 변수이다. strtok 함수를 사용했고, 이때 strtok 함수를 반복해서 NULL과 공백을 인자로 넣고 호출하면 공백을 기준으로 파싱한 값들이 token에 계속해서 저장되는 점을 활용하여 arg의 index를 1씩 늘려가면서 파싱한 결과를 저장했다. 이후 각각 help, pwd, cd, ls 명령어에 대하여 간단하게 구현했는데, arg[0]이 각각 help, pwd, cd, ls이면 각 if-else case에 진입하도록 하였다. 각 case에 진입하게 되면 cmd_pwd, cmd_cd, cmd_ls 함수를 호출한다.

```c
int cmd_pwd(void){
    printf("%s\n", drc);
    return CMD_PASS;
}

int cmd_cd(char *dst){
    //if (strcmp(dst, "/") == 0) {chdir("/");return;}
    if (strcmp(dst, "~") == 0) {chdir("//home/hissen"); getcwd(drc, sizeof(drc)); return CMD_PASS;}
    // if (strcmp(dst, "..") == 0){

    // }
    //char fdst[DRC_MAX_LEN];
    //strcat(drc, "/");
    //strcat(drc, dst);
    if (chdir(dst) == -1) {
        printf("-bash: cd: %s: No such file or directory\n", dst);
        return CMD_FAIL;
    }
    else {
        //printf("Successfully moved to %s\n", dst);
        getcwd(drc, sizeof(drc));
        return CMD_PASS;
    }
}
```

cmd_pwd 함수는 간단하게 drc, 즉 현재 디렉토리의 값을 출력한 뒤 CMD_PASS를 리턴하도록 했다.

cmd_cd 함수는 상대 경로인 ~를 입력 받았을 때만 case를 별도로 구분하여 /home/hissen으로 이동한 뒤, drc에 현재 디렉토리의 값을 저장하고 CMD_PASS를 리턴하도록 했다. 이 경우가 아닌 경우에는 매개변수로 받은 dst라는 경로로 chdir 함수를 통해 이동하고, 이때 -1이 리턴되어 에러가 발생하면 해당 디렉토리가 없다는 문구를 출력하도록 했다. 에러가 아닌 경우에는 getcwd를 통해 drc에 디렉토리를 다시 저장하고, CMD_PASS를 리턴하도록 했다.

```c
int cmd_ls(void){
    DIR *dp;
    struct dirent *dirp;

    dp = opendir(".");
    while (dirp = readdir(dp)){
        printf("%s\n", dirp->d_name);
    }
    printf("\n");
    closedir(dp);
    return CMD_PASS;
}
```

cmd_ls 함수는 DIR 포인터과 dirent 구조체를 이용해서 구현했다. dp에 현재 디렉토리를 열고, readdir 함수를 사용하여 dirent 구조체 포인터가 각 파일을 모두 한번씩 가리키도록 while문을 썼다. 이후 각각의 dirent 구조체의 인자인 d_name을 줄마다 출력하도록 하여 ls를 구현했다. 이후 다시 closefir 함수를 통해 DIR 포인터를 닫아주었다.

---

4. 다중 명령어(; , && , ||) 구현

```
#define CMD_PASS 1
#define CMD_FAIL 0

char drc[DRC_MAX_LEN] = {0,};
//char kw[KW_MAX_LEN] = {0,};

int execute_command(char *cmd);
int seperate_command(char *cmd);
void pipeline(char **cmds, int i, int n);
int cmd_pwd(void);
int cmd_cd(char *dst);
int cmd_ls(void);
//int cmd_echo(void);
```

다중 명령어를 구현하기 위해 pwd, cd, ls가 각각 성공했는 지 실패했는 지 여부를 알아야 했고, 이 때문에 CMD_PASS와 CMD_FAIL을 각각 1과 0으로 define해주었다. 각각의 명령어가 실행 결과로 PASS 또는 FAIL을 리턴하도록 했다. 이후 void형이었던 pwd, cd, ls 함수를 모두 int 형으로 바꿔주었다. execute_command 함수 역시 int형으로 바꾸어서 각각의 명령어가 리턴하는 값을 execute_command 함수가 리턴하도록 바꾸었다.

```c
int seperate_command(char *cmd){
    if (strstr(cmd, "&&") == NULL && strstr(cmd, "&")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, "&");
        while (tokens != NULL && splitn < ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, "&");
        }
        split[splitn] = NULL;
        if (fork() == 0){
            seperate_command(split[0]);
            exit(0);
        }
        return CMD_PASS;
    }
    if (strstr(cmd, ";")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, ";");
        while (tokens != NULL && splitn < ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, ";");
        }
        split[splitn] = NULL;
        for (i = 0; i<splitn; i++){
            seperate_command(split[i]);
        }
        return CMD_PASS;
    }
```

seperate_command 함수의 구조이다. 다중 명령어 연산자 ;, &&, ||의 우선순위를 고려하여 if 문을 짜 주었고, 이때 strstr함수를 통해 각각의 연산자가 입력받은 명령어에 포함되어 있는 지 확인해주었다. 먼저 ; case를 나누었는데, 기존에 짰던 파싱 코드를 활용하여 ;를 기준으로 파싱한 다음, 배열에 저장된 각각의 명령어를 다시 seperate_command 함수의 매개변수로 넣고 호출했다. 이는 ;를 기준으로 파싱했을 때에 &&, || 등 우선순위가 더 높은 연산자들이 포함되어 있을 수 있기 때문에 execute_command가 아닌 seperate_command 함수를 호출했다.

```
else if (strstr(cmd, "||")){
    char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
    tokens = strtok(cmd, "||");
    while (tokens != NULL && ARG_MAX_LEN) {
        split[splitn++] = tokens;
        tokens = strtok(NULL, "||");
    }
    split[splitn] = NULL;
    for (i = 0; i<splitn; i++){
        if (seperate_command(split[i]) == CMD_PASS) {return CMD_FAIL; break;}
        else continue;
    }
    if (i == splitn - 1) return CMD_PASS;
}
else if (strstr(cmd, "&&")){
    char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
    tokens = strtok(cmd, "&&");
    while (tokens != NULL && ARG_MAX_LEN) {
        split[splitn++] = tokens;
        tokens = strtok(NULL, "&&");
    }
    split[splitn] = NULL;
    for (i = 0;i<splitn;i++){
        if (execute_command(split[i]) == CMD_FAIL) {return CMD_FAIL; break;}
        else continue;
    }
    if (i == splitn - 1) return CMD_PASS;
}
```

이후에 ||, && 연산자에 대해서도 같은 방식으로 코드를 구현했다. 파싱을 진행한 다음, ||의 경우에는 반복문을 돌면서 성공한 명령어가 발생한다면 FAIL을 리턴하고 반복문이 종료되도록 했고, &&의 경우에는 실패한 명령어가 발생한다면 FAIL을 리턴하고 반복문이 종료되도록 했다. 이때 &&는 우선순위가 가장 높기 때문에 execute_command 함수를 호출했다.

```
    else return execute_command(cmd);
}
```

다중 명령어 연산자 중 어떤 것도 포함되어 있지 않은 경우에 execute_command 함수를 호출하도록 했다.

---

5. 파이프라인 및 멀티 파이프라인(|) 구현

```c
else if (strstr(cmd, "|")){
    char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
    tokens = strtok(cmd, "|");
    while (tokens != NULL && ARG_MAX_LEN) {
        split[splitn++] = tokens;
        tokens = strtok(NULL, "|");
    }
    split[splitn] = NULL;
    pid_t pid = fork();
    if (pid == 0){
        pipeline(split, 0, splitn);
    }
    else {
        int stat;
        waitpid(pid, &stat, 0);
    }
}
```

파이프라인은 우선순위가 가장 높기 때문에 가장 아래의 else if 문에 구현했다. 이때 파싱을 진행하고, 자식 프로세스를 만들어서 pipeline함수를 통해 지귀적으로 구현하였다.

```c
void pipeline(char **cmds, int i, int n){
    if (i == n - 1) {execute_command(cmds[i]); exit(0);}
    else {
        int fd[2];
        if (pipe(fd) == -1) {
            fprintf(stderr, "pipe error: %s\n", strerror(errno));
            exit(0);
        }
        pid_t pid = fork();
        if (pid == 0) {
            dup2(fd[1], 1);
            close(fd[0]);
            close(fd[1]);
            execute_command(cmds[i]);
            exit(0);
        }
        else {
            dup2(fd[0], 0);
            close(fd[0]);
            close(fd[1]);
            pipeline(cmds, i + 1, n);
        }
    }
}
```

pipeline 함수에서는 위에서 설명한 바와 같이 우선 |로 구분된 것 중 가장 마지막 명령어일 경우 execute_command를 통해 실행하고 종료되도록 했다. 그 외의 경우에는 자식 프로세스를 생성해서, 자식 프로세스의 출력을 pipe의 출력과 연결하고 부모 프로세스인 경우에는 입력을 pipe의 입력과 연결하여 자식 프로세스에서는 execute_command를 통해 명령어를 실행하고, 이 출력 결과를 부모 프로세스가 입력으로 받아 다시 pipline 함수를 다음 index로 호출하여 재귀적으로 이전 명령어의 출력을 다음 명령어가 입력으로 받도록 구현했다.

6. 백그라운드 실행(&) 구현

```
int seperate_command(char *cmd){
    if (strstr(cmd, "&&") == NULL && strstr(cmd, "&")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, "&");
        while (tokens != NULL && splitn < ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, "&");
        }
        split[splitn] = NULL;
        if (fork() == 0){
            seperate_command(split[0]);
            exit(0);
        }
        return CMD_PASS;
    }
```

백그라운드 실행은 가장 우선순위가 낮기 때문에, 가장 위의 if문에서 case를 생각해주었다. 이때 &&가 포함되지 않았으면서 &가 포함된 경우이기 때문에 이를 고려했고, 파싱을 진행했다. 파싱을 한 이후에는 자식 프로세스를 생성하여 자식 프로세스에서는 seperate_command를 호출하여 다시 명령어를 실행하고 프로세스를 종료하도록 했다. 부모 프로세스에서는 이를 기다리지 않고 PASS를 리턴하여 백그라운드에서 프로세스가 실행될 수 있도록 했다.

7. 추가적인 명령어 구현

```
    else {
        char *argv[ARG_MAX_LEN];
        for (int i = 0; i < argn; i++){
            argv[i] = arg[i];
        }
        argv[argn] = NULL;

        pid_t pid = fork();
        if (pid == 0){
            execvp(arg[0], argv);
            exit(1);
        }
        else {
            int stat;
            waitpid(pid, &stat, 0);
            return (WIFEXITED(stat) && WEXITSTATUS(stat) == 0) ? CMD_PASS : CMD_FAIL;
        }
    }
}
```

추가적으로 다른 명령어들을 execvp함수를 이용하여 구현했다. 이때 자식 프로세스를 사용하지 않으면 프로세스가 대체되어 프로그램이 종료되는 문제가 발생했고, 따라서 자식 프로세스 안에서 execvp함수를 실행했다. 부모 프로세스에서는 waitpid를 사용하여 기다린 다음 매크로를 사용하여 성공적으로 실행되었을 때에 PASS를 리턴하도록 했다.

---

8. 전체 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <dirent.h>
#include <errno.h>

#define CMD_MAX_LEN 100
#define DRC_MAX_LEN 200
#define MAX_LEN 1000
#define ARG_MAX_LEN 20
#define CMD_PASS 1
#define CMD_FAIL 0
```

```
char drc[DRC_MAX_LEN] = {0,};
//char kw[KW_MAX_LEN] = {0,};

int execute_command(char *cmd);
int seperate_command(char *cmd);
void pipeline(char **cmds, int i, int n);
int cmd_pwd(void);
int cmd_cd(char *dst);
int cmd_ls(void);
//int cmd_echo(void);
void cmd_exit(void);

int main() {
    char cmd[CMD_MAX_LEN];

    while (1){
        getcwd(drc, sizeof(drc));   // 현재 디렉토리 값을 drc에 저장함.
        printf("hissen@HISSEN:%s$ ", drc);

        if (fgets(cmd, sizeof(cmd), stdin) == NULL) {   // 입력 오류가 발생한 경우우
            printf("Error detected.\n");
            break;
        }

        cmd[strcspn(cmd, "\n")] = 0;

        if (strcmp(cmd, "exit") == 0) break;  // exit 명령어가 들어온 경우우
        seperate_command(cmd);

    }

    return 0;
}

int execute_command(char *cmd) {

    char *arg[ARG_MAX_LEN] = {0,};
    int argn = 0;
```

```
    char *token;
    token = strtok(cmd, " ");
    argn = 0;
    //sscanf(cmd, "%s", kw);
    while (token != NULL && argn < ARG_MAX_LEN){
        arg[argn++] = token;
        token = strtok(NULL, " ");
    }
    arg[argn] = NULL;

    if (strcmp(arg[0], "help") == 0) {
        printf("Available commands: help, exit\n");
    }
    else if (strcmp(arg[0], "pwd") == 0){
        return cmd_pwd();
    }
    else if (strcmp(arg[0], "cd") == 0){
        return cmd_cd(arg[1]);
    }
    else if (strcmp(arg[0], "ls") == 0){
        return cmd_ls();
    }
    else {
        char *argv[ARG_MAX_LEN];
        for (int i = 0; i < argn; i++){
            argv[i] = arg[i];
        }
        argv[argn] = NULL;

        pid_t pid = fork();
        if (pid == 0){
            execvp(arg[0], argv);
            exit(1);
        }
        else {
            int stat;
            waitpid(pid, &stat, 0);
            return (WIFEXITED(stat) && WEXITSTATUS(stat) == 0) ? CMD_PASS
```

```
        }
    }
}

int seperate_command(char *cmd){
    if (strstr(cmd, "&&") == NULL && strstr(cmd, "&")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, "&");
        while (tokens != NULL && splitn < ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, "&");
        }
        split[splitn] = NULL;
        if (fork() == 0){
            seperate_command(split[0]);
            exit(0);
        }
        return CMD_PASS;
    }
    if (strstr(cmd, ";")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, ";");
        while (tokens != NULL && splitn < ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, ";");
        }
        split[splitn] = NULL;
        for (i = 0; i<splitn; i++){
            seperate_command(split[i]);
        }
        return CMD_PASS;
    }
    else if (strstr(cmd, "||")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, "||");
        while (tokens != NULL && ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, "||");
```

```
        }
        split[splitn] = NULL;
        for (i = 0; i<splitn; i++){
            if (seperate_command(split[i]) == CMD_PASS) {return CMD_FAIL; bre
            else continue;
        }
        if (i == splitn - 1) return CMD_PASS;
    }
    else if (strstr(cmd, "&&")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, "&&");
        while (tokens != NULL && ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, "&&");
        }
        split[splitn] = NULL;
        for (i = 0;i<splitn;i++){
            if (execute_command(split[i]) == CMD_FAIL) {return CMD_FAIL; brea
            else continue;
        }
        if (i == splitn - 1) return CMD_PASS;
    }
    else if (strstr(cmd, "|")){
        char *split[ARG_MAX_LEN] = {0,}; int splitn = 0; char *tokens; int i;
        tokens = strtok(cmd, "|");
        while (tokens != NULL && ARG_MAX_LEN) {
            split[splitn++] = tokens;
            tokens = strtok(NULL, "|");
        }
        split[splitn] = NULL;
        pid_t pid = fork();
        if (pid == 0){
            pipeline(split, 0, splitn);
        }
        else {
            int stat;
            waitpid(pid, &stat, 0);
        }
```

```
    }

    else return execute_command(cmd);
}

void pipeline(char **cmds, int i, int n){
    if (i == n - 1) {execute_command(cmds[i]); exit(0);}
    else {
        int fd[2];
        if (pipe(fd) == -1) {
            fprintf(stderr, "pipe error: %s\n", strerror(errno));
            exit(0);
        }
        pid_t pid = fork();
        if (pid == 0) {
            dup2(fd[1], 1);
            close(fd[0]);
            close(fd[1]);
            execute_command(cmds[i]);
            exit(0);
        }
        else {
            dup2(fd[0], 0);
            close(fd[0]);
            close(fd[1]);
            pipeline(cmds, i + 1, n);
        }
    }
}

int cmd_pwd(void){
    printf("%s\n", drc);
    return CMD_PASS;
}

int cmd_cd(char *dst){
    //if (strcmp(dst, "/") == 0) {chdir("/");return;}
    if (strcmp(dst, "~") == 0) {chdir("//home/hissen"); getcwd(drc, sizeof(drc
```

```
    // if (strcmp(dst, "..") == 0){

    // }
    //char fdst[DRC_MAX_LEN];
    //strcat(drc, "/");
    //strcat(drc, dst);
    if (chdir(dst) == -1) {
        printf("-bash: cd: %s: No such file or directory\n", dst);
        return CMD_FAIL;
    }
    else {
        //printf("Successfully moved to %s\n", dst);
        getcwd(drc, sizeof(drc));
        return CMD_PASS;
    }
}

int cmd_ls(void){
    DIR *dp;
    struct dirent *dirp;

    dp = opendir(".");
    while (dirp = readdir(dp)){
        printf("%s\n", dirp->d_name);
    }
    printf("\n");
    closedir(dp);
    return CMD_PASS;
}
```