텍스트이(가) 표시된 사진

자동 생성된 설명

File Systems

Operating Systems

Project 3

32181928 박찬호, 32183641 이창윤

Dankook University

Mobile Systems Engineering

2022 Fall / Operating Systems

**Contents 목차**

**Ⅰ 서론 01**

**Ⅱ 본론 04**

**Ⅲ 결론 27**

Persistent Storage 01

File & Directory 01

Inode & File System 02

구현 목표 03

실행 방법 03

fs.h & disk.img 04

구현 결과 05

실행 결과 17

회고 11

참고자료 19

**서론**

**Persistent Storage**

Random Access Memory (RAM)으로 대표되는 memory는 프로그램의 실행과 함께 프로그램의 data와 instruction이 load되는 공간이며 이는 processor가 instruction을 수행할 수 있게 한다. 그러나 memory는 전원 공급이 차단되면 load된 내용이 사라지고, 이러한 특성 때문에 일반적인 computer는 persistent storage를 사용하여 전원 차단 후에도 내용을 저장한다. 현대의 persistent storage는 Hard Disk Drive(HDD)와 Solid State Drive(SSD)가 주로 사용된다.

**File & Directory**

Persistent Storage에 persistent data를 관리하기 위해서는 가상화와 관련된 두 가지 개념, File 과 Directory이 필요하다.

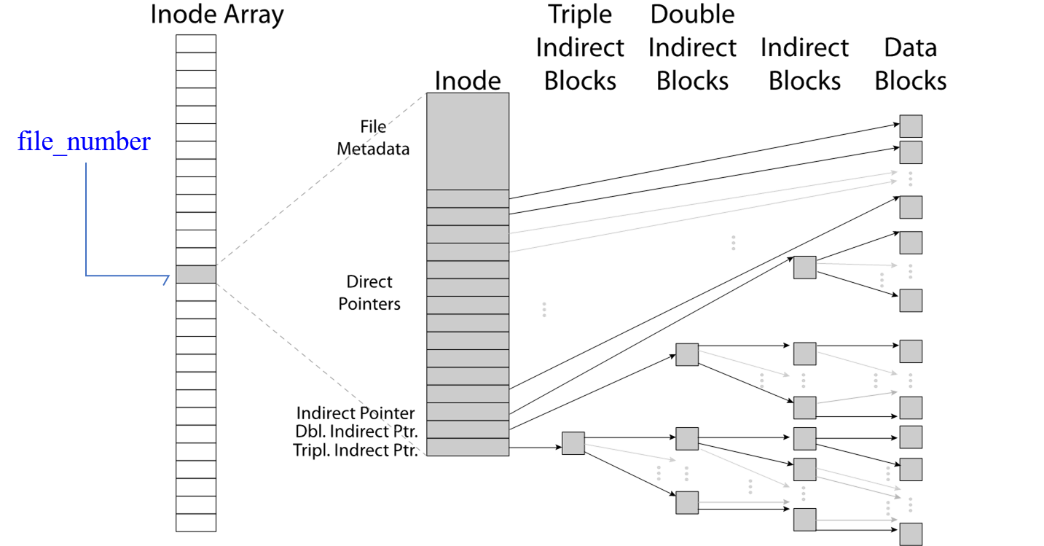
첫 번째로 File은 바이트의 배열로 표현되는 데이터인데, 이를 관리하기 위한 meta-data를 포함한다. Meta-data는 소유자, 접근 권한, 크기 등의 속성을 포함한다. 파일은 해당 파일을 지칭하는 low-level name을 가지고 있고, 이 low-level name은 후술할 inode number이다.

두 번째로 Directory는 여러 file과 directory를 목록으로 가지고, 이를 계층으로 관리할 수 있게 하는 구조이다. Directory도 마찬가지로 meta-data를 가지고 있으며, directory의 data는 <사용자가 읽을 수 있는 이름, low-level name>쌍의 목록으로 단순화하여 생각할 수 있다. 실제의 directory는 Directory Entry와 같은 directory structure를 사용한다. Directory는 Directory Tree 형태로 계층을 구성하며, ‘/’로 표현되는 root directory에서 시작하여 하위 directory를 표현한다.

File과 Directory라는 두 개념을 사용하면 File System을 구성하고, persistent data를 관리할 수 있다. File System은 File에 접근할 때, 사용자가 읽을 수 있는 path를 directory structure를 통해 inode number로 변환하는 것을 반복하여 file의 directory로 찾아가고, 해당 file의 inode에 도달하면 file의 data block에 접근한다.

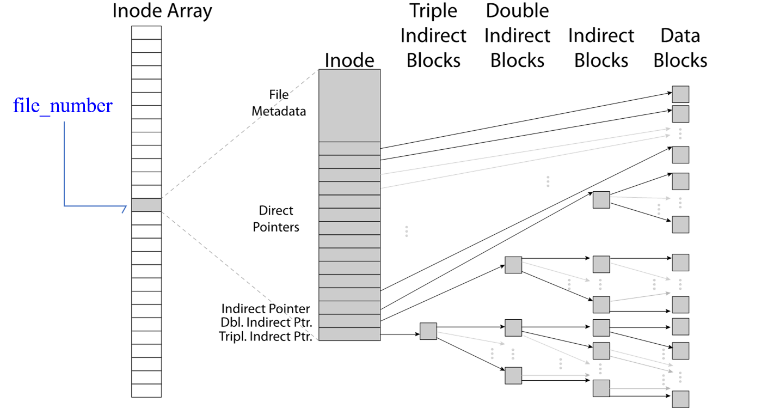
**Inode & File System**

Inode는 File Meta data와 Data Block 혹은 다른 inode를 가리키는 pointer들로 구성되어 있다. 때 File의 data block을 가리키는 pointer는 Direct pointer, 다른 inode를 가리키는 pointer는 Indirect pointer라고 한다. File system은 이러한 indirect pointer와 direct pointer를 사용하여 큰 파일과 작은 파일을 여러 단계의 redirection을 통해 효과적으로 관리할 수 있다.



File Meta data

Pointers

실제의 File system은 Inode를 배열로 구성하여 전체 data block을 크고 작은 file로 mapping한다. 이러한 구조에서 inode number로 file을 읽는 과정은 아래의 그림처럼 표현된다.

여기에 전체 File system에 대한 정보를 가진 Super block이 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 hw1의 shell과 결합하여 파일을 다루도록 할 것이다. 따라서 구현 목표는 아래와 같다

1. Shell에서 disk image를 mount, unmount 할 수 있도록 할 것.
2. Mount 시, root file system을 읽어들이도록 할 것
3. Mount 이후, 아래의 연산을 지원할 것
   1. ls – Directory의 file 목록을 출력
   2. cat – file 의 내용을 출력
   3. pwd – 현재 directory 위치를 absolute directory로 출력
4. 위의 구현을 file system의 동작 방식에 맞게 구현할 것
5. File system의 mount와 unmount, 명령 실행 등에 있어 failure가 없도록 구현할 것

**실행 방법**

1. <https://github.com/mobile-os-dku-cis-mse/2022-os-proj3> 을 clone 받는다.
2. 32181928 branch로 checkout 한다.
   1. make all 을 입력하여 build한 후 ./main을 실행하여 shell을 실행한다.
   2. Disk Image가 있는 directory로 이동 후, mount {disk image name} 명령을 입력하여 mount 한다
   3. ls, cat {file name}, pwd, unmount 등을 사용하여 테스트한다.

**본론**

**fs.h & disk.img**

본 프로젝트에서 문제 해결을 위해 주어진 파일은 fs.h와 disk.img, disk.img.hex이다. disk.img는 현재 프로젝트에서 사용할 수 있는 파일 시스템 이미지 binary이고, disk.img.hex는 binary를 hex로 읽어낸 파일이다. fs.h는 이 프로젝트 구현에 기본 조건이 되는 header이다.

fs.h는 define values, super block structure, inode structure, blocks structure, partition structure, dentry structure로 이루어져 있다. 구현의 시작단계에서 전체 disk.img를 읽어 fs.h의 구조에 맞게 출력하고, 이 형태가 맞는지 disk.img.hex를 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.

|  |  |  |
| --- | --- | --- |
| Super Block | Inode Array | Data Block |

따라서, 이 형태에 맞게 전체 disk.img를 읽어서 출력하면 아래와 같은 결과를 얻을 수 있다. 아래의 결과는 disk.img.hex를 통해 검증을 마친 결과이다. fs.h와 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.txt로 저장하여 레포지토리에 추가하였다.

**구현 결과**

먼저 전체 파일과 그 설명은 아래와 같다

2022-os-proj3

┗ disk.img : 주어진 disk image file

┗ disk.img.hex : 주어진 disk image file의 hex dump

┗ file\_system.c : file\_system을 mount하고 읽는 함수의 구현

┗ file\_system.h : mount후 사용할 mount env 구조체, file\_system.c의 함수들

┗ fs.h : 주어진 file system 구조 헤더

┗ hash\_set.c : 빠른 inode 탐색을 위한 hash set 구현

┗ hash\_set.h : pair, node, set 구조체

┗ LICENSE

┗ main.c : env를 세팅하고 shell을 실행하는 main 함수

┗ Makefile

┗ project3.pdf

┗ README.md

┗ result.txt : disk.img를 fs.h의 형태에 맞게 읽은 결과

┗ SiSH

┗ (Shell 관련 file들)

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

shell에서 mount와 연산을 담당하는 코드 → mount ls, cat, pwd 함수 코드 → hash set코드

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 block의 시작점 offset을 저장

    int block\_finishing\_point; // 편의를 위해 block의 종료점 offset을 저장

    FILE\* disk\_img; // disk.img의 파일 포인터

    Directory\* current\_directory; // 현재 directory를 표현하는 문자열 배열

    BST\_set\* set; // inode 탐색을 위한 binary search set (with hash)

    DIR\_FILES\* dir\_files; // 편의를 위해 현재 directory의 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\_env를 미리 할당하여, mount 후 사용할 수 있도록 한다.

(중략)

Mount시 host name, current path 등을 다른 방식으로 update해야하므로 unmount 시에만 shell의 함수를 사용하여 update한다.

        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);

        }

Mount시 사용할 수 있는 명령이 제한되고, 별도의 구현이 필요하므로 mount 여부를 env에서 관리하고, 그에 따라 다른 동작을 하도록 구현한다.

        if (!strcmp(command, "quit") || !strcmp(command, "exit")) {

            //fprintf(stdout, "\n");

            exit(EXIT\_SUCCESS);

        } // quit 과 exit는 mount여부에 상관없이, shell을 종료하도록 if 밖에 배치

        if (!(env->mount)) { // unmoun의 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.img의 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 set의 root, size 등을 초기화

                read\_block(first\_inode, mount\_env); // root directory에 해당하는 inode를 통해 해당하는 data block을 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\_block함수를, ls시 ls함수를, cat시 cat함수를, pwd시 current\_dir\_to\_string 함수를 사용한다. 각각의 함수는 아래와 같다. (current\_dir\_to\_string함수는 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 block과 inode table에 disk.img의 값을 읽어서 저장하는데, 이 함수를 호출 시 mount env에 속한 super block과 inode table을 사용하여 전체 실행에서 mount env를 통해 이 값들을 사용할 수 있도록 한다.

이후 첫 번째 inode를 사용하여 read\_block을 통해 root의 data block을 읽는다. 다른 inode를 읽을 때도 마찬가지로 read\_block이 사용되는데 그 코드는 아래와 같다.

<file\_system.c>

void\* read\_block(Inode inode, MountEnv\* mount\_env) {

// directory인 경우에는 mount env를, 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 dentry를 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) {

(Dentry에 속하는 변수들에 맞게 읽는 부분 생략)

                if (inode\_num \* dir\_length \* name\_len \* file\_type == 0) {

                    continue; // invalid 한 경우 뛰어넘기

                }

(valid 하다면 directory를 읽는 부분 생략)

                int hash = hash\_func(directory\_name, name\_len);

// directory\_name을 hash\_func에 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 num을 쌍으로 하는 pair에 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 hash로 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.

보통 해시함수는 r과 M이 서로소로 정하기 때문에, 여기서 r과 M은 모두 소수를 사용하였다. r은 31로, M은 2,147,483,647로 정하였는데 이는 각각 , 이며 특히 M은 일반적인 C언어에서 signed integer의 limit이다.

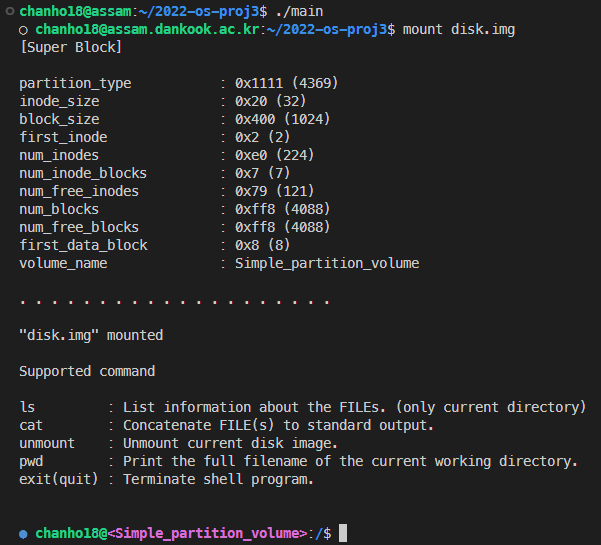
따라서 최종 수식은

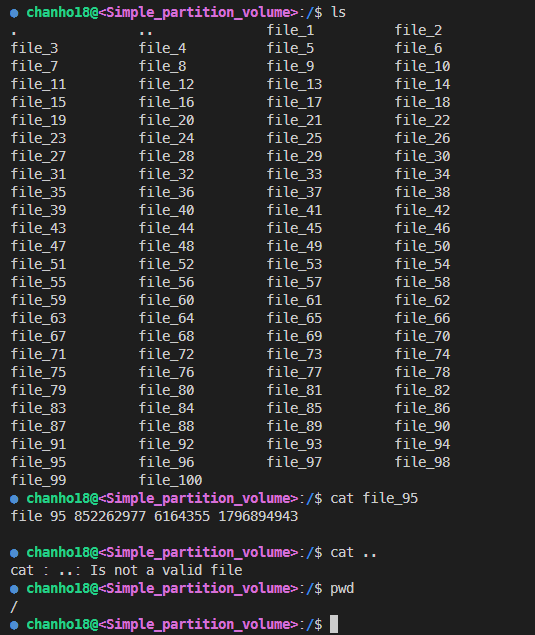
와 같고 연산 상의 최적화를 위해, 매 합에서 31을 곱하는 것을 left bit shift 5를 통해 32를 곱하고 hash를 빼는 것으로, 그리고 전체 합의 mod를 매번 합할 때 마다 mod를 수행한 값(나머지)을 합하는 것으로 대체한다.

Insert node와 find node 는 일반적인 binary search tree와 동일하므로 생략한다.

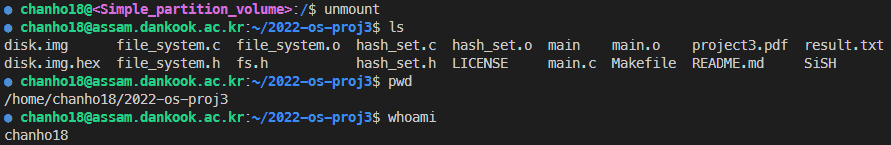
dentry를 통해 얻어낸 file의 이름을 ls로 저장한 후, 그 이름을 통해 file의 inode를 찾고자 할 때 dentry를 그대로 사용하게 되면 전체 dentry배열을 탐색하며 현재 file 이름과 dentry의 이름을 계속 비교하게 된다. 이 경우 전체 탐색은 O(n)의 시간 복잡도를 요구하며, 또한 매 반복 마다 string 비교를 수행해야 하므로 탐색의 cost가 크다. 그러나 위의 방식을 사용할 경우, binary search tree를 사용하므로 탐색 시의 시간 복잡도는 O(log n)이며, 탐색 시에 값의 비교 역시 hash 값이라는 numeric 값을 비교하므로 string의 길이에 무관하게 한 번의 비교만 수행하면 된다. 따라서 cost에 있어 많은 개선이 이뤄진다.

**실행 결과**

Shell 실행과 mount. Mount 시의 host는 현재 volume의 이름으로 하고, 다른 색깔로 표현한다.

cat, ls, pwd 실행

unmount 실행 및 unmount 시의 정상 shell 동작



**결론**

**회고**

본 프로젝트를 수행하며 간단한 형태의 file system을 읽고 shell과 연결하며 file system의 구조를 이해할 수 있었다. 전체 구조에 대한 이해를 바탕으로 최초에는 write 구현에 시도하였으나, 현재의 disk image를 수정하여 테스트하는 부분에서 free block bitmap을 구성하여 disk image를 수정하는 것보다 전체 file system image를 새로 생성하는 것을 시도하는 것이 좋겠다는 생각이 들었다. 이에 전체 file system을 구성하여 free block bitmap을 구성하는 것까지는 성공했으나, 이를 image file로 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 directory를 다룰 수 있도록 코드를 구현하였으나, 실제 이를 테스트할 수 있는 케이스가 없어 코드가 제대로 동작하는지 확인하지 못한 것이다. 이러한 부분에서 여러 종류의 disk image가 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**