2nd READ ME

이번 과제 제출 다음 4가지 파일들로 구성되어 있습니다.

mainstream.c	리눅스의 전반적인 명령어 실행을 위한 코드들이 담겨있는 c파일
directory_struct.c	디렉토리 구조를 표현한 c파일
directory_struct.h	Struct로 표현한 디렉토리 구조를 mainstream과 directory_struct에서 참조할 수 있게 하기 위해 만든 헤더파일

이번 2nd 보고서는 ***mainstream.c***에 대한 내용만을 집중적으로 담고 있습니다.

[개발한 리눅스 내 환경 정보]

*user name = oepickle

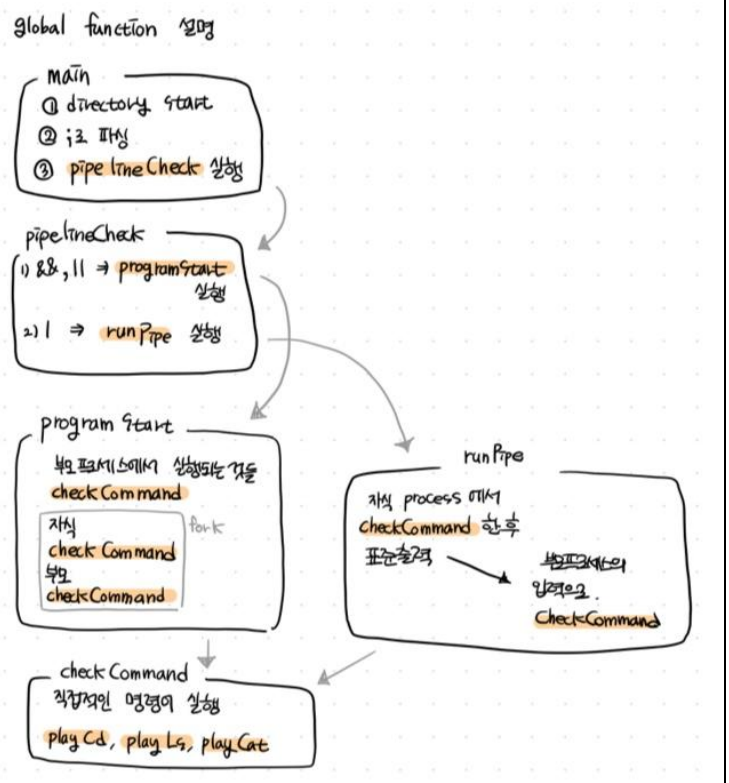
*host name = UNI-CTJ

[구현한 명령어들]

;, &&, ||, |

Cd(.., .까지), echo, pwd, exit, cat, ls

[mainstream 흐름 구조도]



mainstream.c

Step1. 전역변수 설명

```c
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "directory_struct.h"

//------------------------------
//-------------- global values -
#define VALUE_SIZE 256

char* username = "oepickle";
char* hostname = "UNI-CTJ";

Node* nowNode;
File* nowFile;
```

Char 배열을 사용할 일이 많아 #define으로 변수 사이즈를 256으로 선언해 배열 사이즈를 모두 VALUE_SIZE로 맞춤.

Char* username	유저 이름을 담는 전역변수
Char* hostname	호스트 이름을 담는 전역변수

Node* nowNode	현재 위치한 디렉토리 노드를 나타내는 struct 포인터
File* nowFile	현재 읽을 파일의 노드를 나타내는 struct 포인터

*#include "directory_struct.h"를 했기에
바로 Node* ~, File* ~ 을 정의할 수 있었음.

```
char d_path[VALUE_SIZE];
char userInput[VALUE_SIZE];
char catOutput[VALUE_SIZE];

int isPipeMode = 0;
int isBackgroundMode = 0;
```

Char d_path	현재 경로를 저장하는 문자열 Ex: /home/oepickle 이런 식의 문자열을 저장하는 역할
Char userInput	User의 명령을 입력받아 저장해두는 공간(전역 변수)
catOutput	Cat 실행 시 output으로 내보낼 문자열을 저장하는 문자열 공간
isPipeMode	파이프라인(|)의 실행 여부를 알려주는 flag의 역할을 하는 변수
isBackgroundNode	백그라운드 실행(&)의 여부를 알려주는 flag역할을 하는 변수

Step2. 전역함수 설명

```
//------------- global function ----------------------
int pipelineCheck(char* token);
void runPipe(char* left, char* right);
int programstart(char* cmd);
void checkCommand(char* command);

int playCd(char* str);
void playLs();
void playCat(char* filename, char arr[VALUE_SIZE]);
void trim(char* str);
//----------------------------------------------------
//----------------------------------------------------
```

[각 함수들 역할 정리표]

Step1 **main**	Directory start 실행 -> 디렉토리 구조 빌드
	;를 기준으로 파싱
	파싱한 문자열을 pipelineCheck로 넘김
Step2 **pipelineCheck**	받은 문자열 속에 어떤 명령어가 숨겨져 있는지 체크하는게 이 함수의 main 역할
	&(background실행) 체크
	&&, ||, | 체크 *&&, || -> 각각을 기준으로 파싱하여 pipelineCheck로 재귀 *| -> 파싱하여 runpipe로 보냄 *아무것도 없다면 -> programstart
Step3(1) **programStart**	Cd, echo, exit : fork 호출 전 실행
	Fork로 나눠 자식프로세스 실행
	부모 프로세스 실행 **이때 실행한다는 건 checkCommand로 넘긴다는 이야기*
Step3(2) **runPipe**	Fork로 자식과 부모를 나누어 실행
	자식 프로세스의 출력을 부모의 입력으로 받아 실행하는 역할을 함
	오직 |가 들어간 문자열이 input으로 들어왔을 시에만 실행되는 함수
PlatCd	Cd를 실행하는 함수 checkCommand 안에 들어 있음
playLs	Ls를 실행하는 함수 checkCommand 안에 들어 있음

playCat	Cat을 실행하는 함수 checkCommand 안에 들어 있음
trim	문자열의 앞 뒤 공백들을 지우는 함수

Step3. Main 함수

```
int main() {
    directoryStart();
    strcpy(d_path, root->name);
    nowNode = root;

    while (1) {
        printf("%s@%s:%s$ ", username, hostname, d_path);
        fgets(userInput, sizeof(userInput), stdin);
        userInput[strcspn(userInput, "Wn")] = 'W0';

        if (strcmp(userInput, "exit") == 0) break;

        if (strchr(userInput, ';')!=NULL){
            char* token = strtok(userInput, ";");
            while (token != NULL) {
                while (*token == ' ') token++;
                pipelineCheck(token);
                token = strtok(NULL, ";");
            }
        }
        else {
            pipelineCheck(userInput);
        }

    }
}
```

1) 디렉토리 구조 생성

우선 directoryStart를 실행해서 directory_struct.c에서 만들어 놓은 디렉토리 구조와 파일 구조를 빌드함.

2) 입력 루프 실행

그 후 while문 안에서 무한루프를 통해 사용자 입력을 반복적으로 받으려고 함. 프롬포트는 [username@hostname:현재경로$_]로 출력되게 의도한 것.

3) 사용자 입력 처리

- Fgets()로 사용자 입력을 받고 마지막에 붙는 개행 문자를 strcspn으로 제거
- 입력이 exit인 경우 루프를 종료하여 셸을 끝내기 위해 userInput이 exit이랑 같은지 체크함.
- 입력 문자에 세미콜론이 포함된 경우 해당 문자열을 여러 명령어로 나누어 개별적으로 pipelineCheck에 전달.
- 단일 명령어일 경우에는 그대로 pipelineCheck에 전달

***입력을 받을 시 fgets을 사용한 이유**

- 버퍼 오버플로우 방지 목적:

fgets()는 입력최대길이를 제한할 수 있기에 scanf()나 gets() 보다 안전하다고 판단. (sizeof(userInput)만큼만 읽기 때문)

Step4. pipelineCheck

```
//------------ &&, ||, & check ------------
int pipelineCheck(char* token) {
    trim(token);
    if(strlen(token)==0) return 0;

    char temp[100];
    strncpy(temp, token, sizeof(temp) - 1);
    temp[sizeof(temp) - 1] = 'W0';
    //-------------back ground play-------
    isBackgroundMode = 0;
    char* cmd = token;
    int len = strlen(cmd);
```

이 함수는 사용자가 입력한 명령어(token)에 대해 논리 연산자(&&, ||), 파이프(|), 백그라운드 실행(&) 등을 해석하고 동작을 수행하는 명령어 해석기 역할을 함

사용자 입력을 여러 조각으로 나누어 처리하기 위해 재귀적 구조로 코딩했으며, 복합 명령어 실행 또한 가능하도록 의도를 가지고 설계함

전처리 단계

입력 명령어 문자열의 앞뒤 공백을 제거하는 trim() 함수를 호출. If 문자열이 비었다면 처리하지 않고 함수 종료

```
if (len > 0 && cmd[len - 1] == '&') {
    isBackgroundMode = 1;
    cmd[len - 1] = '\0';
    len = strlen(cmd);
    while (len > 1 && cmd[len - 2] == ' ') {
        cmd[len - 2] = '\0';
        len--;
    }
    int val = pipelineCheck(cmd);
    return val;
}
//-----------------------------------------
char* andOper = strstr(temp, "&&");
char* orOper = strstr(temp, "||");
char* pipeOper = strstr(temp, "|");
```

백그라운드 명령 처리 (&)

명령어 끝이 &로 끝나면 isBackgroundMode를 1로 설정하고, &와 그 앞의 공백을 제거한 후 같은 함수에 재귀 호출하여 실제 명령어를 다시 처리.

논리 연산자 분기 (&&, ||)

&&가 문자열 내에 존재하는 경우, 좌측 명령어(left)가 성공적으로 수행되었을 때만 우측 명령어(right)를 수행.
left 명령 실행 결과가 성공이라면(0이 아니면) right도 실행.

```
if (andOper != NULL) {
    *andOper = '\0';
    char* left = temp;
    char* right = andOper + 2;
    trim(left);
    trim(right);
    if (pipelineCheck(left) != 0) {
        pipelineCheck(right);
    }
}
else if (orOper != NULL) {
    *orOper = '\0';
    char* left = temp;
    char* right = orOper + 2;
    trim(left);
    trim(right);
    if (pipelineCheck(left) == 0) {
        pipelineCheck(right);
    }
}
else if (pipeOper != NULL) {
    *pipeOper = '\0';
    char* left = temp;
    char* right = pipeOper + 1;
    trim(left);
    trim(right);
    runPipe(left, right);
}
else {
    int val = programstart(temp);
    return val;
}
```

||가 문자열 내에 존재하는 경우, 좌측 명령어 실행이 실패했을 때에만 우측 명령어를 실행.
left 명령 실행 결과가 실패라면(0일 경우) right를 무시.

파이프 처리 (|)

| 연산자가 문자열 내에 있을 경우 명령어를 좌/우로 분리하고, runPipe() 함수를 통해 파이프 연결 실행.
이때도 trim()으로 공백을 제거한 뒤에 runPipe를 돌림으로써 공백으로 인해 오류가 발생할 일을 미리 방지하고자 함.

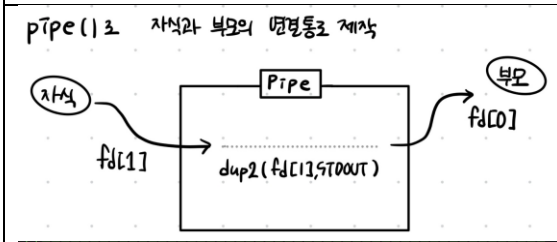
일반 명령 실행

위 조건에 해당하지 않는 명령어는 단일 명령으로 간주되며, programstart() 함수를 통해 실행.
그 결과값(int)을 상위 논리 연산자 분기문에서 활용할 수 있도록 리턴하게 설계함.
->후에 flag 역할을 해줄 것

***재귀적 설계 이유**

이 함수에서 재귀적 설계를 한 데에 애를 좀 썼습니다.
복합 명령어(ls && echo hi || cat)를 정확히 처리하기 위해 **분할 → 재귀 호출** 구조가 가장 적합하다고 생각했습니다.
명령어를 분할하고 좌우 명령어를 pipelineCheck()에 다시 넘기는 방식으로, 여러 개의 논리 연산이 포함된 명령도 계층적으로 처리할 수 있게끔 만들었습니다.

Step5. runPipe




```
//---------------check | pipeline---------------------
void runPipe(char* left, char* right) {
    int fd[2];
    pipe(fd);

    pid_t pid = fork();

    if (pid == 0) {
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        close(fd[1]);
        isPipeMode = 1;
        trim(left);
        checkCommand(left);
        _exit(0);
    }
    else {
        close(fd[1]);
        wait(NULL);
        char buf[400];
        ssize_t n = read(fd[0], buf, sizeof(buf) - 1);
        close(fd[0]);
        if(n>0) buf[n]='₩0';
        else buf[0]='₩0';
        buf[strcspn(buf, "₩n")] = '₩0';
        char str[500];
        trim(buf);
        trim(right);
        snprintf(str, sizeof(str), "%s %s", right, buf);
        isPipeMode = 0;
        checkCommand(str);
    }
}
//-----------------------------------------------------
```

runPipe(char* left, char* right)

| 연산자를 구현하기 위한 함수로, 좌측 명령어(left)의 출력을 우측 명령어(right)의 입력으로 전달하는 역할을 수행. pipe(), fork(), dup2(), read(), snprintf() 등의 시스템 호출을 활용하여 입출력 리디렉션, 부모-자식 간 통신을 구현함.

먼저 pipe(fd) 호출해서 파이프를 생성함. fd[0]은 읽기, fd[1]은 쓰기용. fork()를 호출해서 부모-자식 프로세스를 분기함.

자식 프로세스인 경우 (pid == 0):

- fd[0] 닫음 (읽기 필요 없어서)
- dup2(fd[1], STDOUT_FILENO) 통해 표준 출력(stdout)을 파이프 쓰기 끝으로 리디렉션함
- fd[1] 썼으니까 닫기
- 전역 변수 isPipeMode를 1로 설정
- left 명령어 양끝 공백 제거하고 checkCommand()로 실행
- _exit(0)으로 자식 프로세스 종료

부모 프로세스인 경우:

- fd[1] 닫음 (쓰기 필요 없으니까)
- wait(NULL)로 자식 종료 대기
- 자식 프로세스가 출력한 결과를 read(fd[0], buf, sizeof(buf)-1)로 읽어옴
- 읽은 후 이제 필요 없으니까 fd[0] 닫음
- 읽은 데이터가 있다면 ₩n 제거
- buf 정리한 다음, right 명령어와 붙여서 최종 명령어 문자열 구성 (snprintf 이용)
- isPipeMode를 0으로 복원해서 파이프 모드 해제
- 최종 문자열 checkCommand()로 실행함

결과적으로 runPipe는 left | right 형태의 명령어를 처리함. 자식이 left 실행하고 부모가 그 결과 받아서 right 명령어에 넘기는 구조로 동작함.

*exit()보다 _exit()가 더 보안적으로 뛰어나다고 하길래 _exit를 사용했습니다.

* snprintf를 사용한 이유는 right 명령어와 left의 출력을 붙이기 위함입니다.

Step6. programstart

```
//------------ make child process using fork ----
int programstart(char* cmd) {
    trim(cmd);

    if (strncmp(cmd, "cd", 2) == 0 ||
        strncmp(cmd, "echo", 4) == 0 ||
        strncmp(cmd, "exit", 4) == 0) {
        checkCommand(cmd);
        return 1;
    }
    else {
        pid_t pid = fork();
        if (pid == 0) {
            checkCommand(cmd);
            _exit(0);
        }
        else {
            if (!isBackgroundMode) {
                int status;
                waitpid(pid, &status, 0);
                return WEXITSTATUS(status);
            }
            else {
                return 0;
            }
        }
    }
}
//-----------------------------------------------
```

programstart 함수는 쉘 명령어를 처리할 때 **필요에 따라 자식 프로세스를 생성하고**, 명령을 실행하는 역할을 함.

입력으로는 실행할 명령어 문자열 cmd를 받음.
(cmd = 가공된 userInput 문자열)

가장 먼저 trim(cmd) 함수를 호출해서 명령어 문자열 양끝의 공백을 제거함.
이후 cmd가 "cd", "echo", "exit"로 시작하는지 검사함.
이 명령어들은 쉘 자체에서 동작해야 하는 내장 명령어이므로, 별도로 fork 없이 checkCommand(cmd)로 바로 처리함.
그 후 함수는 return 1; 반환해서 종료시킴.

위 명령어들 같은 내장 명령어가 아니라면...
fork()를 호출해서 자식 프로세스를 생성함. 자식 프로세스에서는 checkCommand(cmd)를 호출하여 명령어를 실제로 실행하고 _exit(0)으로 종료함.

부모 프로세스에서는 두 가지 경우로 나뉨.

1. isBackgroundMode == 0

(즉 백그라운드 모드가 아니면), waitpid()로 자식 프로세스가 종료될 때까지 기다림. 이때 자식의 종료 코드를 WEXITSTATUS(status)로 가져와서 반환함.

2. isBackgroundMode == 1

즉 명령어가 백그라운드에서 실행되도록 요청되었으면, 부모는 자식 종료를 기다리지 않고 바로 return 0;으로 반환함.

---

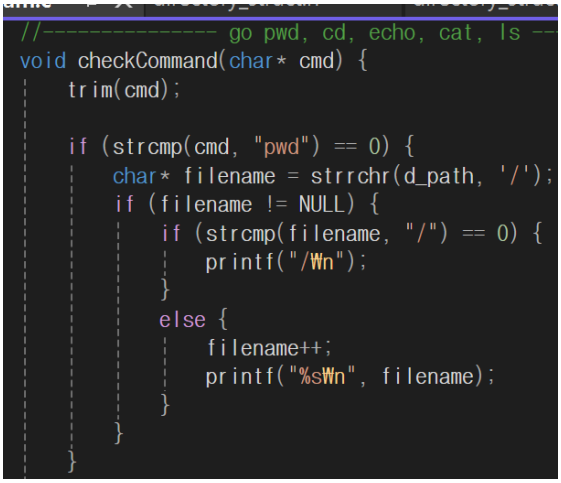
*background실행에 관하여
백그라운드 실행의 핵심: 자식 프로세스를 실행은 하지만, 그 종료를 기다리지 않고 부모는 바로 다음 일을 한다는 점
그래서 부모가 자식을 기다리지 않아야 하기에, background 실행을 알려주는 isBackgroundMode 플래그가 1일 시에는 waitpid를 안하고 넘어가게끔 설계

*wait가 아닌 waitpid를 쓴 이유
Wait는 아무 자식 프로세스나 종료되기를 기다리는 것이고 waitpid는 특정 자식 프로세스가 종료되는 걸 기다리는 것이기 때문. Waitpid가 더 정확할 듯 싶어 waitpid를 썼습니다.

Step7. runPipe

checkCommand 함수는 사용자가 입력한 명령어 문자열 cmd를 실제로 해석하고 실행하는 역할을 담당함.
pwd, cd, echo, cat, ls 같은 기본 쉘 명령어를 구현하였으며, 그 외 명령어에 대해서는 에러 메시지를 출력하게끔 함.

함수 시작 시 trim(cmd)을 호출하여 명령어 앞뒤 공백 제거하고 이후 명령어 내용에 따라 분기 처리함.



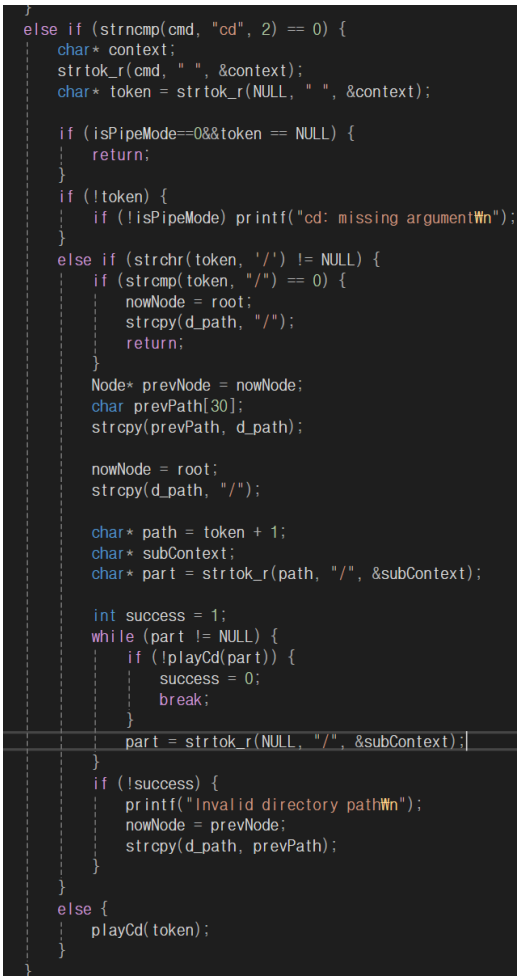
```
//---------------- go pwd, cd, echo, cat, ls --
void checkCommand(char* cmd) {
    trim(cmd);

    if (strcmp(cmd, "pwd") == 0) {
        char* filename = strrchr(d_path, '/');
        if (filename != NULL) {
            if (strcmp(filename, "/") == 0) {
                printf("/₩n");
            }
            else {
                filename++;
                printf("%s₩n", filename);
            }
        }
    }
```

pwd 명령 처리

strcmp(cmd, "pwd") == 0 조건 만족 시 현재 경로를 출력함.
D_path는 /home/oepickle 같이 경로의 전체 내용을 모두 담고 있는 문자열이기에, '/'를 기준으로 끊어줌
=> d_path에서 마지막 '/'를 기준으로 파일명을 파싱

- 루트 디렉토리 /일 경우 / 출력
- 그 외에는 마지막 디렉토리 이름만 출력

cd 명령 처리



```
else if (strncmp(cmd, "cd", 2) == 0) {
    char* context;
    strtok_r(cmd, " ", &context);
    char* token = strtok_r(NULL, " ", &context);

    if (isPipeMode==0&&token == NULL) {
        return;
    }
    if (!token) {
        if (!isPipeMode) printf("cd: missing argument₩n");
    }
    else if (strchr(token, '/') != NULL) {
        if (strcmp(token, "/") == 0) {
            nowNode = root;
            strcpy(d_path, "/");
            return;
        }
        Node* prevNode = nowNode;
        char prevPath[30];
        strcpy(prevPath, d_path);

        nowNode = root;
        strcpy(d_path, "/");

        char* path = token + 1;
        char* subContext;
        char* part = strtok_r(path, "/", &subContext);

        int success = 1;
        while (part != NULL) {
            if (!playCd(part)) {
                success = 0;
                break;
            }
            part = strtok_r(NULL, "/", &subContext);
        }
        if (!success) {
            printf("Invalid directory path₩n");
            nowNode = prevNode;
            strcpy(d_path, prevPath);
        }
    }
    else {
        playCd(token);
    }
}
```

strncmp(cmd, "cd", 2) == 0일 때 경로 변경 처리 수행함.
strtok_r() 함수로 cd 이후 경로 인자 추출함.

- 인자가 없으면 cd: missing argument 메시지 출력 (단, 파이프 모드 아닐 때만)
- 인자가 /로 시작하면 절대 경로 처리:
  현재 노드를 root로 리셋하고, 경로를 /부터 분할해서 하나씩 이동 시도함 (playCd 사용).
  이동 도중 실패하면 원래 위치로 복원하고 오류 메시지 출력함.
- 인자가 상대 경로이면 단순히 playCd(token) 호출

*strtok가 아닌strtok_r을 쓴 이유
strtok()은 문자열을 자를 때 자기가 어디까지 잘랐는지 내부에서 기억함. 그래서 두 개 이상의 문자열을 동시에 자르면 정보가 섞여서 제대로 못 자름.
반면 strtok_r()은 그 위치 정보를 내가 따로 저장해서 넘겨주기 때문에, 여러 문자열을 동시에 잘라도 잘 동작함.
그래서 strtok_r을 사용했다.

```
}
else if (strncmp(cmd, "echo", 4) == 0) {
    printf("%s₩n", cmd + 5);
}
else if (strncmp(cmd, "cat", 3) == 0) {
    char* context;
    strtok_r(cmd, " ", &context);
    char* token = strtok_r(NULL, " ", &context);

    if (token == NULL&&isPipeMode!=0) {
        char buf[200];
        ssize_t n;
        int totalRead=0;
        while ((n = read(STDIN_FILENO, buf, sizeof(buf))) > 0) {
            totalRead += n;
            if(totalRead>100) break;
            write(STDOUT_FILENO, buf, n);
        }
        return;
    }
    catOutput[0] = '₩0';

    while (token != NULL) {
        playCat(token, catOutput);
        token = strtok_r(NULL, " ", &context);
    }

    int len = strlen(catOutput);
    if (len > 0 && catOutput[len - 1] == '₩n') {
        catOutput[len - 1] = '₩0';
    }

    printf("%s₩n", catOutput);
    return;
}
```

echo 명령 처리

strncmp(cmd, "echo", 4) == 0일 때 실행됨.

echo는 어렵지 않게 구현함.

단순히 입력 받은 명령(cmd)에서 "echo " 이후 문자열을 잘라서 그대로 출력함.

즉 cmd + 5를 printf()로 출력

cat 명령 처리

strncmp(cmd, "cat", 3) == 0일 때 실행됨.

- Token(파일 인자)==NULL && isPipeMode==1

표준 입력(STDIN)을 받아서 최대 100바이트까지 읽고,

표준 출력(STDOUT)으로 그대로 출력함.

이는 파이프를 통해 cat이 사용될 때 처리되는 부분임.

- isPipeMode==0

catOutput 버퍼 초기화 후, 파일 인자를 하나씩 받아 playCat() 호출로 파일 내용 읽어옴

모든 파일 내용을 합쳐서 출력하고, 마지막 ₩n제거 후 출력

```
}
else if (strcmp(cmd, "ls") == 0) {
    playLs();
    return;
}
else {
    printf("Error: '%s' is Invalid command.₩n", cmd);
}
```

ls 명령 처리

strcmp(cmd, "ls") == 0일 때 실행됨.

디렉토리 내 항목을 출력하는 playLs() 호출

기타 명령어

위 조건들에 모두 해당하지 않는 경우 Error: 'cmd' is Invalid command. 메시지 출력함.

Step8. playCd

```
//-------------make cd ------------------------------
int playCd(char* token)
{
    trim(token);
    int returnVal = 0;
    if (strlen(token) == 0 || token[0] == ' ') {
        printf("Invalid Syntax₩n");
        return 0;
    }
    else if (strcmp(token, ".") == 0) {
        return 1;
    }
    else if (strcmp(token, "..") == 0) {
        char* lastSlash = strrchr(d_path, '/');
        if (lastSlash != NULL && lastSlash != d_path) {
            *lastSlash = '₩0';
        }
        else {
            strcpy(d_path, "/");
        }
        nowNode = nowNode->parent;
        return 1;
    }
```

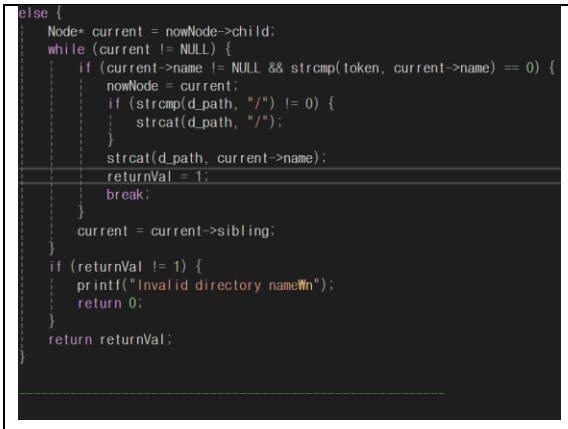
playCd 함수는 cd 명령어의 실제 경로 이동을 처리하는 함수

인자로 받은 token을 기준으로 현재 디렉토리(nowNode)를 이동시키고, 경로 문자열(d_path)도 갱신함.

먼저 trim(token)으로 공백 제거하고, 길이가 0이거나 공백만 있을 경우 "Invalid Syntax" 출력 후 실패 반환함.

- 입력(token)=="."이면

현재 디렉토리에 머무는 의미이므로 그대로 1 반환함.



```
else {
    Node* current = nowNode->child;
    while (current != NULL) {
        if (current->name != NULL && strcmp(token, current->name) == 0) {
            nowNode = current;
            if (strcmp(d_path, "/") != 0) {
                strcat(d_path, "/");
            }
            strcat(d_path, current->name);
            returnVal = 1;
            break;
        }
        current = current->sibling;
    }
    if (returnVal != 1) {
        printf("Invalid directory name₩n");
        return 0;
    }
    return returnVal;
}
```

- 입력(token) == ".."이면

상위 디렉토리로 이동하는 명령이므로:

- d_path에서 마지막 / 이후 경로를 잘라냄
- nowNode를 parent로 이동시킴
- 루트에 도달한 경우에는 d_path를 "/"로 초기화함

- 그 외의 경우

하위 디렉토리로 이동하는 경우임.

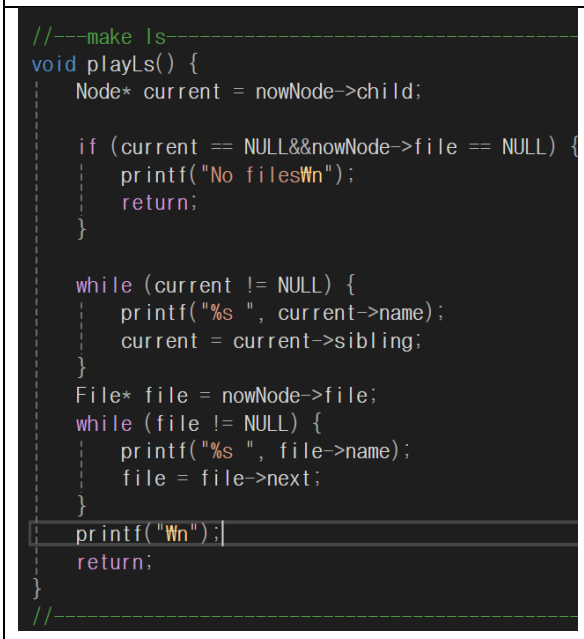
현재 노드의 자식 노드를 순회하면서, token과 같은 이름의 디렉토리를 찾아야 한다. 그 후

- nowNode를 해당 디렉토리로 바꿈
- d_path에 디렉토리 이름을 붙여 경로 갱신함
- returnVal = 1로 설정하고 반복문 종료

만약 이때 디렉토리를 못 찾으면

"Invalid directory name" 출력하고 실패 반환함.

Step9. playLs



```
//---make ls-----------------------------------
void playLs() {
    Node* current = nowNode->child;

    if (current == NULL&&nowNode->file == NULL) {
        printf("No files₩n");
        return;
    }

    while (current != NULL) {
        printf("%s ", current->name);
        current = current->sibling;
    }
    File* file = nowNode->file;
    while (file != NULL) {
        printf("%s ", file->name);
        file = file->next;
    }
    printf("₩n");
    return;
}
//---------------------------------------------
```

이 함수는 현재 디렉토리(nowNode)의 하위 디렉토리(child)와 파일 목록(file)을 출력하는 역할을 함.

먼저 nowNode->child를 current에 저장해서

현재 위치한 디렉토리의 하위 디렉토리 순회 시작점을 저장

- 만약 nowNode에 연결된 디렉토리도 없고(nowNode->child) 파일(nowNode->file)도 없으면 "No files" 출력하고 종료함.

[directory출력]

그렇지 안다면, current가 가리키는 노드부터 while 루프 돌면서 각 디렉토리 이름을 출력함.

→ current = current->sibling으로 순회하는 구조

[file 리스트 출력]

nowNode->file을 따라가면서 각 파일 이름을 출력함.

파일은 file->next로 연결되어 있음.

후 return을 하며 프로그램 종료.

void형이기에 playLs는 아무것도 반환하지 않음

실행과 동시에 화면에 출력하는 방식

Step10. playCat

```
//---------make cat-----------------------------
void playCat(char* str, char arr[VALUE_SIZE]) {
    trim(str);
    nowFile = nowNode->file;
    int found = 0;

    while (nowFile != NULL) {
        if (strcmp(nowFile->name, str) == 0) {
            found = 1;
            strcat(arr, nowFile->text);
            strcat(arr, "₩n");
            break;
        }
        nowFile = nowFile->next;
    }

    if (!found) {
        strcat(arr, str);
        strcat(arr, ": No such file ");
    }
}
//-----------------------------------------------
```

playCat 함수는 cat 명령어를 구현한 function
현재 디렉토리 노드 속 파일 내용을 읽어서 출력 문자열에 붙이는 역할을 함 (이때 출력 문자열: arr)

두 개의 인자를 받음:

- str: 파일 이름
- arr: 출력할 문자열들을 붙일 array

먼저 파일 이름에서 공백 제거하고,
현재 디렉토리의 파일(nowNode->file)을 순회함.

strcmp()로 str과 파일 이름이 일치하는지 확인하고,
파일을 찾으면:

- found = 1로 표시
- 파일 내용(nowFile->text)을 arr에 붙임
- 가독성을 위해 ₩n추가함

만약 해당 파일이 없다면

- "No such file" 메시지를 arr에 붙이고 이 arr를 내보냄으로써 파일이 없다는 메시지를 커널에 전달해줌

Step11. playCat

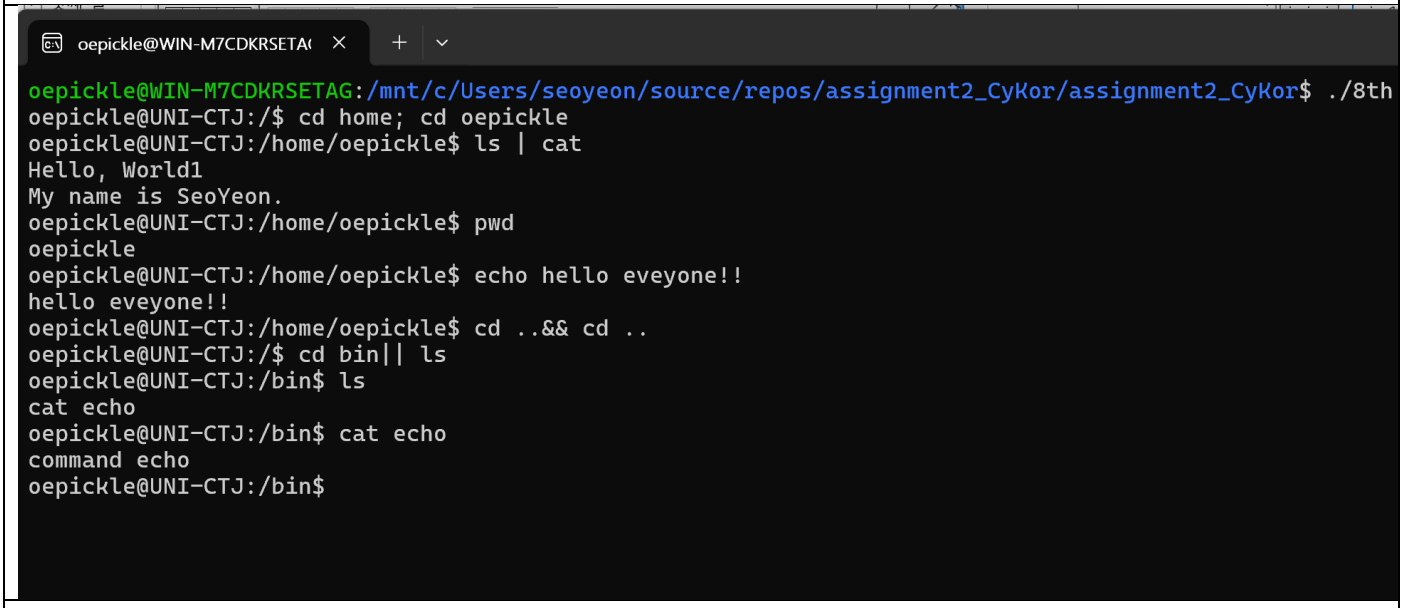
```
//-----------remove space-----------------------
void trim(char* str) {
    int len = strlen(str);
    while (len > 0 && (str[len - 1] == ' '
        || str[len - 1] == '₩n')) {
        str[len - 1] = '₩0';
        len--;
    }
    int index = 0;
    while (str[index] == ' ') {
        index++;
    }
    if (index > 0) {
        memmove(str, str + index, strlen(str + index) + 1);
    }
}
//-----------------------------------------------
```

trim 함수는 문자열 앞뒤 공백(space, ₩n) 제거하게끔 설계함

뒤쪽 공백은 '₩0'로 잘라내고, 앞쪽 공백은 memmove()라는 내장함수를 활용해서 문자열을 앞으로 당겨서 삭제함.

명령어 입력 처리할 때 정확한 비교 위해 만들어주었음
ex: " cd .. "처럼 입력돼도 "cd .."로 처리 가능하게끔

번외: 실행 화면



번외: makefile