CYKOR 2주차 보고서

2025350003 최동주

목차

1. 중요코드 설명

```
int main(){
    //유저네임과 호스트네임 입력받기
    printf("Enter username:");
    scanf("%49s", username);
    printf("Enter hostname:");
    scanf("%49s", hostname);
    //본격적인 bash 실행
    bash(username, hostname);

    return 0;
}
```

main함수의 형태이다. 프롬프트 출력을 위해 크기가 50인 username과 hostname을 입력받는다. 이후 본격적으로 bash함수를 실행한다.

```
void bash(char*user, char*host) {
    char command[COMMAND_SIZE];
    int input_buffer;

    //현재 디렉토리를 /로 설정
    if(chdir("/") !=0) {
        perror("디렉토리 변경 실패");
        return;
}
    getcwd(path, sizeof(path));
    // 최초 실행 시 버퍼에 남은 개행 제거
    while((input_buffer =getchar()) !='\n'&&input_buffer !=EOF);
    //명령어 입력받기 프롬프트
    while(1) {
```

```
        //프롬프트 형식
        dir =opendir(".");
        printf("%s@%s:", user, host);
        printf("%s", path);
        printf("$ ");
        if(fgets(command, COMMAND_SIZE, stdin)==NULL) continue;
        // 개행 문자 제거
        command[strcspn(command, "\n")] ='\0';

        if(command_process(command) ==2) break;
}
}
```

bash 함수이다. 지속적으로 실행이 되야 하기에 void 형식이며 아무런 값도 반환하지 않는다. 디렉토리 변수 dir를 통해 현재 디렉토리를 오픈하며, 프롬프트를 출력한다. 또한 getcwd를 통해 path라는 변수에 현재 디렉토리 경로를 저장해준다. 이후, fgets를 통해 명령어를 입력받아 command에 전달, 이것을 이후 명령어함수인 command_process로 전달하며, 이때 fgets를 통해 COMMAND_SIZE인 256만큼 제한적으로 코드를 입력받기에 버퍼 오버플로우에 대한 위험을 낮추었다. 만약 command_process에서 2의 값을 반환받으면 실행이 종료된다.

```
void remove_space(char*b) {
    int i=0;
    while(b[i] ==' ') i++;
    if(i >0) memmove(b, b +i, strlen(b +i) +1);
    int len =strlen(b);
    while(len >0 &&b[len -1] ==' ') {
        b[len -1] ='\0';
        len--;
}
}
```

문자열의 앞뒤 공백 제거 함수이다. 앞쪽 공백의 크기만큼을 i에 저장하고, memmove 함수를 통해 i만큼 문자열을 앞으로 땡긴다. 이후 반복적으로 마지막 문자(널 문자 제외)가 공백일 시 널문자로 바꾸어 최종적으로 앞,뒤 공백들을 제거한다. 이때 입력을 포인터로 받기에 b는 call by reference에 따라 호출한 함수 내에서도 값이 변경된다.

```
int command_process(char*str) {
    remove_space(str);
```

명령어를 인자로 받아 명령을 수행하는 함수이다. 함수 실행시 가장 먼저 입력받은 문자열의 앞뒤 공백을 제거하고 시작한다. 이 함수는 명령어 수행 성공시 1을, 실패시 0을 반환하며, exit를 입력받을 시에는 2를 반환한다.

```
//다중명령어1: ; 구현
    if(strstr(str,";")!=NULL) {
        char*bookmark =strstr(str,";");
        char front_command[COMMAND_SIZE], back_command[COMMAND_SIZE];
        int bookmark_length =bookmark -str;
        strncpy(front_command,str,bookmark_length);
        front_command[bookmark_length] ='\0';
        remove_space(front_command);
        strcpy(back_command,bookmark+1);
        remove_space(back_command);
        command_process(front_command);
        command_process(back_command);
        return 1;
}
```

다중명령어 ; 구현코드이다. 가장 처음 나오는 ;의 위치를 찾아 bookmark에 저장하고, bookmark 포인터와 명령어 시작 포인터간의 연산을 통해 길이를 구한 후 이 길이만큼 ; 앞의 문자열을 복사해 frontcommand에 저장하고, 남은 ; 뒤의 코드들은 back_command에 복사시킨 후 각각의 공백을 제거시키고 command_process에 재귀호출 시켰다. 이런 방식이면 ;가 여러개 있어도 지속적으로 분리시켜 결국에는 ;가 모두 나누어질 때 까지 명령어를 수행할 수 있다. 이후 ;로 나누는 것을 성공했다는 의미로 1을 반환한다.

```
//다중명령어 &&, || 구현
    else if(strstr(str, "||") !=NULL ||strstr(str, "&&") !=NULL) {
        char *commands[COMMAND_SIZE];
        int num_commands =0;
        int success =0;
        // && 연산자 우선 분리
```

```
        char *token =strtok(str, "&&");
        while(token !=NULL &&num_commands <COMMAND_SIZE) {
            commands[num_commands++] =token;
            token =strtok(NULL, "&&");
}
        for(int i =0; i <num_commands; i++) {
            char *sub_commands[COMMAND_SIZE];
            int sub_num =0;
            // || 연산자 분리 (strtok 로직에 따라 || 없을시 '그대로' 토큰에 저
장!)
            token =strtok(commands[i], "||");
            while(token !=NULL &&sub_num <COMMAND_SIZE) {
                sub_commands[sub_num++] =token;
                token =strtok(NULL, "||");
}
            for(int j =0; j <sub_num; j++) {
                success =command_process(sub_commands[j]);
                // OR 연산자 하나라도 성공시 탈출
                if(success) break;
}
            // AND에서 실패시 전체 중단
            if(!success) break;
}
}
```

다중명령어 &&과 || 구현 코드이다. strtok를 통해 우선 &&를 분리해 command에 저장하고, 다시 strtok를 통해 ||를 찾아 분리해 sub_commands에 저장한다. 이때 strtok의 로직에 따라 문자열에 ||가 있다면 분리해서 토큰에, 없다면 통으로 토큰에 저장시킨다 이후 sub_command의 명령어들을 수행시키는데, 이때 command_process가 반환하는 값들을 success변수에 저장시키고, success값이 1이냐 0이냐에 따라 논리적으로 연산을 계속 수행할지, 아니면 멈출지를 결정한다.

```
// 파이프라인 구현
    else if(strstr(str,"|")!=NULL) {
        char *token =strtok(str, "|");
        char *commands[COMMAND_SIZE];
        int pipe_fd[COMMAND_SIZE -1][2];
        int num_commands =0;
```

```
        while(token !=NULL &&num_commands <COMMAND_SIZE) {
            commands[num_commands++] =token;
            token =strtok(NULL, "|");
}
        // 개수만큼 파이프 생성
        for(int i =0; i <num_commands -1; i++) {
            if(pipe(pipe_fd[i]) ==-1) {
                perror("pipe");
                exit(0);
}
}


        for(int i =0; i <num_commands; i++) {
            pid_t pid =fork();
            if(pid ==-1) {
                perror("fork");
                exit(0);
}
            if(pid ==0) {
                // 이전 프로세스에서 입력 받기
                if(i >0) {
                    dup2(pipe_fd[i -1][0], STDIN_FILENO);
                    close(pipe_fd[i -1][0]);
}
                // 다음 프로세스로 출력 보내기
                if(i <num_commands -1) {
                    dup2(pipe_fd[i][1], STDOUT_FILENO);
                    close(pipe_fd[i][1]);
}
        // 파일 디스크립터 닫기
                for(int j =0; j <num_commands -1; j++) {
                    close(pipe_fd[j][0]);
                    close(pipe_fd[j][1]);
}

                // 각 명령어 실행
                char *args[]={"/bin/sh", "-c", commands[i], NULL};
                execvp(args[0], args);
                perror("execvp");
```

```
                exit(0);
}
}
        // 부모 프로세스에서 파일 디스크립터 닫고 기다리기기
        for(int i =0; i <num_commands -1; i++) {
            close(pipe_fd[i][0]);
            close(pipe_fd[i][1]);
}
        for(int i =0; i <num_commands; i++) {
            wait(NULL);
}
        return 1;
}
```

(멀티)파이프라인 구현 코드이다. strtok를 활용해 |를 기준으로 명령어들을 나누어 commands에 저장하고, 이때 명령의 개수를 num_commands에 저장한다. 이후 num_commands-1의 개수만큼 파이프를 만들고, num_commands의 개수만큼 fork()를 해 명령어가 수행 될 자식 프로세스를 만든다. 이후 각각의 명령어들은 dup2함수와 파이프를 이용해 STDIN_FILENO를 통해 이전 명령어들의 수행값을 받아오고, STDOUT_FILENO를 통해 결과를 파이프를 통해 다음 프로세스로 보낸다. 명령어 수행 이후 부모와 자식에서 더 이상 사용이 되지 않는 파일 디스크립터들을 모두 닫아주고, 자식프로세스에서 각각의 명령어들이 수행이 될 때까지 wait해준다. 이때 wait을 통해 자식 프로세스들을 정리하여 좀비 프로세스를 죽여준다.

```
// 백그라운드 실행 구현
    else if(strrchr(str,'&')!=NULL){
        char*bg =strrchr(str,'&');
        char bgcmd[50];
        int bg_length =bg -str;
        strncpy(bgcmd,str,bg_length);
        bgcmd[bg_length] ='\0';

        background =1;
        //자식 프로세스 생성
        pid_t pid =fork();
```

```
        if(pid <0) {
            perror("fork 실패");
}
        else if(pid ==0) {
            // 자식 프로세스: 실제 명령 실행
            command_process(bgcmd);
            exit(0);
}
        //부모 프로세스
        else
            if(background) {
                //실행 여부 및 자식 프로세스의 식별 ID 출력
                printf("백그라운드 실행중\t[pid: %d]\n",pid);
                fflush(stdout);
} else{
                waitpid(pid,NULL,0);
}
}
        return 1;
}
```

백그라운드 실행 코드이다. &를 찾아 포인터주소와 문자열 시작 포인터주소를 토대로 앞의 명령어를 bgcmd에 저장시키고, background 값을 1로 바꾼다. 이후 자식 프로세스를 만들어 명령어를 수행시키고, 실행 여부와 식별 아이디를 출력하고 프롬프트 중복 출력을 방지하기 위해 표준출력을 비워준다. 이후 좀비 프로세스 방지를 위해 waitpid를 사용해주고, 명령어 수행이 끝났다는 의미에서 1을 반환해준다.

```
// exit 구현
    else if(strcmp(str, "exit") ==0) {
        printf("BASH를 종료합니다.\n");
        return 2;
}
    //pwd 구현
    else if(strcmp(str, "pwd") ==0) {
        printf("%s\n", path);
        return 1;
}
    //chdir 통한 cd 구현
```

```
    else if(strncmp(str, "cd ", 3) ==0) {
        char*change_path =str +3;
        if(chdir(change_path) !=0) {
            perror("디렉토리 변경 실패");
            return 0;
}
        else{
            getcwd(path, sizeof(path));
            return 1;
}
}
    else{// 이 외의 단일 명령어 실행
        pid_t pid =fork();
        if(pid ==-1) {
            perror("fork");
            return -1;
}
        if(pid ==0) {// 자식 프로세스
             execlp("/bin/sh", "sh", "-c", str, NULL);// 단일 명령어 직접 실
행
            perror("execlp");
            exit(0);
} else{
            waitpid(pid,NULL,0);// 자식 프로세스 종료 대기
            return 1;
}
}

    return 0;
```

실질적으로 명령어 수행을 다루는 코드들이다. 우선 exit 입력시 2의 값을 반환하며 bash함수를 종료시킨다. pwd 입력시 앞서 저장한 path의 값을 출력한다. 명령어의 첫 3문자에서 cd와 공백을 탐지할 시 chdir를 통해 뒤에 오는 디렉토리로 변경을 시도하고, path의 값을 변경된 디렉토리로 바꾼다. 이외에 기타 명령어들(ls, date 등등)은 자식프로세스 생성 후 빈폴더에 있는 실제 리눅스 수행 명령어들을 가져와 execlp를 통해 수행시킨다. 물론 이때도 좀비 프로세스 방지를 위해 wait을 해준다. 명령어 수행 이후 1의 값을 리턴해주었다. 만약 위의 모든 명령어들이 수행되지 않았을 경우 0을 반환한다.

2. 실행 결과 예시

```
Enter username:ANDREW3
Enter hostname:BOOK-1122
ANDREW@BOOK-1122:/$ cd sys
ANDREW@BOOK-1122:/sys$ pwd
/sys
ANDREW@BOOK-1122:/sys$ cd /
ANDREW@BOOK-1122:/$ pwd
/
ANDREW@BOOK-1122:/$ ls | grep lib | grep 64
lib64
ANDREW@BOOK-1122:/$ cd sys;pwd
/sys
ANDREW@BOOK-1122:/sys$ ls && echo "Success" || echo "Failure"
block  bus  class  dev  devices  firmware  fs  kernel  module
Success
ANDREW@BOOK-1122:/sys$ pwd &
백그라운드 실행중          [pid: 24903]
/sys
ANDREW@BOOK-1122:/sys$ date
Sun May 11 22:07:05 KST 2025
ANDREW@BOOK-1122:/sys$ exit
BASH를 종료합니다.
```

3. 제작 일지

4/09: 기초 형태 고안

영상을 시청한 이후, 코드를 작성하기에 앞서 어떠한 방식으로 코드를 작성해야할지 고민을 해보았다. 우선, 전체적인 운영방식은

구현에 필요한 전역변수 선언
main함수에서 유저네임과 호스트네임 입력받기
bash함수 호출
입력 프롬프트 띄우기 및 명령어 입력받기

으로 할 것 같다.

```
#include <stdio.h>
#include <string.h>

//bash의 유저네임과 호스트네임 선언
char username[50];
char hostname[50];
char prompt[50][50];

void bash(char* user, char* host);

int main(){
	//유저네임과 호스트네임 입력받기
	printf("Enter username:");
	scanf_s("%s", username,50u);
	printf("Enter hostname:");
	scanf_s("%s", hostname,50u);

	//프롬프트에 유저네임과 호스트네임 추가
	sprintf_s(prompt[0], sizeof(prompt[0]), "%s", username);
	sprintf_s(prompt[1], sizeof(prompt[1]), "%s", username);


	//본격적인 bash 실행
	bash(username, hostname);

	return 0;
}

//리눅스 bash셸의 기본적인 인터페이스 및 명령어 입력받기
void bash(char* user, char* host) {
	char command[50];
	while (1) {
		printf("%s@%s:/", user, host);
		//....
		printf("$ ");
		scanf_s("%s",command,50);
		printf("%s\n", command);

	}
}
```

그에 대한 대략적인 형식이 위의 코드이고, 실행을 해보면

```
Enter username:andrew
Enter hostname:BOOK-1122
andrew@BOOK-1122:/$ cd
cd
andrew@BOOK-1122:/$ ls
ls
andrew@BOOK-1122:/$
```

처럼 나온다. command 변수는 프롬프트에 입력한 명령어를 저장하는 변수인데, 위의 결과를 통해 입력받은 변수가 잘 저장되고, 다음에 입력받았을 때 잘 덮어씌어짐을 확인할 수 있었다. 일단은 여기까지 구현하고, 나머지 방식에 대해서는 더 고민을 해봐야할것 같다. commit!

4/11: bash 기초 구현
이틀간 과제를 어떻게 구현해야 할까 고민한 끝에, 우선 입력받는 방식부터 바꾸기로 했다. 기존의 scanf는 공백을 기준으로 끊기니 cd home 과 같은 명령어 입력이 어려울 것 같고, scanf를 사용하게 된다면 혹시 모를 버퍼 오버플로우 오류가 발생 할 수 있으니 fgets를 사용하기로 했다(물론 이것은 _s를 붙여 버퍼 크기를 지정해주면 해결되긴 한다)

그래서 `fgets(command, COMMAND_SIZE, stdin);` 를 넣어줬더니

```
Enter username:andrew
Enter hostname:BOOK-1122
andrew@BOOK-1122:/$

andrew@BOOK-1122:/$ d
d

andrew@BOOK-1122:/$ d
d

andrew@BOOK-1122:/$
```

처럼 공백 입력과 명령어 입력 후에 줄 바꿈이 자동으로 되는 불상사가

일어났다. 왜 그런지 서칭을 해보니, scanf가 사용된 직후(유저네임과 호스트네임을 받으려고 사용함) fgets를 사용하면 입력버퍼에 남아있는 개행문자 \n 때문에 공백 입력 및 출력이 일어난다고 한다. 이럴땐 입력 버퍼를 비워야 한다고 하니 다음과 같이 수정했다.

```
void bash(char* user, char* host) {
        char command[COMMAND_SIZE];
        int input_buffer;

        // 최초 실행 시 버퍼에 남은 개행 제거
        while ((input_buffer = getchar()) != '\n' && input_buffer != EOF);
        while (1) {
                printf("%s@%s:/", user, host);
                //....
                printf("$ ");
                fgets(command, COMMAND_SIZE, stdin);
                // 개행 문자 제거
                command[strcspn(command, "\n")] = '\0';
                printf("%s\n", command);

        }
}
```

원리는 다음과 같다. 예를 들어 앞서 scanf로 hostname을 입력하면 실제로는 h o s t n a m e \n처럼 입력이 된다 이렇게 되면 입력버퍼에 \n이 남게 되고, fgets를 쓰면 유저의 입력과는 상관없이 \n이 입력되게된다. 따라서 입력버퍼 변수 input_buffer를 지정해주고, while문과 getchar를 통해 입력버퍼에 남아있는 값을 반환하여 input_buffer의 값을 초기화 해주고, 그 값이 /n이면 and 조건문 &&가 깨지게 되어 루프가 종료되고, 입력버퍼에 있는 /n이 다른 변수에 저장되었으므로 입력버퍼는 없어지게 된다. 이후 while(input_buffer = getchar() != '\n')을 gpt에게 디버깅 시켰더니 혹시 모를 무한루프 문제가 생길수도 있다고 해서 input_buffer != EOF, 즉 입력이 끝나면 루프가 종료될 수 있게 조건을 추가해주었다. 실행을 해보니

```
Enter username: andrew
Enter hostname: BOOK-1122
andrew@BOOK-1122:/$ cd
cd
andrew@BOOK-1122:/$ cd home
```

```
cd home
andrew@BOOK-1122:/$ cd /
cd /
andrew@BOOK-1122:/$ exit
exit
andrew@BOOK-1122:/$
```

처럼 결과가 잘 나오는 것을 확인할 수 있었다. 우선은 commit을 하겠다.

이후 exit을 입력하면 종료가 될 수 있도록
if(strcmp(command,“exit”)==0) return을 적었고, 정상적으로 작동하는 것을 확인했다.

```
Enter username:andrew
Enter hostname:BOOK-1122
andrew@BOOK-1122:/$ ls
ls
andrew@BOOK-1122:/$ cd home
cd home
andrew@BOOK-1122:/$ exit
BASH를 종료합니다.
C:\Users\gamin\source\repos\cykor_week2\x64\Debug\cykor_week2.exe(프로세스 14320)이(가) 0 코드(0x0)와 함께 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요...
```

우선은 커밋을 하겠다.

4/14: 디렉토리 관련

c언어에서 호스트머신의 실제 디렉토리를 어떻게 참조할 수 있을지 서칭을 해보니, 디렉토리 참조에 사용할 수 있는 <dirent.h>라는 라이브러리를 발견했다. 그러나 해당 라이브러리는 윈도우 운영체제에서는 기본적으로 있지 않으며, 사용을 위해선 별도의 설치과정이 필요하지만 과정이 복잡해보이길래 대신 비슷한 direct.h를 사용하기로 했다.

우선은 경로를 저장할 변수를 만들었다. 기존의 prompt[50][50]을 활용해 트리구조로 디렉토리들을 저장하려 했지만, direct.h가 제공하는 함수들을 이용해 현재 디렉토리에 대한 문자열을 반환할 수 있으므로 prompt[50][50]을 path[50]으로 바꾸고, bash 내부에 다음과 같은 코드를 작성하였다.

```
//현재 디렉토리를 /(c://)로 설정
if (_chdir("/") != 0) {
        perror("디렉토리 변경 실패");
        return;
}
_getcwd(path, sizeof(path));
```

처음에는 단순히 _chdir(“/”);라고 했지만, 그렇게 하니 반환값이 무시되었다고 경고가 떠 해당 코드로 수정하였다. 이후 strcmp를 이용해 pwd가 입력되었을 때 현재의 경로를 출력하게 해보면

```
Enter username:andrew
Enter hostname:BOOK-1122
andrew@BOOK-1122:/$ pwd
C:\
andrew@BOOK-1122:/$ exit
BASH를 종료합니다.
```

정상적으로 작동한다. 우선은 커밋을 하겠다.

4/15: cd 구현

어제 pwd까지 구현을 완료했다. 이제 cd를 구현할 것이다. pwd 구현 후 다양한 방법을 고안해보았는데, 우선 pwd나 exit와는 다르게 cd는 cd(공백)(디렉토리) 형식으로 입력되므로 cd와 공백을 탐지하면 그 뒤의 디렉토리만 반환할 수 있는 방법을 사용하기로 했다. strcmp를 공부하고 사용하며 strncmp를 알게되었는데, 이 함수는 특정 문자열들의 비교 길이, 즉 몇번째 문자까지 비교할 수 있도록 지정이 가능하기에 이 함수를 사용하기로 했다.

```
else if (strncmp(command, "cd ", 3) == 0)
```

command에서 c d 공백을 발견하는 코드를 작성해 주었다. 이후 디렉토리를 어떻게 뽑을지 고민을 해보았는데, 생각을 해보니 포인터 문법에서

(char + n)처럼 문자열의 특정 부분 주소를 반환할 수 있다는 것이 생각을 났다. 이것을 활용하면 좋을 것 같아 리서칭 하며 코드를 완성한 결과

```
else if (strncmp(command, "cd ", 3) == 0) {
        char* change_path = command + 3;
        if (_chdir(change_path) != 0) {
                perror("디렉토리 변경 실패");
        }
        else {
                _getcwd(path, sizeof(path));
        }

}
```

else if 아래는 사실 위의 기본 경로 설정에서 거의 복붙해온 코드이긴 하다. 또한 경로가 바뀌면 bash의 입력 프롬프트 역시 현재 경로로 바껴야 하므로 출력 while문 아래에

```
printf("%s@%s:", user, host);
printf("%s", path);
printf("$ ");
```

로 바꾸어주고 실행을 했다. 그 결과

```
Enter username:ANDREW
Enter hostname:BOOK-1122
ANDREW@BOOK-1122:C:\$ pwd
C:\
ANDREW@BOOK-1122:C:\$ cd C:\Windows
ANDREW@BOOK-1122:C:\Windows$ pwd
C:\Windows
ANDREW@BOOK-1122:C:\Windows$ exit
BASH를 종료합니다.

Enter username:andrew
Enter hostname:book-1122
andrew@book-1122:C:\$ pwd
C:\
andrew@book-1122:C:\$ cd Windows
andrew@book-1122:C:\Windows$ pwd
C:\Windows
```

andrew@book-1122:C:\Windows$

절대 경로와 상대경로 모두 구현을 완료했다.

또한 강의 자료를 공부하던 중, 리눅스 환경이라고 적혀있는것이 '리눅스 환경을 구현'하는 것이 아닌 '리눅스 환경에서 구현'이라는 것을 알게 되었다. 이에 따라 코드와 git을 vsc ubuntu linux로 환경을 옮기고, linux에서 사용가능하도록 일부 코드를 수정하였다.

과제를 수행하다보니, 리눅스 bash의 기본중에 기본인 ls 명령어를 구현하는 것이 좋겠다는 생각이 들었다. 추가적인 기능을 구현하는거라 필수는 아니지만, 있는 편이 훨씬 낫겠다는 생각이 pwd와 cd를 만들며 계속 들어 만들기로 했다.

리눅스로 옮겼으니, dirent.h를 사용할 수 있으므로 이를 사용하여 만든 코드는 다음과 같다.

```
DIR*dir;
struct dirent*ent;
else if(strcmp(command, "ls") ==0) {
     while((ent =readdir(dir)) !=NULL) {
     printf("%s\n", ent->d_name);
}
}
```

dirent.h에서 정의한 구조체와 함수를 이용해 간단하게 구현해보았다. 원리는 다음과 같다. 우선, 정의된 구조체를 통해 디렉토리 스트림 변수인 dir과 디렉토리에 대한 정보를 담는 변수인 ent를 선언해준다. readdir함수는 dir에 있는 하위 디렉토리들을 읽어오는데, 성공하면 구조체 포인터를, 실패하면 NULL을 반환한다고 하니, NULL을 반환하는 경우가 아니면, 반환받은 구조체 포인터를 ent에 저장하고, 구조체에 사용되는 접근 연산자 ->를 통해 하위 디렉토리의 이름을 불러온다. 실행 결과는 다음과 같다.

Enter username:ANDREW

```
Enter hostname:BOOK-1122
ANDREW@BOOK-1122:/$ cd
ANDREW@BOOK-1122:/$ ls
tmp
wsljJppOp
sbin.usr-is-merged
home
.....(생략).....
wslbejook
mnt
usr
sys
wslNHghlk
opt
ANDREW@BOOK-1122:/$ cd sys
ANDREW@BOOK-1122:/sys$ pwd
/sys
ANDREW@BOOK-1122:/sys$ ls
.
..
kernel
class
devices
dev
fs
bus
firmware
block
module
ANDREW@BOOK-1122:/sys$ exit
```

잘 나오는것을 확인 할 수 있다. 이제 디렉토리 관련한 기본적인 기능들을 구현했으니, 우선은 commit을 하겠다.

4/16: 명령어 함수화

여느때처럼 개념을 공부한 후, 앞으로의 과제를 어떻게 수행하면 좋을지 생각을 해보았다. 그러던 중, 다중 명령어를 구현하려면 결국에는 재귀함수가 필요하다는 것을 알게 되었다. 1차적으로 고안한 알고리즘(의사 코드)은 다음과 같다.

```
command_process(command)
        if command have “A”
                do A
        else if command have “B”
                do B
                .....

        if command have ‘;’
                split command by ‘;’
                command_process(command1)
                command_process(command2)
```

기존에는 각 명령어를 단순히 몇 줄의 코드로 개별적으로 구현하는 방식에 그쳤다면, 이번에는 명령어 처리 방식을 하나의 함수로 통합하여 구조적으로 개선하였다. 새롭게 설계한 이 함수는 입력으로 전달받은 커맨드를 분석한 뒤, 다중 명령어가 포함되어 있을 경우 이를 개별 명령어로 분리하고, 각각을 재귀적으로 다시 함수에 전달하여 처리하도록 하는 것이다.

사실 개념 영상을 보면서 재귀함수가 언급되는 것을 보고 “굳이?” “이게 필요한 개념인가?”라는 생각이 들었었는데, 과제를 진행하다보니 결국에는 강의에서 선배님께서 의도하신 큰 뜻을 알게 되었다.

재귀함수는 항상 코딩테스트 문제를 풀거나 학교 과제를 할때만 썼었는데, 마침네 실제로 무언가를 만드는데 사용할 기회가 오니 참으로 설렌다.

우선 명령어들을 하나로 묶은 후, int값을 반환하는 함수 command_process(char* str)을 만들어 안에 넣어줬다. void형태가 아닌 int값을 반환하는 이유는, 해당 함수가 bash함수의 while문 안에 들어가져 있는데, 기존의 코드를 그대로 쓰면 이전과 달리 exit함수를 입력해도

command_process함수만 끝날 뿐 그것이 담긴 while문이 끝나진 않았다. 따라서 exit가 입력된 경우 command_process가 1을 반환하도록 하고, if(command_process(command) == 1) break; 코드를 통해 command_process를 호출하여 실행함과 동시에 그것이 1을 반환하면 조건문을 탈출할 수 있도록 구현하여, 하나의 줄에 두개의 기능을 담은 일석이조 코드를 완성했다. commit을 하겠다.

4월 16(2): 다중명령어 의사 코드 작성
재귀함수의 형태를 고안했지만, 구체적인 방식까지 생각해낸것은 아니었다. 그래서 곰곰히 고민을 해본 결과, 아주 좋은 아이디어가 떠올랐다. 위의 의사코드를 그대로 활용해본다고 가정해보자. 만약 command가

ls ; pwd ; cd sys ; exit

라고 해보자. 만약 이것을 모든 ;를 기준으로 전부 나누어 ls, pwd, cd, exit을 나누게 된다면, 몇개일지도 모르는 다중명령의 개수들과 ;를 모두 탐지하고 나누는데 있어서 코드와 알고리즘이 꽤나 복잡해질 것이다.

그러나 만약 입력되는 커맨드에서의 첫 ;만 탐지하고 그것을 기준으로 앞,뒤를 나눈다면?
그렇다면 위의 명령어는
ls / pwd ; cd sys ; exit 으로 나뉠꺼고, 후자를 command로 입력한다면 해당 명령어는 pwd/ cd sys ; exit으로, 또 나뉘고 입력된 마지막 명령어는 cd sys/ exit으로 나뉘게 된다. 여기서 얘기하고자 하는 것은, ;를 기준으로 전부 나누는 것이 아닌, 입력된 명령어의 첫번째 ; 만을 기준으로 나누고, 이것을 재귀적으로 호출한다면 더 효율적으로, 자동적으로 명령어들이 나뉘어서 실행되게 될 것이다.

이 기능은 문자열에서 특정 패턴을 찾아주는 strstr함수를 사용하여 만들수 있을것 같다. strstr함수가 패턴을 찾으면 문자열에서 패턴이 시작하는 위치의 주솟값을 반환하는 것을 고려하여, 구현해본 코드는 다음과 같다.

```
else if (strstr(str,";")! =NULL) {
                char* bookmark = strstr(str,";");
                char front_command[COMMAND_SIZE],
back_command[COMMAND_SIZE];
```

```
                int bookmark_length = bookmark - str;
                strncpy(front_command,str,bookmark_length);
                front_command[bookmark_length] = '\0';
                strcpy(back_command,bookmark+1);

                command_process(front_command);
                command_process(back_command);

        }
```

만약 ;를 찾아 주소값을 반환하면, 그것을 저장하고, 그 위치를 기준으로 앞, 뒤를 복사해 저장한 후, 재귀적으로 실행하는 것이다. 물론, 이때 공백으로 인해 문제가 생기면 안되므로

```
int i = 0;
    while (str[i] == ' ') {
        i++;
    }
    if (i > 0) {
        memmove(str, str + i, strlen(str + i) + 1);
    }
       // exit 구현
       if (strcmp(str, "exit") == 0) {
               printf("BASH를 종료합니다.\n");
               return 1;
```

라는 코드도 추가했다. 실행을 하니

```
aa@aa:/$ pwd;pwd
/
/
aa@aa:/$ pwd;pwd;pwd
/
/
aa@aa:/$ aa@aa:/$
```

처럼 한개의 ;에는 잘 작동하지만 두개 이상부터는 버그가 걸린다. 지금

시각이 새벽 3시 57분인데, 아무리 머리를 쥐어짜도 왜 안되는지 전혀 모르겠다. 찜찜하지만 일단은 하루밤을 자고 생각을 해봐야겠다. 우선은 commit.

4/20: 입력 문자 공백 제거
시험공부에 바쁜 요새, 틈틈히 분석하며 무엇이 문제인가 고민을 해보았지만 아무래도 답이 나오지 않는다. 다만 요 몇일 사이에 수정한것이 있다면, 만약 입력의 시작 혹은 끝부분이 공백이라면 코드가 정상작동하지 않는것을 발견하였고, 그것에 대해 문자열의 앞,뒤 공백을 없애주는 함수를 <string.h>에 있는 memmove를 활용해 만들었다.

```
void remove_space(char*b) {
    int i=0;
    while(b[i] ==' ') i++;
    if(i >0) memmove(b, b +i, strlen(b +i) +1);
    int len =strlen(b);
    while(len >0 &&b[len -1] ==' ') {
        b[len -1] ='\0';
        len--;
}
}
```

이를 사용한 결과
dd@dd:/$ pwd
/
dd@dd:/$ pwd ; pwd
/
/
dd@dd:/$ dd@dd:/$
공백 문제는 해결했다.
그러나 여전히 위의 문제에 대한 답이 나오지를 않는다. 이 부분에 대한 답을 찾지 못한다면 다른 다중명령어나 파이프라인, 다중 파이프라인을 구현하는 것은 물 건너간 것이기에 이 부분을 완벽히 해결해야 할 것 같다. 아니면 백그라운드 구현 먼저 만들어봐도 좋을 것 같다.
우선은 commit을 하겠다.

4/21: 문제해결; 다중명령어 ‘;’ 구현
한 순간에 모든 문제를 해결 해버렸다.
문제의 원인은 간단했다.

```
#define COMMAND_SIZE 10
```

command_size 가 10으로 정의되어있었다.....
본래 100 아니면 50으로 설정 되어있던게 실수로 10으로 바꼈고, 이에 따라 pwd;pwd;pwd처럼 명령어가 길어지며 버퍼 오버플로우를 일으켰던 것이다(...)
이것을 256으로 매우 넉넉하게 고쳤다.

이것과 함께 부가적인 문제점을 해결했는데, 다중명령어 탐지 기능이 if이고, 아래에 명령어 수행도 if로 시작해 다중명령어를 탐지해 나누어도 끝나지 않고 아래에 명령어 수행 부분으로 넘어가기에 혹시 모를 문제발생을 예방하기 위해 명령어 수행 부분을 else if로 시작하게끔 만들었다.

이것을 실행해 테스트 해보면
aaaa@aaaa:/$ pwd;pwd;pwd
/
/
/
aaaa@aaaa:/$ pwd;pwd;pwd;pwd
/
/
/
/
aaaa@aaaa:/$
매우 잘 나온다...
commit을 하겠다.

4/24: 다중명령어 ||, && 구현
이제 다른 다중명령어들을 구현할 차례이다. 어떻게 구현을 하면 좋을까를 고민해봤는데, 역시 토큰을 사용해야 할 것 같다. 사실 ;를 구현하기 위해

리서칭하고 다닐때, 대부분의 경우가 토큰을 이용한 파싱 시스템으로 구현을 했는데, 해당 개념에 대해 생소하기도 하고 무엇보다 쓰지 않고도 구현을 할 수 있다고 자신했기에 사용하지 않았었다. 그러나 이번에는 리서칭을 해보니, 토큰을 활용하지 않는다면 무지막지하게 복잡해질 것이 너무 뻔했다. 그래서 알고리즘을 약간 수정했는데, 수정된 알고리즘은 다음과 같다.

다중명령문 ; 우선 분리(재귀함수의 인자에 ;가 전달되지 않을때까지)
문자열 파싱을 통해 &&과 || 감지시 입력된 문자열을 파싱, 토큰에 저장
우선순위에 대한 알고리즘을 활용해 순서가 맞게 명령어 실행

현재 첫번째 부분까지는 구현을 하였으니, 토큰을 활용해 아래부분들을 만들어보자!

우선은, 바로 소스코드에 작성을 하기에 앞서 파싱과 토큰 시스템, 연산의 우선순위들을 다른 c컴파일러에서 테스트를 해보았다.

이후 기존의 command 함수에서 몇가지 기능을 덧붙였는데, 수행 성공 여부에 따라 그것을 판별할 수 있는 특정 int값을 반환하게 하였다. ||과 &&가 쓰이면, 결국 앞에 오는 명령어의 수행 성공 여부가 중요해지기 때문이었다. 따라서 수행 성공시 1을, 실패시 0를 반환하게 하였다(exit 제외). 이것을 주석과 함께 추가하였다.

4/25: 파싱 알고리즘 개발
하룻동안 다른 c컴파일러를 이용해 특정 문자열에 대해 &&과 ||를 감지해 문자열을 파싱하고 토큰 배열에 저장하는 알고리즘을 만들었다.

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "abcd && EFGH||1234 && adsadsad ||       1283
9032189       ";
    char token[100][100];
    int pars = 0;
```

```
    if (strstr(str,"&&") != NULL || strstr(str,"||") != NULL) {

        char *ptr;
        char *start = str;
        int len;


        for(int i = 0; i < strlen(str);i++) {
            if((str[i] == '&' && str[i+1] == '&') || (str[i] == '|' &&
str[i+1] == '|')){
                ptr = &str[i];
                len = ptr - start;
                strncpy(token[pars],start,len);
                start = ptr + 2;
                pars++;
                //다중명령어 저장
                if(str[i] == '&') strcpy(token[pars],"&&");
                else strcpy(token[pars],"||");
                pars ++;
                //여기까지는 다중명령어 기준 '앞' 명령어 들만 저장하는
코드임.
            }
        }
        //마지막 다중명령어 기준 '뒤' 명령어 저장
        strcpy(token[pars],start);
        pars++;
    }
    for (int i = 0; i < pars; i++) {
        for (int j = 0; j < 20; j++) {
            printf("%c", token[i][j]);
        }
        printf("\n");
    }
```

```
    return 0;
}
```

지피티, 딥시크, 제미나이 등 어떠한 AI의 개입 없이 순수히 내 대가리로 만든 알고리즘의 코드이다. 실행을 하면

```
abcd
&&
 EFGH
||
1234
&&
 adsadsad
||
      1283    9032189
```

잘 나오는 것을 확인 할 수 있다.
사실은 원래 AI로 어떻게 구현하는건지 맛이라도 볼려고 했는데, 얘네가 작성한 코드들이 하나같이 복잡하고 무슨소린지도 모르겠는데 작동마저 안해서 ”내가 해도 이것보단 잘 짜겠다“는 마인드로 반나절동안 포인터 개념을 복기하며 이악물고 만들었다. 그래서 지금 코드를 완성하고 결과가 제대로 출력한 것을 확인한 지금, 기분이 너무 뿌듯하다.

이제 다중명령어의 연산 과정과 연산 우선순위에 대한 알고리즘을 만들고, 해당 알고리즘에 위의 코드를 대입하면 다중명령어 구현은 성공할 것 같다. 우선은 해당 코드를 주석문으로 vs에 저장한 뒤 commit을 하겠다.

4/29: 백그라운드 프로세스

현재 시험기간인데, 시험 공부와 다중연산자 알고리즘 제작을 병행하는 것은 아무래도 무리일것 같다고 생각한다. 따라서 시험이 끝나기 전까지 다중 연산자보다는 다른 기능들부터 구현하고자 한다.

우선 백그라운드 프로세스를 구현해 보겠다. 문자열의 마지막에서 &를 발견한다면, 그 앞까지의 문자열을 command_process에 집어넣는 코드를

작성하였다.

```
else if(strrchr(str,'&')!=NULL){
        char*bg =strrchr(str,'&');
        char bgcmd[50];
        int bg_length =bg -str;
        strncpy(bgcmd,str,bg_length);
        bgcmd[bg_length] ='\0';
        command_process(bgcmd);
}
```

정상적으로 작동하였다. 이제 이것이 백그라운드에서 작동하게끔 만들겠다.
인터넷 블로그 글과 이런저런 정보글들을 토대로 작성한 코드는 다음과 같다.

```
else if(strrchr(str,'&')!=NULL){
        char*bg =strrchr(str,'&');
        char bgcmd[50];
        int bg_length =bg -str;
        strncpy(bgcmd,str,bg_length);
        bgcmd[bg_length] ='\0';

        background =1;
        //자식 프로세스 생성
        pid_t pid =fork();

        if(pid <0) {
            perror("fork 실패");
}
        else if(pid ==0) {
            // 자식 프로세스: 실제 명령 실행
            command_process(bgcmd);
}
        //부모 프로세스
        else
            if(background) {
                //실행 여부 및 자식 프로세스의 식별 ID 출력
                printf("백그라운드 실행중\t[pid: %d]\n",pid);
}
}
}
```

조건문은 한개인데 fork로 인해 자식, 부모에서 실행되어 조건문이 총 두번

수행된다는 것이 신기했다. 테스트를 해보니
dd@dd:/$ pwd &
백그라운드 실행중 [pid: 15557]
dd@dd:/$ /
dd@dd:/$ pwd
/
dd@dd:/$ cd sys &
백그라운드 실행중 [pid: 15699]

처럼 작동은 잘 하는데.....이전에 나타난 프롬프트 겹침 문제가 발생한다. .
Ai들에게 디버깅에 대한 조언을 물어보니, 부모 프로세스가 자식 프로세스의 실행이 완료될 때 까지 기다리지 않고 바로 프롬프트를 입력받으려 해서 그렇다고 한다. 따라서 waitpid(pid,NULL,0)을 추가해 자식 프로세스를 기다리게끔 하였다. 그랬더니 결과가
ff@ff:/$ pwd &
백그라운드 실행중 [pid: 19988]
/
ff@ff:/$ ff@ff:/$ ^C
자식 프로세스가 출력되는 것은 고쳤지만, 프롬프트 중복 입력은 해결하지 못했다.
근데 생각을 해보니 실행을 한 후 return을 하지 않은 것과, 자식 프로세스가 작동 후 종료하는 것을 하지 않았다. 이것을 고치고 실행을 해보니
Enter username:dd
Enter hostname:dd
dd@dd:/$ pwd &
백그라운드 실행중 [pid: 24106]
/
dd@dd:/$
잘 뜨는 것을 알게 되었다. commit을 하겠다!

5/4: 기타 명령어 구현
만약 command process로 받은 명령어에 대한 대응 조건문 없다면, /bin/ 폴더 어딘가에서 자동으로 찾아 부모/자식 프로세스로 나누어 execlp로 실행시킬 수 있지 않을까? 라는 생각이 들었다. 그래서 원래 else부분을

이것으로 바꾸기 위해 이곳 저곳 찾아봤고, 찾아보며 완성시킨 코드는 다음과 같다.

```
else{// 이 외의 단일 명령어 실행
        pid_t pid =fork();
        if(pid ==-1) {
            perror("fork");
            return -1;
}
        if(pid ==0) {// 자식 프로세스
             execlp("/bin/sh", "sh", "-c", str, NULL);// 단일 명령어 직접 실
행
            perror("execlp");
            exit(0);
} else{
            waitpid(pid,NULL,0);// 자식 프로세스 종료 대기
            return 1;
}
}
```

이전의 백그라운드 코드를 거의 배끼다싶이 만든 코드이다. 한번에 완성한것은 아니고, 인터넷으로 execlp부분을 완성시키려 3번정도의 시행착오를 거쳤다.
실행 결과
dd@dd:/$ date
Sun May 4 21:36:25 KST 2025
dd@dd:/$
잘 나오는 것을 알 수 있다. 우선은 commit을 하겠다.

5/6: 파이프라인 구현
이제 파이프라인을 구현해 볼 것이다. 파이프라인 구현에 앞서 생각해낸 대략적인 알고리즘은 다음과 같다.

만약 |이 있다면 입력 문자열 파싱
|를 토대로 커맨드 분리한 후 토큰 생성
토큰 개수만큼 파이프 & 자식 프로세스 생성
상호작용 후 결과 출력

원래는 명령어 다루는것을 온전히 command process 재귀호출로 구현하려

했는데, 생각을 해보면 command_process는 단일 명령문 출력만 고려한지라 자식 프로세스 생성 및 파이프 연결이 곤란 할 것 같았다. 따라서 어그제 기타명령어 구현할 때 사용했던 execlp()를 사용하고자 한다.
또한 파싱을 원래 짯던 알고리즘을 쓰기로 했지만, 코드가 길기도 하고 strtok라는 토큰을 해주는 함수를 발견해서 해당 함수를 사용하기로 했다.
만들어본 첫번째 코드는 다음과 같다

```
char *token =strtok(str, "|");
        char *commands[COMMAND_SIZE];
        int pipe_fd[COMMAND_SIZE -1][2];
        int num_commands =0;
        while(token !=NULL &&num_commands <COMMAND_SIZE) {
            commands[num_commands++] =token;
            token =strtok(NULL, "|");
}
        // 개수만큼 파이프 생성
        for(int i =0; i <num_commands -1; i++) {
            if(pipe(pipe_fd[i]) ==-1) {
                perror("pipe");
                exit(EXIT_FAILURE);
}
}


        for(int i =0; i <num_commands; i++) {
            pid_t pid =fork();
            if(pid ==-1) {
                perror("fork");
                exit(EXIT_FAILURE);
}
            if(pid ==0) {
                // 이전 프로세스에서 입력 받기
                if(i >0) {
                    dup2(pipe_fd[i -1][0], STDIN_FILENO);
                    close(pipe_fd[i -1][0]);
}
                // 다음 프로세스로 출력 보내기
                if(i <num_commands -1) {
                    dup2(pipe_fd[i][1], STDOUT_FILENO);
```

```
                close(pipe_fd[i][1]);
}
        // 파일 디스크립터 닫기
                for(int j =0; j <num_commands -1; j++) {
                    close(pipe_fd[j][0]);
                    close(pipe_fd[j][1]);
}

                // 각 명령어 실행
                char *args[]={"/bin/sh", "-c", commands[i], NULL};
                execvp(args[0], args);
                perror("execvp");
                exit(0);
}
}
```

필요한 변수들을 선언했고, num_commands를 토큰 배열의 인덱스 변수로 사용하기로 했다. 오늘 c언어 수업시간에 배열과 인덱스를 배웠는데, 조금더 공부를 해보니 배열[인덱스 ++]처럼 인덱스?부분에서 전위 또는 후위 연산자 사용이 가능한 것을 알게 되었다. 따로 num_command ++을 하지 않고, 바로 후위연산자로 작성해보았다.

토큰을 저장한 후, 토큰의 개수 -1 개만큼 파이프를 생성했고, 명령어의 개수만큼 fork를 해주어 자식 프로세서를 만들었다.
이후 dup2함수와 STDIN_FILENO와 STDOUT_FILENO를 통해 각각의 자식프로세스들이 결과값을 주거니 받거니 할 수 있도록 하였다.
사실 이부분을 공부하면서 신기했다. 함수를 통해서 표준 출력으로 결과값을 터미널에 출력하는 것이 아닌 다음 프로세스에 파이프를 통해 보내고, 해당 프로세스는 터미널을 통해 입력 받는 것이 아닌 파이프를 통해 입력을 받는것, 그리고 이것이 dup2함수를 통해 생각보다도 간단히 이루어질 수 있다는 것이 신기했다. (과제가 끝나고 여유가 생긴다면 이 부분을 좀 더 공부해볼지도..??)

이후 기존에 사용했던 파이프(정확히는 파일 디스크립터)를 닫아주었고, 명령어가 실행이 될 수 있도록 기타명령어에서 구현한 코드를 따와 조금 수정한 후 넣어주었다. 이를 실행해보면

```
dd@dd:/$ ls | sort
bin
bin.usr-is-merged
boot
dev
etc
home
init
lib
lib.usr-is-merged
lib64
lost+found
media
mnt
opt
proc
root
.....
wslNMNlNp
wslbejook
wslhJjmlk
wsljJppOp
wsljkcDPp
wslphMDpk

dd@dd:/$ ls | sort | grep sys
sys
```

잘 나오는 것을 확인 할 수 있다.
근데 오히려 한번에 너무 잘 나오니 괜히 불안해졌다. 하는 수 없이 해당 코드를 들고 AI에게 디버깅을 부탁하니, 코드가 정상작동 할 수는 있지만 부모 역시 close를 통해 파일 디스크립터를 닫아줘야 한다고 한다. 그 이유는 fork와 파이프 생성을 하면 부모 역시 FD를 가지게 되는데, 때문에 명령어를 수행하는 자식의 FD만 닫아도 부모의 FD가 살아있기에 파이프가 살아있다고 판단할 수도 있다고 하며, 그렇게 된 경우 좀비 프로세스 혹은 무한한

기다림이 발생할 수 있다고 했다. 이 말을 듣고, 똑같이 부모 프로세스에게도 파일 디스크립터를 닫는 코드와 자식 프로세스가 끝날때까지 기다려주는 코드를 작성했다.

```
for(int i =0; i <num_commands -1; i++) {
            close(pipe_fd[i][0]);
            close(pipe_fd[i][1]);
}
        for(int i =0; i <num_commands; i++) {
            wait(NULL);
}
        return 0;
}
```

실행을 해보니 결과가 이전과 같이 잘 나온다. commit을 하겠다.

5/8: 다중명령어 ||, && 구현

드디어 다중명령어만 구현하면 과제의 조건들을 모두 구현할 수 있게 된다. 근데 구현에 앞서 난관이 펼쳐졌다.

“여러개의 다중명령어가 섞인 명령어면 각각의 명령어들의 수행 결과를 출력없이 저장하고 연산해야하는데, 그렇다면 어떡하지?”

처음에 생각하고 구현하려던 방식은, 우선 모든 명령어를 수행하고, 반환값을 저장한 뒤, 반환값인 1,0들과 다중연산자들을 조합하여 연산결과를 출력하려 했다. 근데 이렇게 한다면, 우선 불필요한 연산들이 많아지고 코드가 복잡해질뿐더러, 다중연산자를 구현하는 취지에 어긋?난다고 생각이 들었다.

그러던 중, 결과의 출력을 가리면서 여러 명령어들의 수행결과를 연산하는 것을 다중 파이프라인에서 구현했다는 것을 깨닳았다. 그래서 해당 코드를 가져와 조금 수정하여 다중명령어 코드를 짜기로 하였다.

&&과 ||의 연산 순서(&&가 우선인것)를 고려해서 짜본 코드는 다음과 같다.

```
else if(strstr(str, "||") !=NULL ||strstr(str, "&&") !=NULL) {
        char *commands[COMMAND_SIZE];
        int num_commands =0;
        int success =0;
        // && 연산자 우선 분리리
        char *token =strtok(str, "&&");
        while(token !=NULL &&num_commands <COMMAND_SIZE) {
            commands[num_commands++] =token;
```

```
            token =strtok(NULL, "&&");
}
        for(int i =0; i <num_commands; i++) {
            char *sub_commands[COMMAND_SIZE];
            int sub_num =0;
            // || 연산자 분리리
            token =strtok(commands[i], "||");
            while(token !=NULL &&sub_num <COMMAND_SIZE) {
                sub_commands[sub_num++] =token;
                token =strtok(NULL, "||");
}
            for(int j =0; j <sub_num; j++) {
                success =command_process(sub_commands[j]);
                // OR 연산자 하나라도 성공시 탈출
                if(success) break;
}
            // AND에서 실패시 전체 중단
            if(!success) break;
}
}
```

우선순위를 고려해 &&를 기준으로 먼저 분리해 commands로 분리해주었다. 이후, 분리된 코멘드를 다시 선회하며 ||를 탐색해주었고, 이것을 따로 sub_commands에 저장해주었다. strtok의 로직에 따라 만약 ||가 있으면 그것이 분리되서 저장되고, ||가 없다면 그 자체로 저장이 된다.
이후 command_process를 재귀호출하며, 이전에 설정해 둔 것처럼 실행 성공시 1을, 아닐시 0을 반환하게 한 것의 반환값을 success에 저장하고, 해당 success의 값을 토대로 연산을 멈출지 말지를 계산한다. 이를 실행해보면 다음과 같다.

```
dd@dd:/$ ls && echo "Success" || echo "Failure"
bin                dev  init                lib64       mnt  root
sbin.usr-is-merged  sys  var          wslIDkiMp  wslbejook  wsljkcDPp
bin.usr-is-merged  etc  lib                 lost+found  opt  run
snap               tmp  wslENlFpk  wslNHghlk  wslhJjmlk
wslphMDpk
boot               home  lib.usr-is-merged  media       proc  sbin
```

srv usr wslEOdlOp wslNMNlNp wsljJppOp
Success
dd@dd:/$
리눅스 명령어 ECHO를 활용한 예시인데, 잘 작동하는 것을 알 수 있다.
또한 Makefile도 작성하였다.

```
CC= gcc
CFLAGS= -g -Wall
TARGET= main
SRC= main.c
OBJ$(SRC:.c=.o)
all: $(TARGET)
$(TARGET): $(OBJ)
    $(CC)$(CFLAGS)-o $@$^
$(OBJ): $(SRC)
    $(CC)$(CFLAGS)-c $<
clean:
rm -f $(OBJ)$(TARGET)
```

fury48@BOOK-5EBM7U67HE:~/cykor_week2$ make
gcc -g -Wall -c main.c
gcc -g -Wall -o main main.o
fury48@BOOK-5EBM7U67HE:~/cykor_week2$ ./main
잘 작동한다.
COMMIT을 하겠다!

5/12: 유저네임과 호스트네임 입력 버퍼 오버플로우 대비

마지막으로 코드를 둘러보며 해결할 문제를 찾던도중, 명령어를 입력받는 것은 fgets로 입력받기에 오버플로우 걱정이 없지만 처음에 scanf로 유저네임과 호스트 네임을 받을때 50글자를 넘어간다면 오버플로우가 발생 할 수 있어 이것을 고쳐보기로 했다.

다만 fgets로 수정을 하니, 왠지 모르게 계속 입력을 3번 받아야 넘어가게 되었다. 하는 수 없이 scanf에서 %49처럼 하는 식으로 수정했다. (마지막에 널문자가 추가 되어야 하므로 사이즈를 50 -1 로 했다)