2024350222 오동현

우선 작업을 진행할 새로운 디렉토리를 만들었다.

```
]seed@VM:~$ mkdir ~/my_shell
]seed@VM:~$ cd ~/my_shell
```

그리고 이제 그 디렉토리 안에서 nano 명령어로 뼈대가 될 main.c 코드를 작성했다.

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

코드를 작성하는데 필요한 헤더파일을 우선 include 하였다.

그리고 입력 명령어 최대 길이를 임의로 지정해주고, 상대경로 사용 가능하게 만들 cd와 pwd 함수를 선언해주었다.

그리고, 사용자 입력을 저장할 배열과, 현재 디렉토리 경로를 저장할 배열을 선언했다.

```
#define MAX_CMD_LEN 1024
void my_cd(char *path);  // cd
void my_pwd();           // pwd
```

```
int main() {
    char cmd[MAX_CMD_LEN];
    char cwd[PATH_MAX];
```

그런 다음 첫번째 조건 "1. 현재 디렉토리 명을 쉘이 표시."를 만족시키기 위해 현재 디렉토리의 경로를 출력하도록, 현재 디렉토리 경로를 얻는 함수 getcwd를 사용해, 이를 표시했다.

디렉토리의 경로를 출력하였다면, 그 다음 동작은 사용자의 입력을 받는것이다. 우선 fgets() 함수를 사용하여, 입력값을 아까 생성한 배열에 저장한다. fgets를 사용할까 gets를 사용할까 고민하였

는데, fgets는 gets에 비해 버퍼크기를 지정해서 저장하기 때문에,(sizeof(cmd)) 버퍼 오버플로우의 위험이 gets는 있는 반면, fgets는 안전할 것이라 생각하여 fgets를 사용하였다. 입력값이 들어오지 않는다면, perror 함수를 통해 오류가 발생했다는 것을 알리고 다음 루프로 넘어간다. 그리고 더 정확한 문자열을 저장하기 위해서, ₩n을 제거했다. strcspn() 함수를 사용해 ₩n을 찾고, ₩0 문자로 바꿔서 문자열을 끝내준다.

```
while (1) {
    getcwd(cwd, sizeof(cwd));
    printf("%s$ ", cwd);

    if (!fgets(cmd, sizeof(cmd), stdin)) {
        perror("fgets");
        continue;
    }


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

그런 다음 이제 들어온 입력으로 동작 가능한 함수들을 내부에서 처리한다.

"7. exit 입력 시 프로그램 종료"

Exit을 구현하는 것이 제일 간단하다. 입력된 명령어가 exit과 일치하는지 확인하고, 일치하면 루프를 탈출하면 된다. Strcmp 함수를 사용하여, cmd와 "exit"이 일치하는지 확인했다. 둘이 같다면 0이 결과값으로 나오므로, 0이라면 break가 되도록 하였다.

```
    // exit
    if (strcmp(cmd, "exit") == 0) break;
```

그 다음은 pwd 함수다. 아직 pwd 함수 자체를 구현하지는 않았지만, 일단 pwd가 입력으로 들어왔다면, 아까 선언한 my_pwd() 함수가 실행되도록 만들었다. pwd함수를 실행하고, pwd 함수가 동작을 마치면, 다음 루프로 넘어간다.

```
        // pwd
        if (strcmp(cmd, "pwd") == 0) {
            my_pwd();
            continue;
        }
```

cd 함수도 pwd 함수와 동일한 방식이다. 다만 pwd는 입력값이 그저 pwd로 끝나지만 cd는 입력값이 cd (이동하고 싶은 디렉토리) 이다. 따라서 앞의 "cd " 3개의 문자열만 비교해야 일관성이 유지되므로, strcmp대신, 범위까지 지정 가능한 strncmp를 사용하여 앞의 3개만 비교하였다. 그리고 나머지 뒤의 문자열, 즉 이동을 원하는 디렉토리 값은 cd 함수의 입력값이 된다.

```
        // cd
        if (strncmp(cmd, "cd ", 3) == 0) {
            my_cd(cmd + 3);
            continue;
        }
```

이제 exit, pwd, cd 내부 구현 명령어들을 다루었다면, ls, cat 등과 같은 외부 명령을 실행할 차례이다. 이를 위해 우선 fork()함수를 통해 새 자식 프로세스를 생성했다. 그 다음 pid == 0 이라는 조건문을 통해 자식프로세스에서 동작을 구현한다. 아까 만든 char *argv[]에 외부 명령을 실행하는데 필요한, "/bin/sh" , "-c", cmd, NULL 을 입력해준다. 그리고 이를 execvp()함수를 통해, /bin/sh에 있는 프로그램 경로를 실행하도록 한다. 즉, 우리가 터미널에 /bin/sh -c (cmd) 를 친것과 동일한 효과를 얻을 수 있다. 실패시 아까와 마찬가지로 오류를 출력한다. 마지막으로 자식프로세스가 종료될 때까지 기다리고, 종료상태를 회수하여, 좀비 프로세스를 방지한다. -> wait(NULL)

```
        pid_t pid = fork();
        if (pid == 0) {
            char *argv[] = {"/bin/sh", "-c", cmd, NULL};
            execvp(argv[0], argv);
            perror("exec error");
            exit(1);
        } else {
            wait(NULL);
        }
    }

    return 0;
```

3. cd, pwd 명령어를 반드시 구현 (상대경로 사용 가능해야 함, 단순히 exec을사용해서는 안됨, 나머지 명령어는 exec 시스템콜을 이용해 구현해도 됨)

이제 my_cd, my_pwd 함수를 작성하여 명령어를 구현해야한다.

Pwd 명령어는 현재 디렉토리를 출력한다. Ex) pwd ₩n /home/seed

이처럼 현재 디렉토리가 출력되도록 my_pwd 함수를 만들었다. 아까처럼 getcwd 함수를 통해 cwd에 현재 디렉토리를 저장한다. 그리고 그 값을 출력하면 된다.

```
void my_pwd() {
    char cwd[PATH_MAX];
    getcwd(cwd, sizeof(cwd));
    printf("%s\n", cwd);
}
```

Cd 함수는 cd 뒤에 입력값으로 디렉토리를 이동한다. 그래서 cd 뒤에 디렉토리 입력값을 매개변수로 받아와서 그 위치로 디렉토리를 이동시키는 chdir()함수를 사용했다. 만약 이동이 성공하면 0, 아니면 -1을 반환하므로 0이 아닐 때 오류 메시지를 출력한다.

```
void my_cd(char *path) {
    if (chdir(path) != 0) {
        perror("cd error");
    }
}
```

이로써 main.c에서 기본 인터페이스와, cd, pwd 등 명령어를 구현하였다.

그 다음 프로그램을 빌드하기 위한 Makefile을 만들어주었다. 이 역시 nano 명령어를 사용하였다. Makefile은 혼자 어떻게 작성해야할지 난해해서 gpt의 도움을 받았다.

```
CC = gcc
CFLAGS = -Wall
TARGET = myshell

all: $(TARGET)

$(TARGET): main.c
	$(CC) $(CFLAGS) -o $(TARGET) main.c

clean:
	rm -f $(TARGET)
```

CC는 컴파일러 지정, CFLAGS는 컴파일 할 때 쓸 플래그들 지정, TARGET 은 최종 실행 파일 이름이다. All :$(TARGET)은 사용자가 make 라고 입력했을 때 규칙이 실행되게 하고, 그 아래는 그 소스 파일이 main.c라는 것을 의미한다. 그리고 그 아래줄은 위를 바탕으로 한 실제 컴파일 명령어이다. 그리고 마지막은 make clean을 입력하면, myshell 파일이 삭제되는 코드이다.

이제 파일을 make 하고 `./myshell` 실행하였다.

```
/home/seed/my_shell$ pwd
/home/seed/my_shell
/home/seed/my_shell$ ls
main.c  Makefile  myshell
/home/seed/my_shell$ cd ..
/home/seed$ cd my_shell
```

```
/home/seed/my_shell$ exit
[05/12/25]seed@VM:~/my_shell$
```

이와 같이 정상적으로 작동하는 것을 볼 수 있다. 지금까지 조건 1, 2, 3, 7을 만족한 것을 볼 수 있다.

이제 파이프라인(|) 이 작동하게 해야한다. 이를 위해 main.c를 수정해주었다.

우선 파이프라인을 처리할 함수를 만들었다.

```
void pipeline(char *cmdline); //pipeline
```

Strchr은 문자열 내에 원하는 값이 존재하는지 찾는 함수다. 이를 이용해 입력받은 값에 | 이 포함되어 있다면, 파이프라인 함수를 호출한다.

```
        if (strchr(cmd, '|')) {
            pipeline(cmd);
            continue;
        }
```

이제 파이프라인 함수를 구현할 차례이다. Strtok는 문자열을 |을 기준으로 자르는 명령어이다. Cmds는 16개의 포인터 배열이고, num_cmds는 현재 존재하는 배열의 개수이다. |를 기준으로 따로따로 분해해서 보아야하기 때문에 입력값을 |를 기준으로 잘라서 저장하게 해주었다. 예를 들어 ls -l | grep txt | wc -l 라면 ls -l, grep txt, wc -l을 분리해서 다루어야 하기 때문이다. Cmds 배열이 16개가 최대이기 때문에, 토큰 분리를 토큰이 없거나, 16개가 넘지 않을 때까지 반복해준다. 또한 입력할 때 공백문자가 있을 수 있기 때문에, 공백이 있을 때는 저장하지 않고, 토큰 주소를 1칸 증가시킨다. 그리고 그렇지 않은 경우에는 cmds 포인터 배열에, 토큰 포인터가 순서대로 저장되고 카운트가 증가되도록 하였다. 그리고 그 다음 |까지 이 과정을 반복하도록 했다.

```
void pipeline(char *cmdline) {
    char *cmds[16];
    int num_cmds = 0;

    char *token = strtok(cmdline, "|");
    while (token != NULL && num_cmds < 16) {
        while (*token == ' ') token++;
        cmds[num_cmds++] = token;
        token = strtok(NULL, "|");
    }
```

```
    int pipefd[2], prev_fd = -1;
```

pipefd: 현재 명령어의 출력용 파이프, prev_fd : 이전 명령어의 출력; 초기화.

0부터 존재하는 배열의 개수까지 반복해준다. 그리고 pipe함수를 사용해 현재 명령어가 쓸 파이프를 생성한다. 그리고, 각 명령어들마다 fork()함수를 통해 자식 프로세스를 생성해주었다.

```
for (int i = 0; i < num_cmds; ++i) {
    pipe(pipefd);
    pid_t pid = fork();
```

If pid == 0 , 즉 자식 프로세스가 잘 만들어 졌을 때의 조건문을 작성했다. i != 0, 즉 첫번째 명령어가 아닐 때는 이전 출력의 다음 명령어의 입력이 되어야 한다. 이를 위해서 사용한 것이 dup2() 함수이다. Dup2(a,b)는 간단히 말해서, a->b로 복제가 된다. 이를 사용해 이전 파이프의 출력이 현재 명령의 입력이 되도록했다. 다시 출력이 발생하면, 그걸 다음 명령의 입력으로 사용해야하므로, 마지막 명령이 아니면 dup2를 또 사용해, 현재 출력을 파이프 write에 연결한다.

```
    if (pid == 0) {
        if (i != 0) {
            dup2(prev_fd, STDIN_FILENO);
            close(prev_fd);
        }
        if (i != num_cmds - 1) {
            dup2(pipefd[1], STDOUT_FILENO);
        }
        close(pipefd[0]);
        close(pipefd[1]);
```

입출력 리다이렉션 과정이 끝났다면, 명령을 실행해주어야 한다. 그래야 출력 값도 나올 테니 말이다. 아까 외부 명령어 실행과 같은 방식으로, 명령어를 실행한다. 만약 에러가 발새한다면 메시지를 띄우고 종료.

```
        char *args[] = {"/bin/sh", "-c", cmds[i], NULL};
        execvp(args[0], args);
        perror("exec error");
        exit(1);
```

마지막으로, wait()함수로 자식 프로세스가 종료될 때까지 기다리고, prev_fd != -1, 즉 첫번째 명령이 아니라면, prev_fd도 닫는다. 마지막으로 이번 명령의 출력 값을 prev_fd에 갱신해주었다.

```
        exit(1);
    } else {
        wait(NULL);
        close(pipefd[1]);
        if (prev_fd != -1) close(prev_fd);
        prev_fd = pipefd[0];
    }
```

이를 저장하고, make를 이용해 다시 실행시키고, ./myshell 을 눌러 셸을 실행시켜서 파이프라인을 테스트했다.

```
/home/seed/my_shell$ ls |grep main
main.c
/home/seed/my_shell$ ps aux | grep bash | wc -l
1
```

정상적으로 동작하는 것을 확인하였다. 3개도 되는 것을 확인하였고, 4번 조건도 만족하였다.

이제 5. 다중 명령어 (; , &&, ||) 6. 백그라운드 실행(&) 을 만족시켜야한다.

Main.c에 두 조건을 구현하는 코드를 추가한다. 우선 다중 명령어(;,&&,||)를 실행할 함수를 추가한다. 기본적으로 다중명령어 실행 때 연속적으로 여러 명령 코드(pwd, ls 등) 가 실행될 수 있다 생각해서 그냥 main 함수에서 분리해서 새로운 함수에 집어넣는게 좋겠다고 생각했다.

```
void single_command(char *cmd); //single command
```

이 다음, main에서 exit 호출 아래 부분(cd, fwd, 외부함수호출)을 전부 새로운 함수에 옮겨주었다. 그리고 main 부분에는 다중명령 부분을 처리하는 코드를 새로 구현하였다.

Strsep(a,b) 함수는 b가 문자열에 하나라도 있으면 그 단위로 문자열을 끊어 저장한다. 즉, current에 cmd 문자열을 ; 기준으로 끊어 저장한다. 예: pwd ; ls ; echo hi 라면 3개로 끊어 저장. Last_status 변수는 이전 명령의 실행 결과를 저장하기 위해 만들었다. 0이면 이전 결과가 성공이다. 이를 통해 ;, &&, || 사이의 두 명령어의 실행 여부를 판단 가능하다.

```
char *rest = cmd;
char *current;
int last_status = 0;

while ((current = strsep(&rest, ";")) != NULL) {
```

&&, ||, ; 일때를 분리하여 처리한다.

&&: strstr함수는 &&이 있다면 0을 반환하지 않으므로, &&이 있다면 if문이 실행되도록한다. &&을 기준으로 왼쪽을 left, 오른쪽을 right로 분리했다.

```
        if (strstr(current, "&&")) {
            char *left = strtok(current, "&&");
            char *right = strtok(NULL, "");
```

아까처럼, 공백이 있다면 제거하고, 단일 명령어를 실행한다. 성공한다면 0이 반환되었을 것이다.

```
            while (*left == ' ') left++;
            last_status = single_command(left);
```

그래서 &&는 앞이 성공해야 뒤 명령도 실행되므, 이전 명령 반환 값이 0이고,(성공했다는 뜻), right 이 존재하면, right도 실행하도록 했다.

```
            if (last_status == 0 && right) {
                while (*right == ' ') right++;
                last_status = single_command(right);
            }
```

||: 이번에는 ||일때의 경우다. 이때는 앞의 명령이 동작하지 않아야 뒷 명령을 실행한다. 똑같이 left, right으로 분리했다.

```
        else if (strstr(current, "||")) {
            char *left = strtok(current, "||");
            char *right = strtok(NULL, "");
```

이번에는 &&와 반대로 left의 명령을 실행하고, 그 반환값이 0이 아니면(실패하면), right 명령을 실행하도록 코드를 작성했다.

```
            while (*left == ' ') left++;
            last_status = single_command(left);

            if (last_status != 0 && right) {
                while (*right == ' ') right++;
                last_status = single_command(right);
            }
        }
```

;: 마지막으로 ;는 그냥 앞 명령어의 성공여부와 관계없이 실행하므로, 현재 명령을 실행한다음, 그 실행이 성공했는지 실패했는지 여부만 저장하도록 했다.

```
        else {
            while (*current == ' ') current++;
            last_status = single_command(current);
        }
    }
```

Single_command 함수는 main에 있을때와 큰차이는 없다. Pwd, cd, 파이프라인을 실행하고 실행이 정상적으로 되면 0을 리턴한다. 또한 외부 함수 호출에서도, 원래는 wait(NULL)로 좀비 프로세스를 방지하였으나, main함수로 0 또는 1의 리턴값을 보내야 하므로, waitpid(pid, &status, 0); 으로, 자식프로세스가 종료되면 status에 그 값이 저장된다. WEXITSTATUS(status)는 그 중 **"정상 종료 시 종료 코드"만 추출**하는 매크로이다. 이를 통해서, 외부 함수 호출에서도 main 함수로 호출 정상 작동 여부를 보낼 수 있도록 하였다.

```
void single_command(char *cmd){
    if (strcmp(cmd, "pwd") == 0) {
        my_pwd();
        return 0;
    }

    if (strncmp(cmd, "cd ", 3) == 0) {
        my_cd(cmd + 3);
        return 0;
    }

    if (strchr(cmd, '|')) {
        pipeline(cmd);
        return 0;
```

```
    pid_t pid = fork();
    if (pid == 0) {
        char *argv[] = {"/bin/sh", "-c", cmd, NULL};
        execvp(argv[0], argv);
        perror("exec");
        exit(1);
    } else {
        int status;
        waitpid(pid, &status, 0);
        return WEXITSTATUS(status);
    }
}
```

이제 마지막으로 6. 백그라운드 실행(&)을 구현하면 된다. 이는 함수호출이 single_command 함수에서 이루어지므로 single_command 함수를 수정해줄 필요가 있다. 우선 변수 background를 선언한 뒤, 공백 제거도 해주고, 입력이 background 실행인지 판별해야한다. 우선 명령어 길이를 strlen 함수로 구한 다음, 길이가 0보다 크고, 마지막 입력값이 &인지 확인한다. &이 맞으면 background 실행이므로, background = 1로 수정하고, &를 ₩0으로 교체해서 제거해준다. &이 없어야 명령이 정상적으로 실행된다. 또한 &앞에 공백이 붙어있었다면 제거하여서, 진짜 명령어 부분만 남도록 하였다.

```
int background = 0;

while (*cmd == ' ') cmd++;

int len = strlen(cmd);
if (len > 0 && cmd[len - 1] == '&') {
    background = 1;
    cmd[len - 1] = '\0';
    while (len > 1 && cmd[len - 2] == ' ') {
        cmd[len - 2] = '\0';
        len--;
    }
```

그리고 나머지(pwd, cd ,,,) 부분은 그대로 두고, wait()이 있는 외부 함수 호출 부분만 수정해주었다. 만약 백그라운드 입력이 아니라면 !background 값이 0이므로 첫번째 if 분기를 타고 이전과 같이 동작한다. 하지만 백그라운드 입력이라면 else로 넘어가서 백그라운드에서 명령이 실행되었음을 출력하고, 정상적으로 작동되었음을 의미하는 0 값을 리턴한다.

```
} else {
    if (!background) {
        int status;
        waitpid(pid, &status, 0);
        return WEXITSTATUS(status);
    } else {
        printf("[background pid: %d]\n", pid);
        return 0;
    }
}
```

실행하니 오류가 떴다. 자세히 살펴보니, single_command 함수를 void 형으로 만드는 어처구니 없는 실수를 했다. 그래서 이 부분을 int로 수정하고 다시 실행해주었다.

;와 ||는 잘 작동하는데 &&만 에러가 남.

```
/home/seed/my_shell$ ls && echo success
main.c  Makefile  myshell
/bin/sh: 1: Syntax error: "&" unexpected
```

&가 처리과정에서 제대로 제거되지 않은 듯 해서 코드를 수정해주었다.

기존의 strtok가 원인인듯 싶어서, 코드를 수정하였다. op는 "&& (~~~)" 문자열을 가리킨다. 그리고 *op의 첫문자를 NULL문자로 바꾸고, 좌, 우를 분리해 주었다. 오른쪽은 +2를 해서 &&를 제외한 나머지 문자열 값을 가진다. 그리고 공백을 제거하고 아까와 마찬가지로 성공하면 실행, 성공못하면 실행 안하도록 코드를 수정하였다.

```
        char *op = strstr(current, "&&");
        if (op) {
        *op = '\0';
        char *left = current;
        char *right = op + 2;

        while (*left == ' ') left++;
        while (*right == ' ') right++;

        last_status = single_command(left);
        if (last_status == 0)
        last_status = single_command(right);
        }
}
```

코드를 수정하고 실행하니 ;, &&, || 모두 정상적으로 작동하는 모습이다.

```
/home/seed/my_shell$ echo hi ; echo ok
hi
ok
/home/seed/my_shell$ ls && echo success
main.c  Makefile  myshell  tmp
success
/home/seed/my_shell$ false || echo failed
failed
```

```
/home/seed/my_shell$ sleep 5 &
[background pid: 29366]
```

마지막으로 백그라운드 실행도 잘 작동하는 것을 확인하였다. 요구 기능을 구현하는 shell을 완성하였다.