**OS Project 3**

**FUSE File System**

**Implementation**

컴퓨터공학과 2014313366 홍기원

컴퓨터공학과 2014312406 심하영

**CONTENTS**

1. **개요 ······································································ 3**
2. **파일 시스템 설계 내용 ·············································· 4**

**2-1) 구현 환경** **······················································· 4**

**2-2) 파일 데이터 관리 방식 및 자료구조 ························ 4**

1. **구현 기능 설명 ························································ 7**

**- Initialization ······················································· 7**

**- Get Empty Space ················································ 7**

**- Set Empty Space ················································ 7**

**- Get Index ···························································· 8**

**- Insert / Delete Directory or File ·························· 8**

**- Make / Remove / Read Directory ························ 9**

**- Make / Remove / Read / Write File ··················· 10**

**- Get Attributes ··················································· 11**

**- Open File or Directory ······································ 11**

**- 추가 기능 구현 ···················································· 12**

1. **실행 결과 ····························································· 13**

**1. 개요**

FUSE란 File System in User Space, 즉 유저 공간에서 커널 코드를 건드리지 않고도 파일 시스템을 만들 수 있도록 도와주는 소프트웨어 인터페이스이다.

기존의 파일 시스템에서 사용자가 read라는 함수를 사용하는 경우를 생각해보자. 이 때 system call을 통해 sys\_read라는 함수가 호출되고, 이는 다시 VFS의 vfs\_read함수를 호출하여 마지막으로 커널 영역의 file\_operation에 등록되어 있는 read 함수를 호출한다. 그런데 FUSE를 사용하면 커널 영역의 file\_operation이 아닌, 유저 공간의 FUSE의 file\_operation에서 함수가 호출된다. 기존의 파일 시스템은 커널 공간에서 동작하기 때문에 복잡하고 디버깅이 어려운 반면 FUSE는 아래와 같은 장점들을 가진다.

1. 유저 공간에서 동작하기 때문에 디버깅이 쉽다.
2. 간단한 설치, 간단한 API library를 가진다.
3. 어플리케이션 레벨에서 작업이 이루어지기 때문에 안전성이 높다.
4. 루트권한이 없는 사용자도 사용 가능하다.
5. 여러 언어, 여러 운영 체제에서 동작 가능하다.

( 단점 : 계층이 추가됨으로 인해 성능 저하가 발생함 )

FUSE가 동작하기 위해서 사용자는 파일 시스템을 만들고 연산들을 file\_operation에 제공해야 한다.

**2. 파일 시스템 설계 내용**

**2-1) 구현 환경**

OS: Ubuntu v.15.04

FUSE: v.26

**2-2) 파일 데이터 관리 방식 및 자료구조**

우리는 수업 시간 중 배운 파일 시스템의 구성 요소, 구조 등의 개념을 적용하여 파일 시스템을 만들고, 여러 가지 operation들을 직접 구현해보았다. 우리가 설계한 파일 시스템은 계층적인 디렉토리 구조를 가지고, 파일들은 이러한 디렉토리에 block 단위로 non-contiguous하게 저장된다.

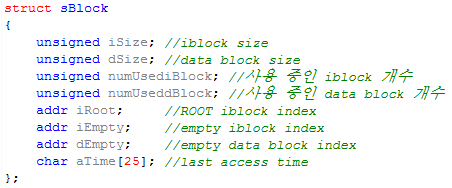
파일의 전체적인 구조는 다음과 같다.

* inode block과 data block의 index는 unsigned short 형 변수이다. (인덱스는 음수가 아니므로 unsigned를 사용하였다) unsigned short는 총 2^16개의 수를 표현가능 한데, 이중 한개의 숫자(2^16)을 특수한 상황에 쓰기위한 수로 예약해두고 나머지 2^16-1개를 최대 블록의 개수로 잡았다. 따라서 inode와 data block의 최대 수는 각각 0xFFFE, 즉 2^16-1개이다.



1. super block은 1개로, 한 블록의 크기는 48 Byte이다.

전체 파일 시스템의 메타 데이터를 담고 있다.



1. inode block은 2^16-1개로, 한 블록의 크기는 256 Byte이다.

각각의 파일이나 디렉토리의 정보, data block들의 위치와 같은 정보를 담고 있다.

구조체 descriptor에서 부모, 형제, 첫 번째 자식의 inode index 정보를 제공하여

계층적인 디렉토리 구조에서의 조작을 쉽게 하였고,

stat.h에 정의된 구조체 stat에 파일의 기본적인 정보를 저장하도록 하였다.

이번 프로젝트에서 사용한 구조체 stat의 내용은 아래와 같다.

struct stat

{

dev\_t st\_dev; /\* device \*/

mode\_t st\_mode; /\* protection \*/

nlink\_t st\_nlink; /\* number of hard links \*/

uid\_t st\_uid; /\* user ID of owner \*/

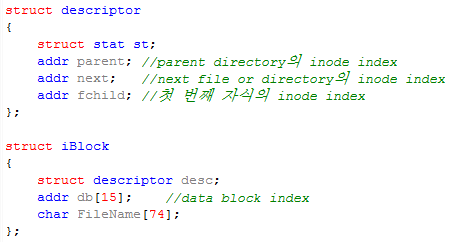
gid\_t st\_gid; /\* group ID of owner \*/

off\_t st\_size; /\* total size, in bytes \*/

time\_t st\_atime; /\* time of last access \*/

time\_t st\_mtime; /\* time of last modification \*/

}



1. Data block은 2^16-1개로, 한 블록의 크기는 256 Byte이다.

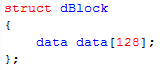
실질적으로 데이터가 저장되는 공간이고, 파일들은 block 단위로 나누어져서 non-contiguous하게 저장된다.

data block들은 direct block 12개와 각각 하나의 single, double, triple indirect block들로 이루어진다.

각 direct block pointer은 data block 1개를 가리킨다. single, double, triple block pointer는 또 다른 block을 가리켜 그 block에서부터 다른 data block을 가리킬 수 있게 한다. single, double, triple block이 가리키는 이 block은 data block으로 하는 게 큰 파일 용량과 많은 파일의 개수를 가질 수 있게 하는 것이겠지만, 그렇게 큰 파일 용량이나 개수가 필요하지는 않으므로 단순하고 관리하기 쉽게 inode block을 가리키는 것으로 하였다.

총 12+15+15\*15+15\*15\*15 = 3627개의 파일 및 디렉토리를 가질 수 있고,

저장 가능한 한 파일의 최대 용량은 약 3627\*256 = 928512바이트가 된다.



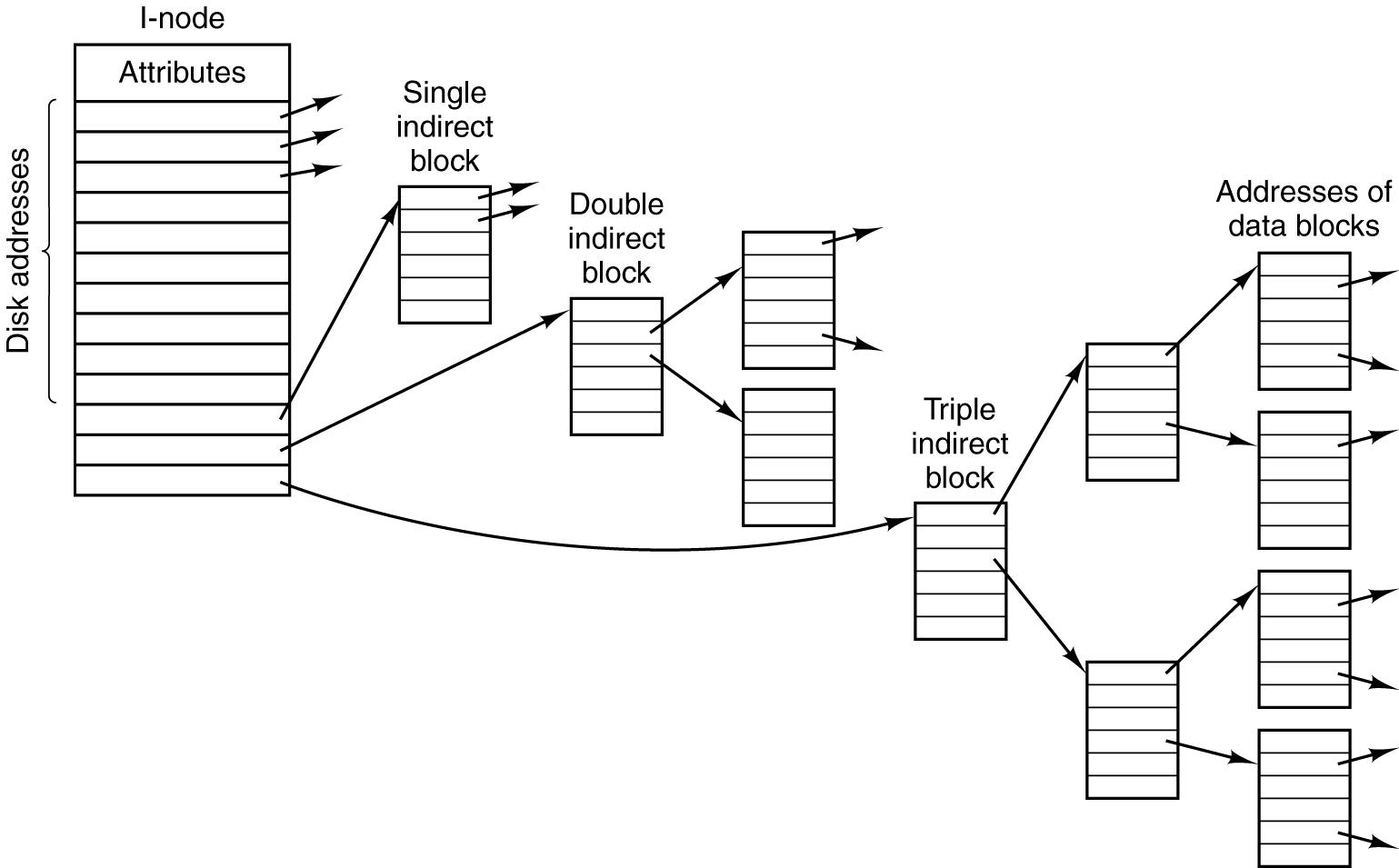


Figure 1 http://cs.sru.edu/~whit/cpsc464/Notes/ch4.html

* 효율적인 구조체의 선언을 위해서는 구조체의 멤버들의 순서를 잘 배열할 필요가 있다. (구조체에서 데이터 구조 패딩이 일어나기 때문)

따라서 이번 프로젝트에서는 최대한 공간을 효율적으로 사용하는 구조체를 만들기 위해 노력하였다.

**3. 구현 기능 설명**

**Initialization**

1. void init\_storage()

super block, root inode, data block의 정보를 초기화한다.

처음에는 모든 inode가 비어 있으므로, inode는 데이터 블록을 가리키지 않고 inode[0].db[0] = 1, inode[1].db[0] = 2 등과 같이 다음의 empty inode를 가리킨다.

data block도 마찬가지의 방법으로 초기화 해주고,

inode와 data block의 가장 마지막 블록은 NO\_NEXT\_BLOCK의 값을 가리키도록 한다.

1. void load\_storage(FILE \* fp)

file pointer가 가리키는 곳의 정보를 super block, inode, data block으로 읽어들인다.

**Get Empty Space**

1. addr get\_empty\_inode()

super block으로부터 empty inode의 index 값을 얻어서 반환하고,

super block은 next empty inode, 즉 원래의 empty inode가 가리키던 블록의 index 값을 갖도록 한다.

1. addr get\_empty\_dblock()

super block으로부터 empty data block의 index 값을 얻어서 반환하고,

super block은 next empty data block, 즉 원래의 empty data block이 가리키던 블록의 index 값을 갖도록 한다.

**Set Empty Space**

1. void set\_empty\_inode(addr idx)

idx의 위치에 있는 inode를 empty inode로 만든다.

inode의 부모 노드는 ROOT(0), 형제와 자식 노드는 NO\_NEXT\_BLOCK의 값을 갖게 하는 것으로 간단히 구현 가능하다. (empty inode라는 표시)

1. void set\_empty\_dblock(addr idx)

idx의 위치에 있는 data block을 empty data block으로 만든다.

**Get Index**

1. addr get\_path(char\* path)

계층적인 디렉토리 구조에서의 path는 ROOT node에서부터 ‘/’를 기점으로 한 칸씩 하위의 계층으로 내려간다. 주어진 path에 위치하는 node의 index를 찾기 위해, ROOT node에서부터 시작하여 해당 계층에서 주어진 path의 폴더(혹은 파일) 이름과 같은 것이 있으면 한 칸씩 하위의 계층으로 내려가며 계속 찾아나가고,

같은 이름의 폴더(혹은 파일)이 존재하지 않으면 에러를 반환한다.

1. addr get\_data\_index(addr target, int i)

target inode의 i번째 data block의 index를 구하는 함수이다.

target inode의 1~12번째 data block은 바로 접근해서 index를 구할 수 있는 반면

13~15번째의 single, double, triple indirect data block의 index를 구하기 위해서는 경우를 나누어 추가적인 연산을 해야 한다.

1. int put\_data\_to\_index(addr tinode, addr newdblock, int i)

target inode의 i번째 node에 새로운 data block을 할당시키는 함수이다.

target inode의 1~12번째 블록은 곧바로 data block을 가리킬 수 있는 반면

13~15번째의 single, double, triple indirect data block을 가리키게 하기 위해서는 경우를 나누어 추가적인 연산을 해야 한다.

새로운 data block이 제대로 할당되었으면 0을 리턴한다.

**Insert / Delete Directory or File**

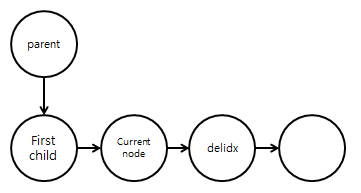
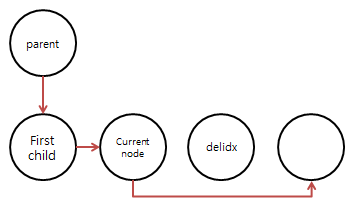
1. void insert\_dirorfile(addr parent, addr newidx)

parent의 하위 폴더(혹은 파일)에 newidx를 삽입하는 함수이다.

만약 parent에 하위 폴더(혹은 파일)이 없다면 새로 만들어주고,

있다면 마지막 자식 노드의 형제 노드로 삽입해준다.

1. void delete\_dirorfile(addr delidx)

delidx라는 폴더(혹은 파일)을 삭제하는 함수이다.

delidx의 부모 노드의 첫 번째 자식 노드에서 검사를 시작한다.

만약 현재 노드의 다음 노드가 delidx를 가리킨다면

현재 노드가 delidx가 가리키는 노드를 가리키도록 한 후,

set\_empty\_inode 함수를 이용하여 delidx를 empty node로 만들어 링크를 끊어버린다.

**Make / Remove / Read Directory**

1. int fuse\_mkdir(const char\* path, mode\_t flag)

fuse\_mkdir은 주어진 경로에 새로운 폴더를 만드는 역할을 한다.

먼저 새로운 inode dir을 선언하고, 사이즈, 접근시간, 연결된 링크 개수 등 stat 정보들을 모두 초기화한다. ‘폴더’이므로 st\_mode는 S\_IFDIR | S\_IRWXU | S\_IRWXG | S\_IROTH로 설정한다.

그리고 주어진 path의 ‘/’를 기점으로 ROOT inode에서부터 한 칸씩 하위 계층으로 내려가서 주어진 path의 마지막 위치까지 이동했다면 insert\_dirorfile 함수를 이용하여 폴더를 삽입한다.

폴더가 잘 만들어졌다면 0이 반환된다.

1. int fuse\_rmdir(const char\* path)

fuse\_rmdir은 주어진 경로에 있는 폴더를 제거하는 역할을 한다.

1. 주어진 경로의 길이가 너무 길거나
2. 주어진 경로가 empty block을 가리키거나
3. ROOT 폴더를 제거하려 하거나
4. 주어진 경로에 있는 것이 폴더가 아닌 경우 등에는 에러를 반환하고

그렇지 않은 경우에 delete\_dirorfile 함수를 이용하여 폴더를 제거한다.

1. int fuse\_readdir(const char \*path, void \*buf, fuse\_fill\_dir\_t filler, off\_t offset, struct fuse\_file\_info \*info)

해당 inode의 파일과 디렉토리 목록을 읽는다.

**Make / Remove / Read / Write File**

1. int fuse\_mknod(const char\* path, mode\_t flag, dev\_t dev)

fuse\_mknod는 주어진 경로에 새로운 파일을 만드는 역할을 한다.

먼저 새로운 inode dir을 선언하고, 사이즈, 접근시간, 연결된 링크 개수 등 stat 정보들을 모두 초기화한다. ‘파일’이므로 st\_mode는 S\_IFREG | S\_IRWXU | S\_IRWXG | S\_IROTH로 설정한다.

그리고 주어진 path의 ‘/’를 기점으로 ROOT inode에서부터 한 칸씩 하위 계층으로 내려가서 주어진 path의 마지막 위치까지 이동했다면 insert\_dirorfile 함수를 이용하여 파일을 삽입한다.

파일이 잘 만들어졌다면 0이 반환된다.

1. int fuse\_unlink(const char\* path)

fuse\_unlink는 주어진 경로에 있는 파일을 제거하는 역할을 한다.

1. 주어진 경로의 길이가 너무 길거나
2. 주어진 경로가 empty block을 가리키거나
3. 주어진 경로에 있는 것이 정규 파일이 아닌 경우 등에는 에러를 반환하고

그렇지 않은 경우에 delete\_dirorfile 함수를 이용하여 파일을 제거한다.

1. int fuse\_read(const char \*path, char \*buf, size\_t size, off\_t offset, struct fuse\_file\_info \*info)

fuse\_read는 주어진 경로의 파일을 읽어 오는 역할을 한다.

get\_data\_index 함수를 이용하여 target inode이 가리키는 data block들을 buffer에 저장한다.

1. int fuse\_write(const char \*path, const char \*buf, size\_t size, off\_t offset, struct fuse\_file\_info \*info)

fuse\_write는 주어진 경로의 파일에 새로운 내용을 적는 역할을 한다.

파일에 적을 내용을 담고 있는 buffer의 사이즈를 재면 총 몇 개의 data block이 필요한지 알 수 있다.

필요한 data block의 개수만큼 get\_empty\_dblock() 함수를 이용하여

빈 data block을 할당 받아 내용을 write하고, put\_data\_to\_index() 함수를 이용하여 data block들을 inode가 가리키도록 한다. fuse\_write 함수를 실제로 사용하기 위해 fuse\_mknod 함수의 맨 마지막에서 fuse\_write 함수를 호출하였다. 즉, 파일이 생성되고 난 후 내용을 쓰도록 하여 fuse\_write 함수가 실제로 동작함을 보일 수 있게 하였다.

**Get Attributes**

1. int fuse\_getattr(const char \*path, struct stat \*statbuf, fuse\_file\_info \*info)

fuse\_getattr은 주어진 경로에 있는 폴더(혹은 파일)의 속성을 리턴한다.

파일의 속성은 구조체 stat 내에 정의되어 있고, st\_size, st\_mode, st\_nlink, st\_uid, st\_gid, st\_atime, st\_mtime의 정보를 가진다.

1. int fuse\_chmod(const char\* path, mode\_t newmode, struct fuse\_file\_info \*info)

주어진 경로에 있는 폴더(혹은 파일)에 대한 mode를 주어진 newmode로 바꾼다.

S\_IFMT : type of file

S\_ISDIR : directory

S\_ISCHR : character special file

S\_ISBLK : block special file

S\_ISREG : regular file

S\_ISFIFO : FIFO special file, or a pipe

S\_ISLNK : symbolic link

S\_ISSOCK : socket

**Open File or Directory**

1. int fuse\_open(const char \*path, struct fuse\_file\_info \*fi)

주어진 경로의 파일이 OPEN 가능하면 0을, 그렇지 않은 경우 에러를 리턴한다.

파일을 읽을 수 없는 경우는 주어진 경로가 empty block을 가리키거나, 파일 형식이 아니거나, 접근 권한이 없는 경우이다.

(접근 권한이 있는 경우는 O\_RDONLY, O\_WRONLY, O\_RDWR)

1. int fuse\_opendir(const char \*path, struct fuse\_file\_info \*fi)

주어진 경로의 폴더가 OPEN 가능하면 0을, 그렇지 않은 경우 에러를 리턴한다.

폴더를 읽을 수 없는 경우는 주어진 경로가 empty block을 가리키거나, 폴더 형식이 아니거나, 접근 권한이 없는 경우이다.

(+) 추가적으로 구현한 기능은 다음과 같다.

* 먼저, 상황에 맞는 에러를 사용하였다. 함수를 수행하다가 오류가 발생하면 상황에 따른 에러 값이 리턴되어, 구체적인 오류의 발생 여부를 확인할 수 있다.

해당 프로그램에서 사용한 에러 상수의 종류는 아래와 같다.

ENAMETOOLONG : file name too long

ENOMEM : out of memory

ENOENT : no such file or directory

ENOTDIR : not a directory

EPERM : operation not permitted

ENOTEMPTY : directory not empty

EISDIR : is a directory

EACCESS : permission denied

* 파일 시스템에 마지막으로 접근한 시간을 super block에 저장하여, 파일 시스템을 실행할 때마다 출력하도록 하였다.
* 사용 중인 inode block과 data block의 수를 각각 super block에 저장한다. get\_empty\_inode()와 get\_empty\_dblock()이 실행될 때마다 numUsediBlock, numUseddBlock의 수를 1씩 증가, set\_empty\_inode()와 set\_empty\_dblock()이 실행 될 때는 1씩 감소시켰다. 이를 파일 시스템을 실행할 때마다 출력하여 보여줌으로써, 파일 시스템에 어느 정도의 공간이 남았는지 사용자가 알 수 있도록 하였다.
* int fuse\_chmod(const char\* path, mode\_t newmode, struct fuse\_file\_info \*info)

파일의 r/w/x 권한을 폴더나 파일이 생성될 때 부여하고, 사용자가 fuse\_chmod 함수를 이용하여 권한을 바꿀 수 있도록 하였다.

**4. 실행 결과**

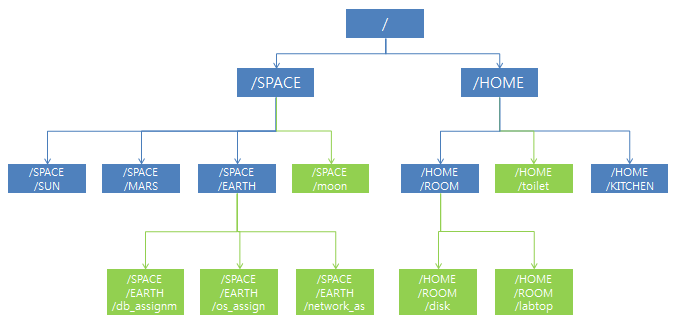
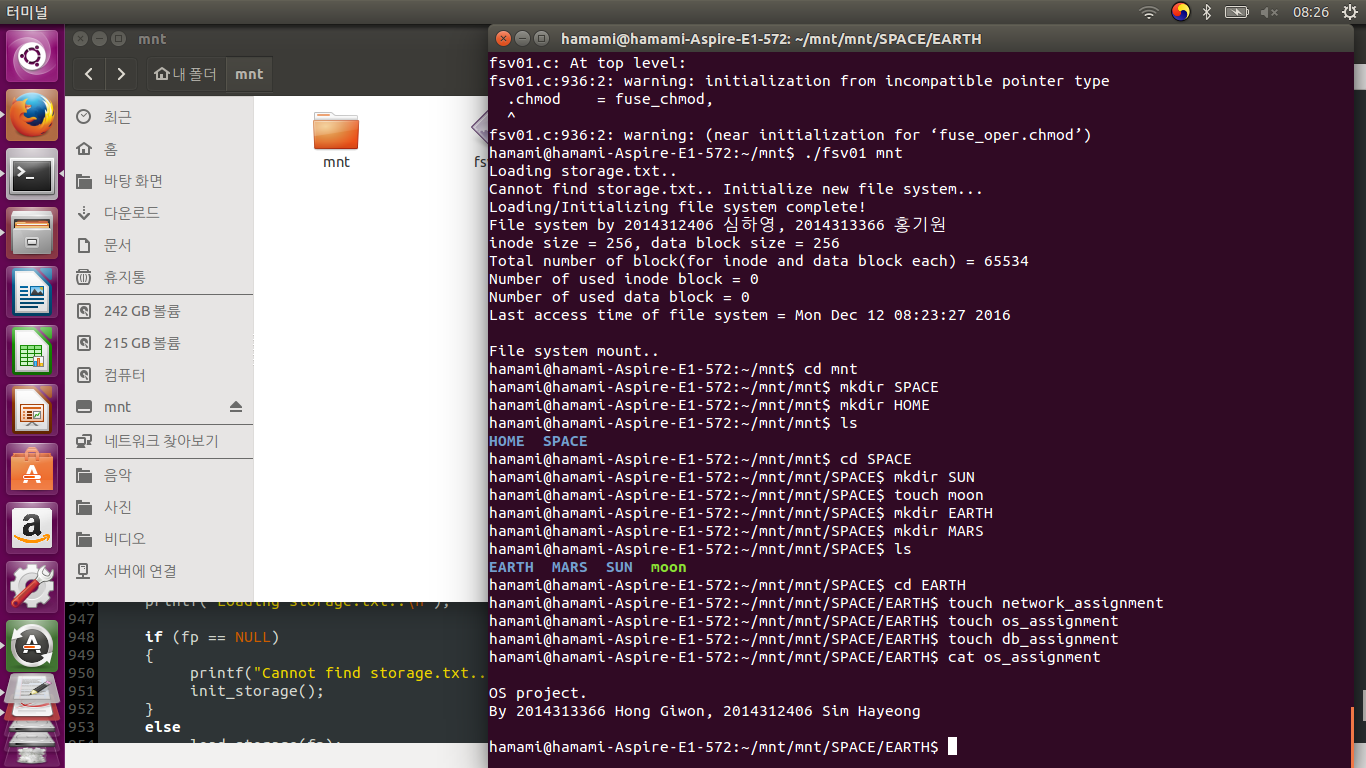


Figure 2

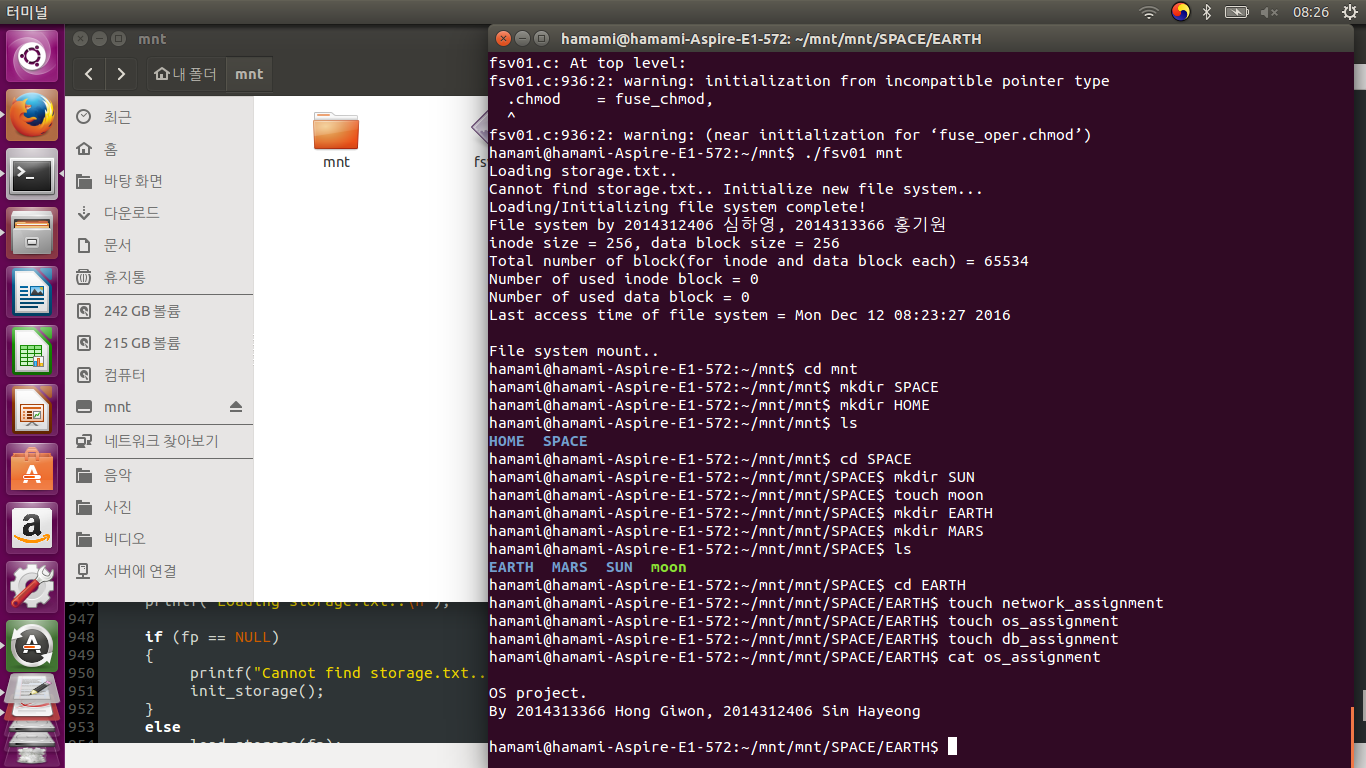
파일 시스템의 계층적인 구조를 보이기 위해, Figure 2와 같이 폴더와 파일을 구성해보았다.



처음 file system을 실행했으므로 initialize 해준다.

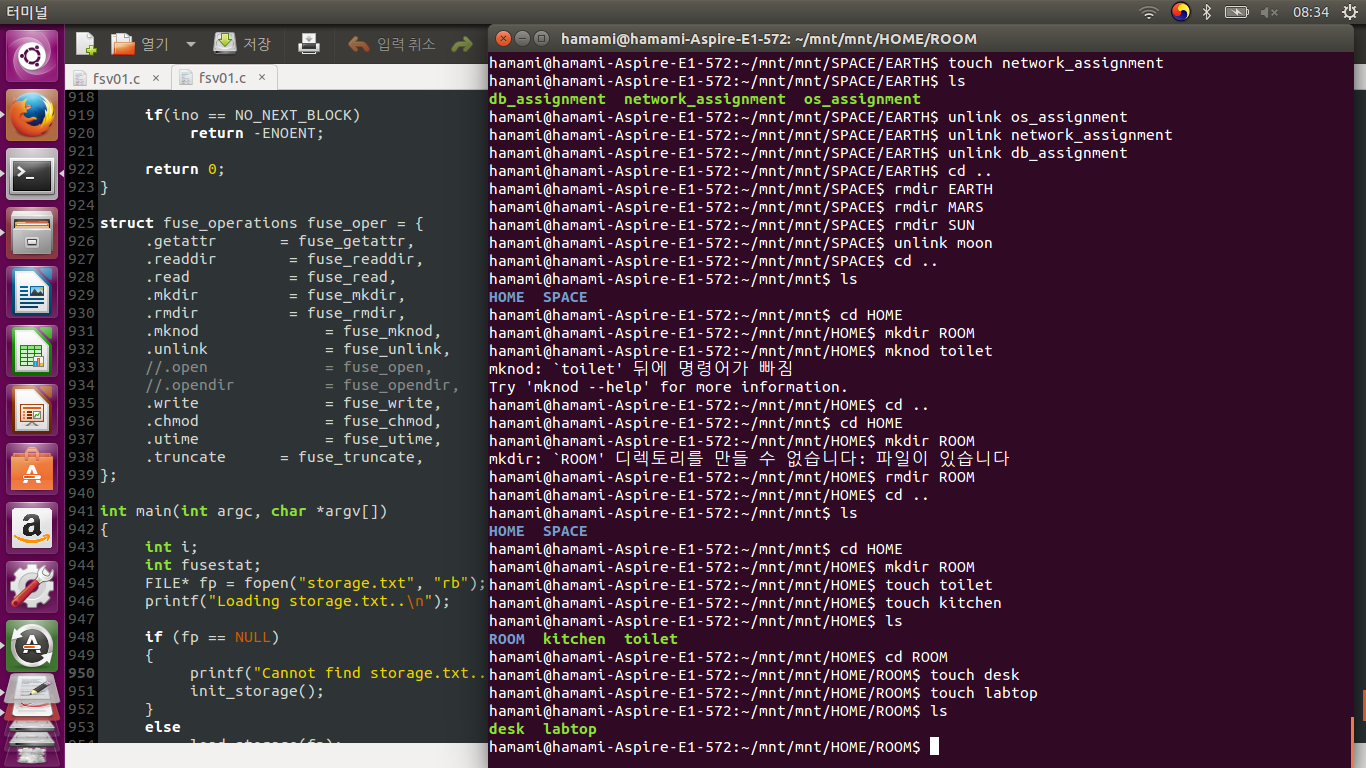
inode, data block의 크기와 개수, 사용 중인 블록의 수가 출력된다..

가장 마지막으로 파일 시스템에 접근했을 때의 시간도 출력된다.



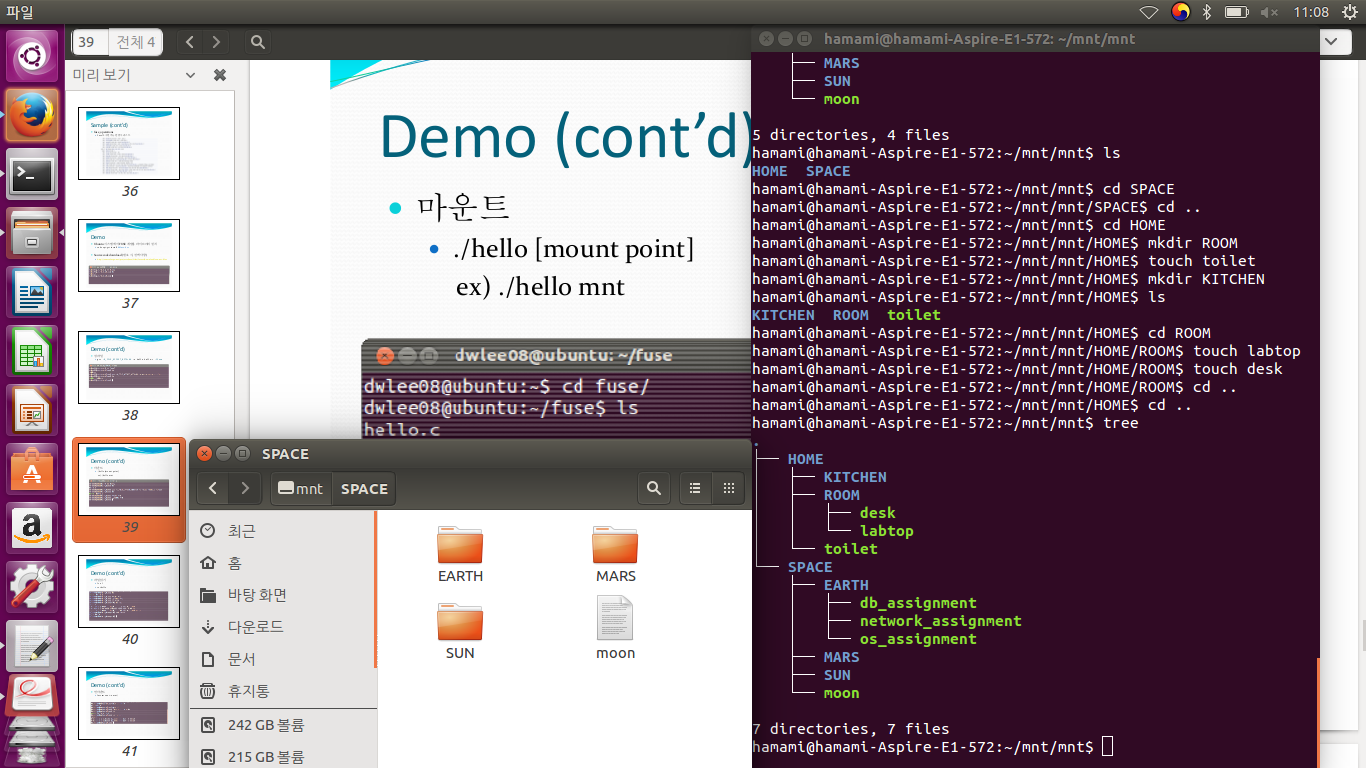
/SPACE의 하위 디렉토리에 /SUN, /MARS, /EARTH, /moon을 만든다.

/EARTH의 하위 디렉토리에 /network\_assignment, /os\_assignment, /db\_assignment를 만들고, os\_assignment를 오픈한다.



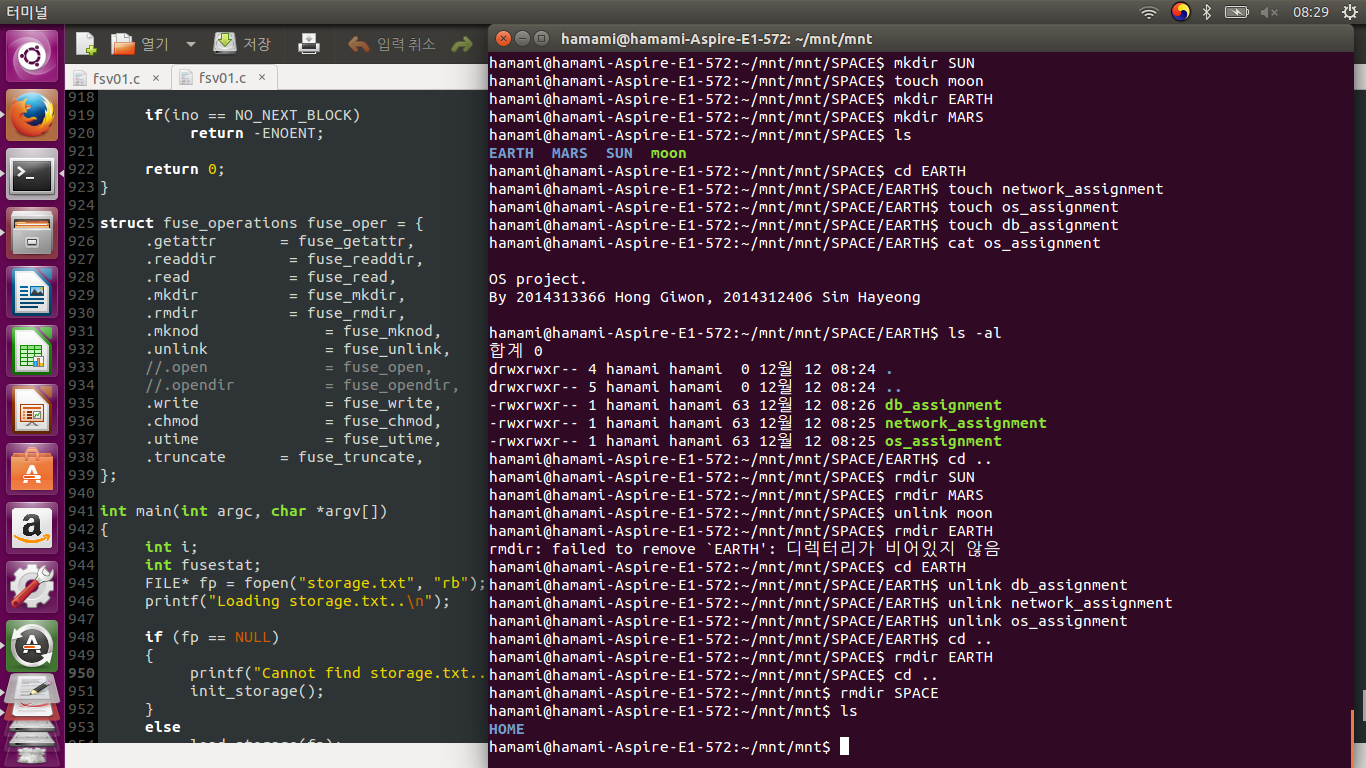
/HOME의 하위 디렉토리에 /ROOM, /toilet, kitchen을 만든다.

/ROOM의 하위 디렉토리에 /desk, labtop을 만든다.



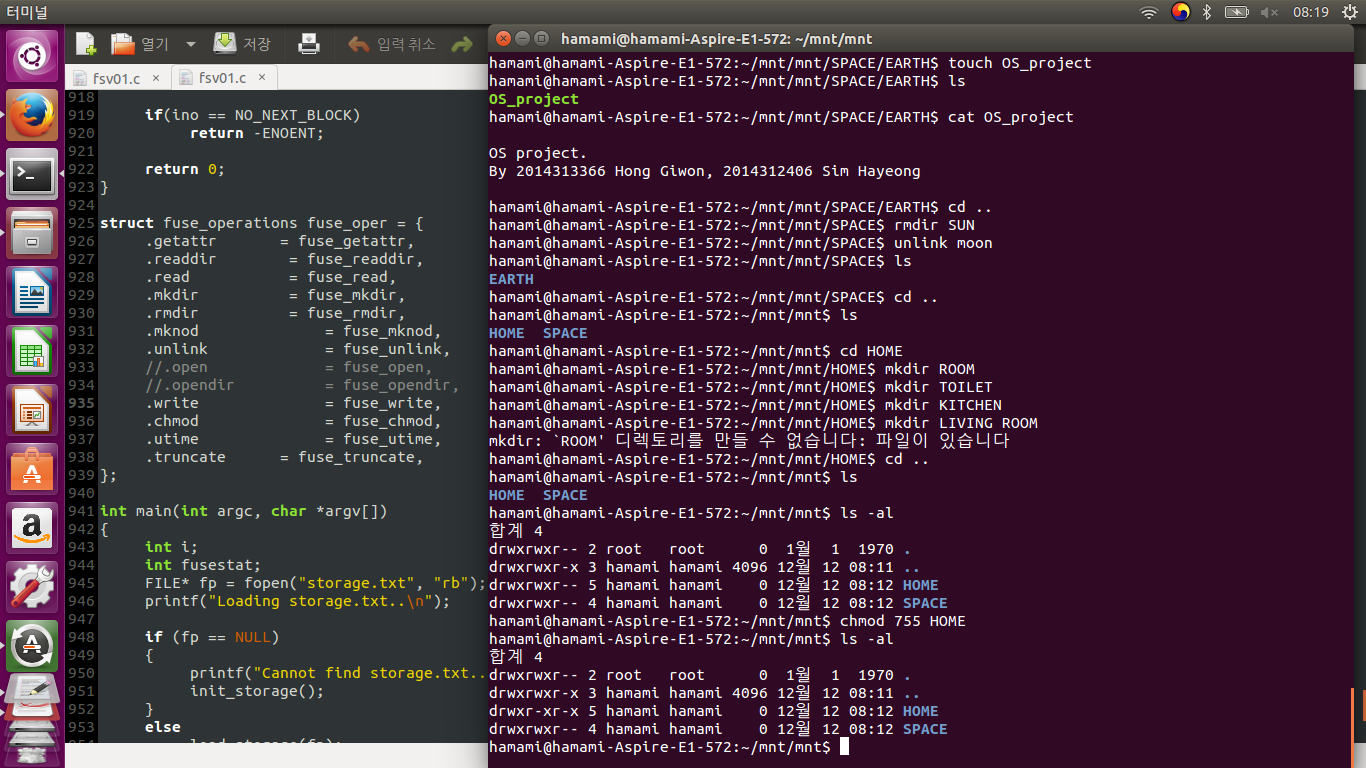
tree 명령어를 사용하여 디렉토리의 전체 구조를 확인할 수 있다.

그리고 /SPACE 하위에 실제로 파일들과 폴더가 생성된 것을 확인할 수 있다.

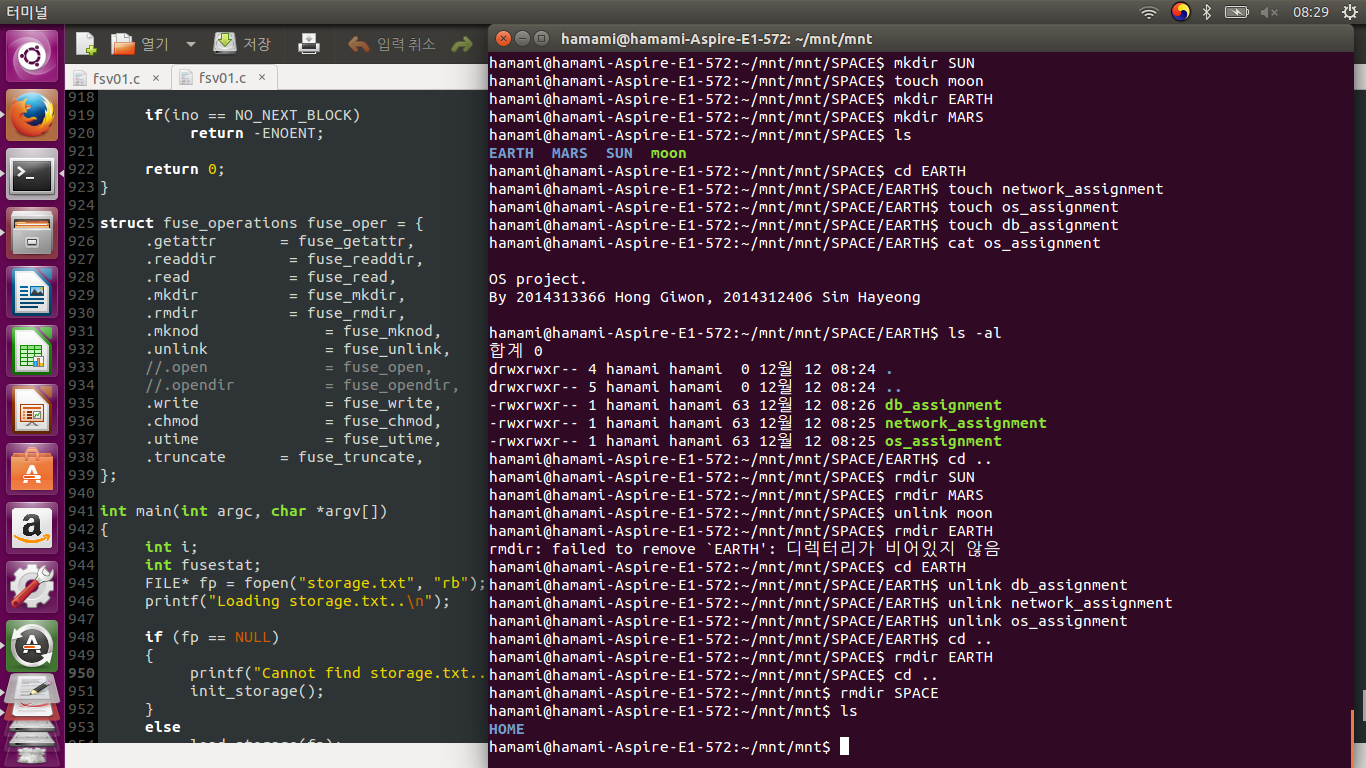


/SPACE/EARTH의 파일 정보를 보인다.

접근 권한과 상위 디렉토리, 최종 접근 시간 등의 정보가 출력된다.



chmod를 사용하여 HOME의 접근 권한을 수정하였다.



/SPACE와 그의 하위 디렉토리에 존재하는 폴더와 파일을 모두 삭제하였다.