File Systems

Operating Systems

Project 3

32181928 박찬호, 32183641 이창윤

Dankook University

Mobile Systems Engineering

2022 Fall / Operating Systems

Contents 목차

서론

Persistent Storage

Random Access Memory (RAM)으로 대표되는 memory 는 프로그램의 실행과 함께 프로그램의 data 와 instruction 이 load 되는 공간이며 이는 processor 가 instruction 을 수행할 수 있게 한다. 그러나 memory 는 전원 공급이 차단되면 load 된 내용이 사라지고, 이러한 특성 때문에 일반적인 computer 는 persistent storage 를 사용하여 전원 차단 후에도 내용을 저장한다. 현대의 persistent storage 는 Hard Disk Drive(HDD)와 Solid State Drive(SSD)가 주로 사용된다.

File & Directory

Persistent Storage 에 persistent data 를 관리하기 위해서는 가상화와 관련된 두 가지 개념, File 과 Directory 이 필요하다.

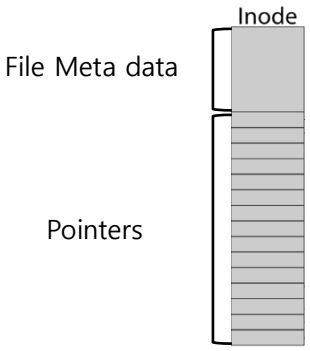
첫 번째로 File 은 바이트의 배열로 표현되는 데이터인데, 이를 관리하기 위한 meta-data 를 포함한다. Meta-data 는 소유자, 접근 권한, 크기 등의 속성을 포함한다. 파일은 해당 파일을 지칭하는 low-level name 을 가지고 있고, 이 low-level name 은 후술할 inode number 이다.

두 번째로 Directory 는 여러 file 과 directory 를 목록으로 가지고, 이를 계층으로 관리할 수 있게 하는 구조이다. Directory 도 마찬가지로 meta-data 를 가지고 있으며, directory 의 data 는 <사용자가 읽을 수 있는 이름, low-level name>쌍의 목록으로 단순화하여 생각할 수 있다. 실제의 directory 는 Directory Entry 와 같은 directory structure 를 사용한다. Directory 는 Directory Tree 형태로 계층을 구성하며, '/'로 표현되는 root directory 에서 시작하여 하위 directory 를 표현한다.

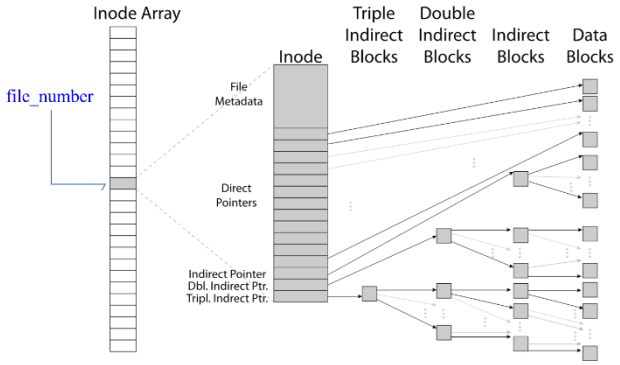
File 과 Directory 라는 두 개념을 사용하면 File System 을 구성하고, persistent data 를 관리할 수 있다. File System 은 File 에 접근할 때, 사용자가 읽을 수 있는 path 를 directory structure 를 통해 inode number 로 변환하는 것을 반복하여 file 의 directory 로 찾아가고, 해당 file 의 inode 에 도달하면 file 의 data block 에 접근한다.

Inode & File System

Inode 는 File Meta data 와 Data Block 혹은 다른 inode 를 가리키는 pointer 들로 구성되어 있다. 때 File 의 data block 을 가리키는 pointer 는 Direct pointer, 다른 inode 를 가리키는 pointer 는 Indirect pointer 라고 한다. File system 은 이러한 indirect pointer 와 direct pointer 를 사용하여 큰 파일과 작은 파일을 여러 단계의 redirection 을 통해 효과적으로 관리할 수 있다.



실제의 File system 은 Inode 를 배열로 구성하여 전체 data block 을 크고 작은 file 로 mapping 한다. 이러한 구조에서 inode number 로 file 을 읽는 과정은 아래의 그림처럼 표현된다.



여기에 전체 File system 에 대한 정보를 가진 Super block 이 추가하면 아래와 같이 단순화된 형태의 file system 구조가 만들어진다.

Super Block	Inode Array	Data Block

구현 목표

프로젝트의 요구사항을 바탕으로, 구현 목표를 정해보자. 먼저 본 프로젝트에서는 hw1 의 shell 과 결합하여 파일을 다루도록 할 것이다. 따라서 구현 목표는 아래와 같다

1. Shell 에서 disk image 를 mount, unmount 할 수 있도록 할 것.
2. Mount 시, root file system 을 읽어들이도록 할 것
3. Mount 이후, 아래의 연산을 지원할 것
    A. ls – Directory 의 file 목록을 출력
    B. cat – file 의 내용을 출력
    C. pwd – 현재 directory 위치를 absolute directory 로 출력
4. 위의 구현을 file system 의 동작 방식에 맞게 구현할 것
5. File system 의 mount 와 unmount, 명령 실행 등에 있어 failure 가 없도록 구현할 것

실행 방법

1. https://github.com/mobile-os-dku-cis-mse/2022-os-proj3 을 clone 받는다.
2. 32181928 branch 로 checkout 한다.

3-1. make all 을 입력하여 build 한 후 ./main 을 실행하여 shell 을 실행한다.

3-2. Disk Image 가 있는 directory 로 이동 후, mount {disk image name} 명령을 입력하여 mount 한다

3-3. ls, cat {file name}, pwd, unmount 등을 사용하여 테스트한다.

본론

fs.h & disk.img

본 프로젝트에서 문제 해결을 위해 주어진 파일은 fs.h 와 disk.img, disk.img.hex 이다. disk.img 는 현재 프로젝트에서 사용할 수 있는 파일 시스템 이미지 binary 이고, disk.img.hex 는 binary 를 hex 로 읽어낸 파일이다. fs.h 는 이 프로젝트 구현에 기본 조건이 되는 header 이다.

fs.h 는 define values, super block structure, inode structure, blocks structure, partition structure, dentry structure 로 이루어져 있다. 구현의 시작단계에서 전체 disk.img 를 읽어 fs.h 의 구조에 맞게 출력하고, 이 형태가 맞는지 disk.img.hex 를 이용하여 검증하면 빠르게 구현에 접근할 수 있는데, 이 때 필요한 내용은 partition 구조체이다. Partition 구조체는 아래와 같다.

```
/* physical partition structure */
struct partition {
    struct super_block s;
    struct inode inode_table[224];
    struct blocks data_blocks[4088]; //4096-8
};
```

Partition 구조체를 살펴보면, super block 구조체, inode 구조체 배열, blocks 구조체 배열로 이루어져 있고, 이는 위에서 전체 구조를 표현하는데 사용한 아래 그림과 동일한 구조임을 알 수 있다.

따라서, 이 형태에 맞게 전체 disk.img 를 읽어서 출력하면 아래와 같은 결과를 얻을 수 있다. 아래의 결과는 disk.img.hex 를 통해 검증을 마친 결과이다. fs.h 와 비교하여 보이면 아래와 같다.

```
struct super_block {
    unsigned int partition_type;
    unsigned int inode_size;
    unsigned int block_size;
    unsigned int first_inode;
    unsigned int num_inodes;
    unsigned int num_inode_blocks;
    unsigned int num_free_inodes;
    unsigned int num_blocks;
    unsigned int num_free_blocks;
    unsigned int first_data_block;
    char volume_name[24];
    unsigned char padding[960];
};

struct inode {
    unsigned int mode;
    unsigned int locked
    unsigned int date;
    unsigned int size;
    int indirect_block;
    unsigned short blocks[0x6];
};
```

[Super Block]
partition_type : 0x1111 (4369)
inode_size : 0x20 (32)
block_size : 0x400 (1024)
first_inode : 0x2 (2)
num_inodes : 0xe0 (224)
num_inode_blocks : 0x7 (7)
num_free_inodes : 0x79 (121)
num_blocks : 0xff8 (4088)
num_free_blocks : 0xff8 (4088)
first_data_block : 0x8 (8)
volume_name :Simple_partition_volume
[Inode 0]
...(중략)...
[Inode 2]
mode : 0x20777 (132983)
locked : 0 (0)
date : 0 (0)
size : 0xcc0 (3264)
indirect_block : 0 (0)
blocks[0] : 0 (0)
blocks[1] : 0x1 (1)
blocks[2] : 0x2 (2)
blocks[3] : 0x3 (3)
...(후략)...

이 결과는 result.txt 로 저장하여 레포지토리에 추가하였다.

구현 결과

먼저 전체 파일과 그 설명은 아래와 같다

2022-os-proj3

- ㄴ disk.img : 주어진 disk image file
- ㄴ disk.img.hex : 주어진 disk image file 의 hex dump
- ㄴ file_system.c : file_system 을 mount 하고 읽는 함수의 구현
- ㄴ file_system.h : mount 후 사용할 mount env 구조체, file_system.c 의 함수들
- ㄴ fs.h : 주어진 file system 구조 헤더
- ㄴ hash_set.c : 빠른 inode 탐색을 위한 hash set 구현
- ㄴ hash_set.h : pair, node, set 구조체
- ㄴ LICENSE
- ㄴ main.c : env 를 세팅하고 shell 을 실행하는 main 함수
- ㄴ Makefile
- ㄴ project3.pdf
- ㄴ README.md
- ㄴ result.txt : disk.img 를 fs.h 의 형태에 맞게 읽은 결과
- ㄴ SiSH
  - ㄴ (Shell 관련 file 들)

아래에서는 이 전체 파일들 중 중요한 코드 위주로 설명을 진행하는데, 그 순서는 다음과 같다.

shell 에서 mount 와 연산을 담당하는 코드 → mount ls, cat, pwd 함수 코드 → hash set 코드

Mount 관련 코드

먼저 mount 후 사용할 수 있는 mount env 구조체를 확인하겠다.

<file_system.h>

```
typedef struct _mount_env {
    SuperBlock* super_block; // file system 이미지로부터 읽은 super block
    Inode* inode_table; // file system 이미지로부터 읽은 inode table (배열)
    Block* block; // file system 이미지로부터 읽은 data block
    int block_starting_point; // 편의를 위해 block 의 시작점 offset 을 저장
    int block_finishing_point; // 편의를 위해 block 의 종료점 offset 을 저장
    FILE* disk_img; // disk.img 의 파일 포인터
    Directory* current_directory; // 현재 directory 를 표현하는 문자열 배열
    BST_set* set; // inode 탐색을 위한 binary search set (with hash)
    DIR_FILES* dir_files; // 편의를 위해 현재 directory 의 파일명을 caching
    char* disk_img_name; // disk.img 파일 이름
} MountEnv;


typedef struct _directory {
    char** directory;
    int dir_length;
} Directory;
```

이를 사용하여 실제 mount 시 동작하는 코드는 아래와 같다.

<SiSH.c>

```
int sish_loop(Env* env) {
(중략)

    MountEnv* mount_env = malloc(sizeof(*mount_env));
```

mount_env 를 미리 할당하여, mount 후 사용할 수 있도록 한다.

```
(중략)
```

Mount 시 host name, current path 등을 다른 방식으로 update 해야하므로 unmount 시에만 shell 의 함수를 사용하여 update 한다.

```
        if (!env->mount) {
            env->host_name = get_host_name(env->host_name_max);
            env->current_path_abs = get_current_path_abs(env->path_max);
            env->current_path = get_current_path(env->current_path_abs, env-
>home_dir);
            env->login_name = get_login_name(env->login_name_max);
            env->home_dir = get_home_dir(env->login_name);
        }
```

Mount 시 사용할 수 있는 명령이 제한되고, 별도의 구현이 필요하므로 mount 여부를 env 에서 관리하고, 그에 따라 다른 동작을 하도록 구현한다.

```
        if (!strcmp(command, "quit") || !strcmp(command, "exit")) {
            //fprintf(stdout, "\n");
            exit(EXIT_SUCCESS);
        } // quit 과 exit 는 mount 여부에 상관없이, shell 을 종료하도록 if 밖에 배치

        if (!(env->mount)) { // unmoun 의 경우
            if (!strcmp(command, "mount")) { // 원래의 코드에서 mount 명령만 추가
                (에러 처리 생략)

                env->mount = 1; // mount 되었음을 체크

(mount env 초기화 일부 생략)

                mount_env->disk_img = mount(mount_env->disk_img_name,
&(mount_env->super_block), &(mount_env->inode_table));
// mount 함수로 전체 disk.img 의 super block, inode table 등을 읽어서 저장

(mount env 초기화 일부 생략)

                mount_env->set = malloc_s(sizeof(*(mount_env->set)),
__func__);
                mount_env->set->size = 0;
                mount_env->set->root = NULL;

                mount_env->set->root_inode = mount_env->super_block-
>first_inode;

// hash set 의 root, size 등을 초기화

                read_block(first_inode, mount_env); // root directory 에
해당하는 inode 를 통해 해당하는 data block 을 읽음

                (출력 생략)
```

```
            (host, path 등 업데이트 생략)

                continue;
            }

(중략)

        } else { // mount 되었을 때
            if (!strcmp(command, "ls")) {

(중략)


                ls(mount_env);

                continue;
            }

            if (!strcmp(command, "cat")) {

(중략)
                cat(file, mount_env);

                continue;
            }

            if (!strcmp(command, "pwd")) {

(중략)

                printf("%s\n", current_dir_to_string(mount_env-
>current_directory, env->path_max));

                continue;
            }

            if (!strcmp(command, "unmount")) {
                env->mount = 0;

                fclose(mount_env->disk_img);

                continue;
            }
        }

        clear_buffer(buffer, sizeof(buffer));
    }
}
```

위에서 확인할 수 있듯, mount 시 mount 함수와 read_block 함수를, ls 시 ls 함수를, cat 시 cat 함수를, pwd 시 current_dir_to_string 함수를 사용한다. 각각의 함수는 아래와 같다. (current_dir_to_string 함수는 생략한다)

<file_system.c>

```
FILE* mount(char* disk_img_name, SuperBlock** super_block, Inode**
inode_table) {

    FILE* fp;

    if ((fp = fopen(disk_img_name , "rb+")) == NULL) {
// 추후 write 구현을 위해 rb+로 open

        fprintf(ERR_STREAM, "Failed to open file\n");
    }

    (super block 할당 생략)

    fread(*super_block, sizeof(**super_block), 1, fp);
// SuperBlock 크기만큼 읽어들임

    (super block 출력 생략)

    (inode table 할당 생략)

    for (int i = 0; i < (*super_block)->num_inodes; i++) {
        fread(&((*inode_table)[i]), sizeof(**inode_table), 1, fp);
    }
// inode 크기를 한 단위로 하나씩 읽어서 저장

    return fp;
}
```

이를 통해 super block 과 inode table 에 disk.img 의 값을 읽어서 저장하는데, 이 함수를 호출 시 mount env 에 속한 super block 과 inode table 을 사용하여 전체 실행에서 mount env 를 통해 이 값들을 사용할 수 있도록 한다.

이후 첫 번째 inode 를 사용하여 read_block 을 통해 root 의 data block 을 읽는다. 다른 inode 를 읽을 때도 마찬가지로 read_block 이 사용되는데 그 코드는 아래와 같다.

<file_system.c>

```
void* read_block(Inode inode, MountEnv* mount_env) {
// directory 인 경우에는 mount env 를, file 인 경우에는 block 배열을 반환하므로
void 포인터를 반환형으로 설정

    int block_size = mount_env->super_block->block_size;

    int block_index = inode.size / block_size;

    int is_dir = ((inode.mode >> 12) << 12) == INODE_MODE_DIR_FILE;


    if (is_dir) { // directory 라면 dentry 를 읽는다.

        (mount env update 생략)

        for (int i = 0; i <= block_index; i++) { // 각 block 에 따라

(읽기 시작하는 지점 read_point 세팅 생략)

            fseek(mount_env->disk_img, read_point, SEEK_SET);

            while (ftell(mount_env->disk_img) < read_point + mount_env-
>super_block->block_size) {

(Dentry 에 속하는 변수들에 맞게 읽는 부분 생략)

                if (inode_num * dir_length * name_len * file_type == 0) {
                    continue; // invalid 한 경우 뛰어넘기
                }

               (valid 하다면 directory 를 읽는 부분 생략)

                int hash = hash_func(directory_name, name_len);

// directory_name 을 hash_func 에 전달하여 hash 값 생성


                Pair* pair = (Pair*)malloc_s(sizeof(*pair), __func__);

                pair->inode = inode_num;

                pair->hash = hash;

                strcpy(pair->name, directory_name);
// hash - inode num 을 쌍으로 하는 pair 에 값을 저장하고, hash collision 확인에
사용하기 위해 directory name 도 저장
```

```
(mount env update 생략)

                insert_node(mount_env->set, pair);
// hash를 사용한 binary search tree set에 pair 추가
            }
        }

        return mount_env; // mount env 반환

    } else { // directory가 아니라 file인 경우

(읽기 시작하는 지점 read_point 세팅 생략)

        fseek(mount_env->disk_img, read_point, SEEK_SET);

        Block* blocks = malloc(sizeof(*blocks) * (block_index + 1));

        for (int i = 0; i <= block_index; i++) {
            fread(blocks + i, sizeof(*blocks), 1, mount_env->disk_img);
        }

        return blocks;

// block을 읽어서 반환
    }

    return NULL; // 에러가 발생한 경우 NULL 반환
}
```

그리고 ls, cat 함수는 아래와 같다.

<SiSH.c>

```
void ls(MountEnv* mount_env) {
    int size = mount_env->dir_files->size;
    char** file_list = mount_env->dir_files->file_list; // 현재 디렉토리의
파일들

    for (int i = 0; i < size; i++) {
        printf("%s\t\t", file_list[i]);
        if ((i + 1) % 4 == 0) {
            printf("\n");
        }
    }
    printf("\n");
}
```

```
void cat(char* file, MountEnv* mount_env) {
    int inode_num = find_inode(mount_env->set, file); // file 명을 hash 로 바꿔
set(binary search tree)에서 binary search

    if (inode_num == -1) {
        printf("cat : %s: Is not a valid file\n", file);
        return;
    } // 찾지 못한 경우

    Inode inode = mount_env->inode_table[inode_num];

    int is_dir = ((inode.mode >> 12) << 12) == INODE_MODE_DIR_FILE;

    if (is_dir) {
        printf("cat : %s: Is a directory\n", file);
        return;
    } // directory 인 경우 cat 사용 불가

    int block_size = mount_env->super_block->block_size;
    int block_index = inode.size / block_size;

    Block* blocks = read_block(inode, mount_env);

    for (int i = 0; i <= block_index; i++) {
        printf("%s", blocks[i].d);
    }
    printf("\n");
}
```

Hash set 의 구현은 아래와 같다

<hash_set.c>

```
int hash_func (char* str, int len) {
    unsigned long hash = 0;
    int c;
    for (int i = len - 1; i >= 0; i--)
    {
        c = str[i] - 'a' + 1;
        hash = (((hash << 5) - hash) + c) % TABLE_SIZE;
        continue;
    }
    return hash % TABLE_SIZE;
}
```

먼저 hash 함수는 아래의 수식을 사용하였다.

$$H = \sum_{i=0}^{l-1} a_i r^i \ mod\ M$$

l : $length\ of\ string$
H : $hash\ value\ of\ string$
M : $large\ number\ to\ limit\ hash$
a_i : $ASCII\ num\ of\ ith\ character\ of\ string$
r^i : $ith\ power\ of\ multiplier\ r$

보통 해시함수는 r 과 M 이 서로소로 정하기 때문에, 여기서 r 과 M 은 모두 소수를 사용하였다. r 은 31 로, M 은 2,147,483,647 로 정하였는데 이는 각각 $2^5 - 1$, $2^{32} - 1$ 이며 특히 M 은 일반적인 C 언어에서 signed integer 의 limit 이다.

따라서 최종 수식은

$$H = \sum_{i=0}^{l-1} a_i 31^i \ mod\ 2{,}147{,}483{,}647$$

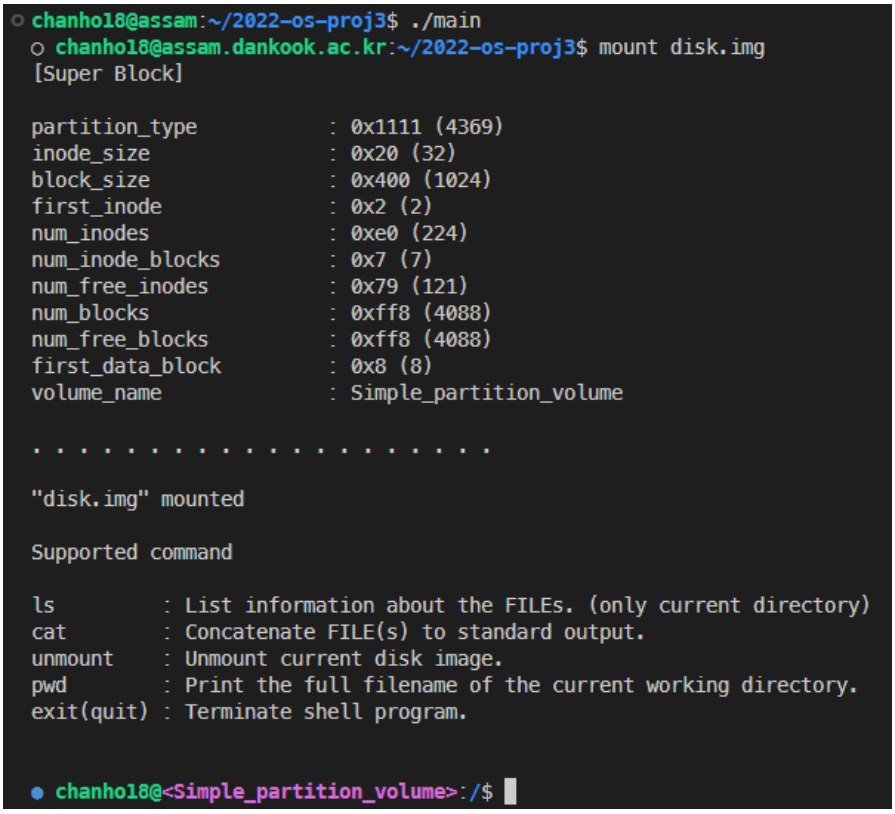
와 같고 연산 상의 최적화를 위해, 매 합에서 31 을 곱하는 것을 left bit shift 5 를 통해 32 를 곱하고 hash 를 빼는 것으로, 그리고 전체 합의 mod 를 매번 합할 때 마다 mod 를 수행한 값(나머지)을 합하는 것으로 대체한다.

Insert node 와 find node 는 일반적인 binary search tree 와 동일하므로 생략한다.

dentry 를 통해 얻어낸 file 의 이름을 ls 로 저장한 후, 그 이름을 통해 file 의 inode 를 찾고자 할 때 dentry 를 그대로 사용하게 되면 전체 dentry 배열을 탐색하며 현재 file 이름과 dentry 의 이름을 계속 비교하게 된다. 이 경우 전체 탐색은 O(n)의 시간 복잡도를 요구하며, 또한 매 반복마다 string 비교를 수행해야 하므로 탐색의 cost 가 크다. 그러나 위의 방식을 사용할 경우, binary search tree 를 사용하므로 탐색 시의 시간 복잡도는 O(log n)이며, 탐색 시에 값의 비교 역시 hash 값이라는 numeric 값을 비교하므로 string 의 길이에 무관하게 한 번의 비교만 수행하면 된다. 따라서 cost 에 있어 많은 개선이 이뤄진다.

실행 결과

Shell 실행과 mount. Mount 시의 host 는 현재 volume 의 이름으로 하고, 다른 색깔로 표현한다.



cat, ls, pwd 실행

```
● chanho18@<Simple_partition_volume>:/$ ls
.               ..              file_1          file_2
file_3          file_4          file_5          file_6
file_7          file_8          file_9          file_10
file_11         file_12         file_13         file_14
file_15         file_16         file_17         file_18
file_19         file_20         file_21         file_22
file_23         file_24         file_25         file_26
file_27         file_28         file_29         file_30
file_31         file_32         file_33         file_34
file_35         file_36         file_37         file_38
file_39         file_40         file_41         file_42
file_43         file_44         file_45         file_46
file_47         file_48         file_49         file_50
file_51         file_52         file_53         file_54
file_55         file_56         file_57         file_58
file_59         file_60         file_61         file_62
file_63         file_64         file_65         file_66
file_67         file_68         file_69         file_70
file_71         file_72         file_73         file_74
file_75         file_76         file_77         file_78
file_79         file_80         file_81         file_82
file_83         file_84         file_85         file_86
file_87         file_88         file_89         file_90
file_91         file_92         file_93         file_94
file_95         file_96         file_97         file_98
file_99         file_100
● chanho18@<Simple_partition_volume>:/$ cat file_95
file 95 852262977 6164355 1796894943

● chanho18@<Simple_partition_volume>:/$ cat ..
cat : ..: Is not a valid file
● chanho18@<Simple_partition_volume>:/$ pwd
/
● chanho18@<Simple_partition_volume>:/$
```

unmount 실행 및 unmount 시의 정상 shell 동작

```
● chanho18@<Simple_partition_volume>:/$ unmount
● chanho18@assam.dankook.ac.kr:~/2022-os-proj3$ ls
disk.img      file_system.c  file_system.o  hash_set.c  hash_set.o  main    main.o    project3.pdf  result.txt
disk.img.hex  file_system.h  fs.h           hash_set.h  LICENSE     main.c  Makefile  README.md     SiSH
● chanho18@assam.dankook.ac.kr:~/2022-os-proj3$ pwd
/home/chanho18/2022-os-proj3
● chanho18@assam.dankook.ac.kr:~/2022-os-proj3$ whoami
chanho18
```

결론

회고

본 프로젝트를 수행하며 간단한 형태의 file system 을 읽고 shell 과 연결하며 file system 의 구조를 이해할 수 있었다. 전체 구조에 대한 이해를 바탕으로 최초에는 write 구현에 시도하였으나, 현재의 disk image 를 수정하여 테스트하는 부분에서 free block bitmap 을 구성하여 disk image 를 수정하는 것보다 전체 file system image 를 새로 생성하는 것을 시도하는 것이 좋겠다는 생각이 들었다. 이에 전체 file system 을 구성하여 free block bitmap 을 구성하는 것까지는 성공했으나, 이를 image file 로 저장하고 다시 읽어 들이는 것까지는 시간의 부족하여 구현을 마무리하지 못했다. 이 부분에서 많은 아쉬움을 느낀다.

또한 한 가지 아쉬운 점은, 큰 크기의 파일이나 여러 계층의 directory 를 다룰 수 있도록 코드를 구현하였으나, 실제 이를 테스트할 수 있는 케이스가 없어 코드가 제대로 동작하는지 확인하지 못한 것이다. 이러한 부분에서 여러 종류의 disk image 가 있고, 이를 통해 다양한 경우를 테스트하면 좋을 것 같다는 생각이 든다.

아쉬운 부분이 있지만 학기를 마무리하며 file system 프로젝트를 성공적으로 마무리하게 되어 기쁘고, 한 학기동안 OS 수업을 통해 다양한 것을 시도하고 배우며 OS 전반에 대한 깊은 이해를 얻어갈 수 있어 감사함을 느낀다.

참고자료

Operating Systems : Three easy pieces – Remzi H. Arpaci-Dusseau

강의 자료

감사합니다