REPORT

FUSE based File System Implementation 보고서

2015년 2학기 운영체제론(CSE3008-41)
엄영익 교수님

2015년 12월 8일
OSHW #2

컴퓨터공학과
2013312448 이수영
2013312707 이수인

목차

1. 개발 환경

(1) OS

Ubuntu 14.04 (64bit)

(2) 컴파일러

gcc (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4

(3) 컴파일 및 방법

```
suinee@ubuntu:~/Desktop/OSHW2$ gcc -D_FILE_OFFSET_BITS=64 -o ssf_final ssf_final
.c -lfuse
suinee@ubuntu:~/Desktop/OSHW2$ ./ssf_final mnt
```

Figure 1 컴파일 및 실행

컴파일 시 발생하는 warning의 경우 무시 가능하다.

2. SSF(Sooyoung-Suin File system) 소개

(1) 기본 설계

우리가 설계했던 SSF의 모형은 다음과 같다.

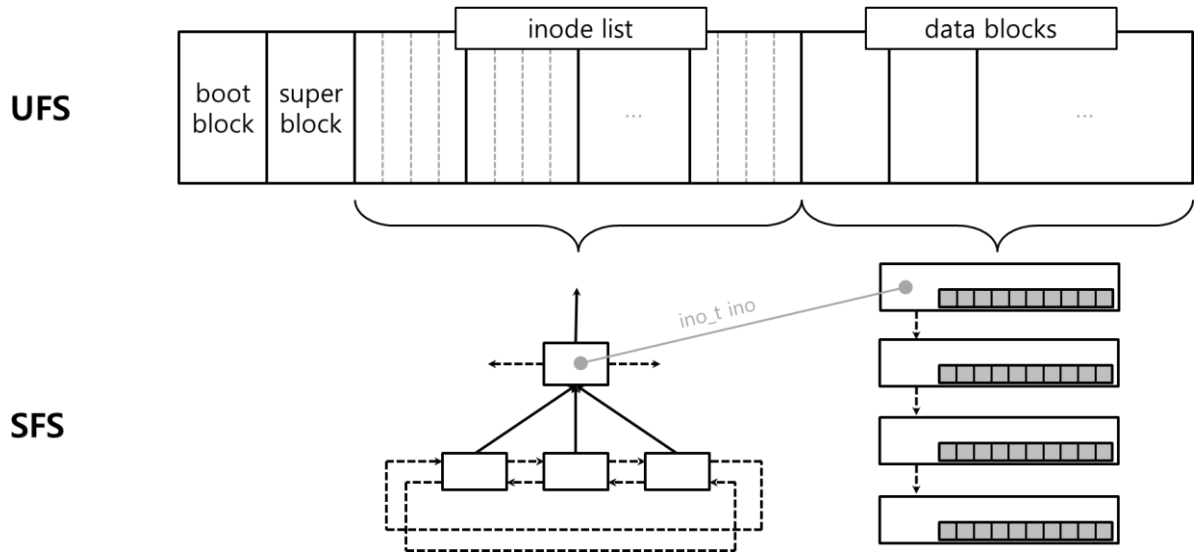

Figure 2 SSF 설계 모형

(2) 변경 사항

설계한 모형을 바탕으로, 필요한 요소들을 덧붙여서 SSF를 구현하였다.

먼저 각 파일당 할당하여 inode의 역할을 하도록 하고 tree 구조를 이루는 struct snode struct snode를 수정하였다.

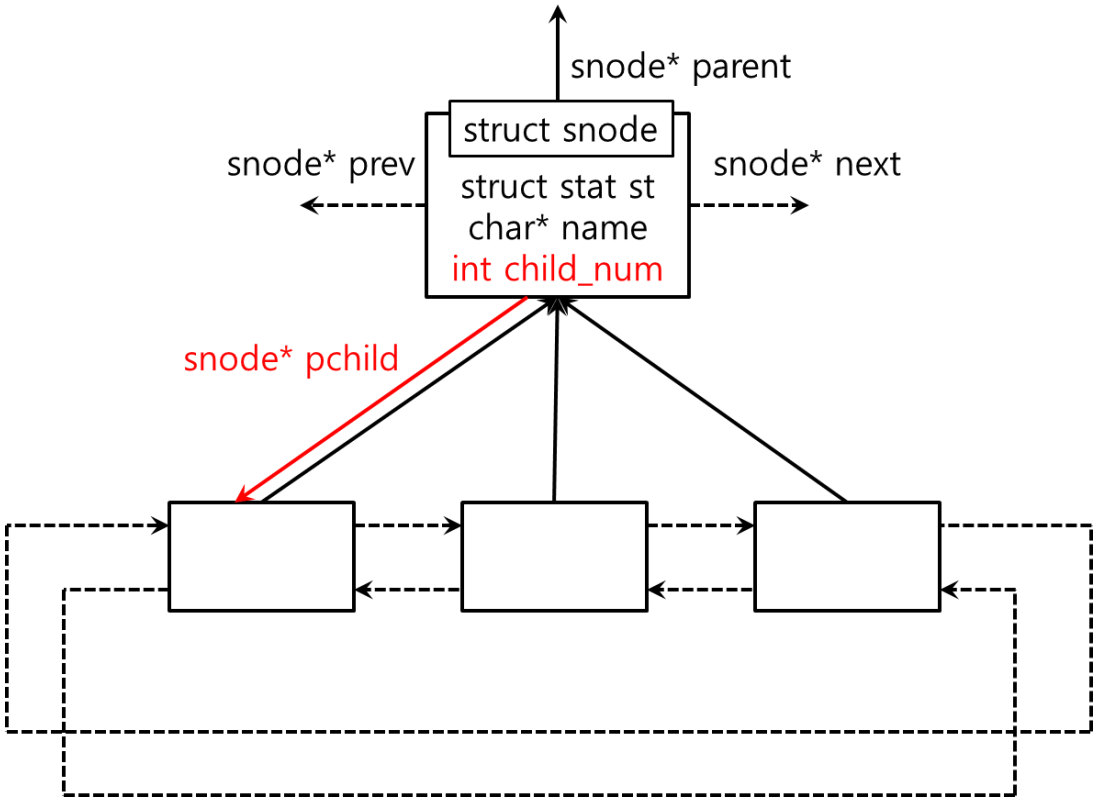

Figure 3 수정된 snode

- struct stat st : 실제 파일의 meta data를 저장하는 linux 제공 struct이다. inode number(ino_t st_ino), permission(mode_t st_mode), 파일의 크기(off_t st_size), hard link의 수(nlink_t st_nlink) 등을 포함한다. 특히 inode number는 각 파일마다 unique한 값을 갖는다.

- snode* prev, snode* next, snode* parent : 각 snode를 tree 구조로 연결하는 데 사용되는 pointer이다. prev와 next로 children끼리 연결되며, 원형으로 연결되어 node를 search할 때의 편리성을 추구한다.

- char* name : 해당 파일의 이름을 저장한다.

- int child_num : 새로 추가한 변수로, 해당 snode에 연결되어 있는 children의 수를

저장한다.

- snode* pchild : 새로 추가한 변수로, children 중 가장 첫 번째 child를 가리킴으로써 parent에서 child로의 접근을 용이하게 하였다.

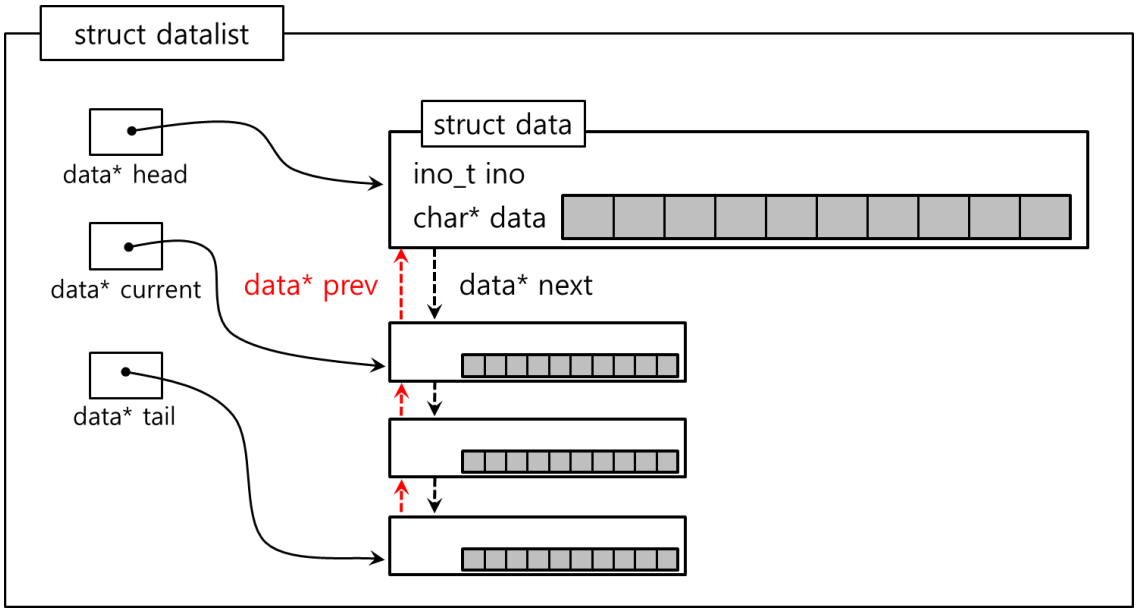

Figure 4 수정된 data와 datalist

data block은 data* prev 변수를 추가하여 double linked list로 구현하고, struct snode의 inode number를 불러와 저장함으로써 각 inode number에 해당하는 data를 저장, search 할 수 있도록 하였다. 이 과정에서 data들간의 보다 편리한 탐색이 가능해졌다.

또한, struct data들을 쉽게 관리하기 위하여 struct들을 감싸는 형태의 struct datalist를 만들었다. 포인터 변수들을 추가하여 data linked list의 의미있는 지점을 나타내고 있다.

3. SSF 구성

(1) struct 소개

1) struct _snode

```
13  struct _snode
14  {
15      char* name;
16      struct stat st;
17      int child_num;
18      struct _snode* pchild;
19      struct _snode* parent;
20      struct _snode* prev;
21      struct _snode* next;
22  };
23  typedef struct _snode snode;
24
```

Figure 5 struct _snode

앞에서 설명한 바와 같이 char* name은 해당 파일 또는 디렉토리의 이름이며, prev, next, parent와 pchild는 tree 구조로 연결하는데 사용되는 포인터이다. child_num은 해당 노드의 child 노드의 개수를 저장한다.

2) struct _data

```
27  struct _data
28  {
29      int st_ino;
30      char* data;
31      struct _data* prev;
32      struct _data* next;
33  };
34  typedef struct _data data;
```

Figure 6 struct _data

struct snode의 inode number를 불러와 st_ino에 저장하여 tree구조와 data를 연결한다. 파일 시스템에서 생성되는 파일에 들어갈 데이터가 저장되는 자료구조이다. char* data는 저장되는 데이터를 가리키는 포인터이며, prev와 next는 data들을 double linked list로 연결하기 위해 사용되는 포인터이다.

3) struct _datalist

```
36  struct _datalist
37  {
38      data* head;
39      data* current;
40      data* tail;
41  };
42  typedef struct _datalist datalist;
```

Figure 7 struct _datalist

Linked list를 보다 쉽게 구현하고 관리하기 위한 struct이다. head는 List의 가장 첫 data를 가리키는 포인터이고, current는 현재 data를 가리키며 search에 사용되는 포인터이다. tail은 list의 가장 마지막 data를 가리키고 data를 insert할 때 쓰인다.

(2) snode 와 관련된 함수

1) snode* make_snode(char* name, mode_t mode, uid_t uid, gid_t gid)

파일의 metadata를 저장하는 snode를 만든다. 파일의 이름과 읽기 전용, 쓰기 전용 등을 결정하는 mode_t mode, uid와 gid를 parameter로 가진다. 동시에 snode와 inode number를 공유하는 data 파일을 만들어 linked list에 넣어준다.

2) void insert_snode(snode* parent, snode* new)

make_snode로 만든 snode* new와 이 snode를 연결해 줄 부모 snode* parent를 parameter로 가진다. snode* parent가 디렉토리가 아닌 경우 error를 return하고, 디렉토리일 경우 child snode들의 수에 따라 tree 구조로 연결해 준다.

3) void delete_snode(snode* node)

삭제할 snode를 parameter로 가진다. 삭제할 snode가 존재하지 않거나, child snode들이 있는 디렉토리 일때는 error를 return 한다. 다른 snode와의 연결을 없앤 후 free()를 통해 snode를 없애준다.

4) snode* search_snode(char* name, snode* search_dir)

찾을 snode의 이름인 name과 찾기 시작할 위치인 search_dir을 parameter로 가진다. Search_dir는 반드시 디렉토리 여야만 하고, search_dir이 레귤러 파일일 때와 찾는 snode가 존재하지 않을 때 error를 return 한다. 현재 snode를 가리키는 cur 포인터를 이용해 search_dir부터 순서대로 name을 비교해 일치하는 snode를 return한다.

5) snode* search_snode_path(const char* path, snode* root)

경로 char* path와 snode들의 tree 구조의 root를 parameter로 가진다. 항상 root부터 검색을 시작하며, 받아온 path에 존재하는 snode를 return한다. 해당하는 snode가 없을 경우 NULL값을 return 한다.

6) char* get_child_name(char* path)

파일 경로 char* path를 parsing하여 해당 path의 가장 마지막에 해당되는 파일 또는 디렉토리(child) 이름을 return한다.

7) char* get_parent_name(char* path)

파일 경로 char* path를 parsing하여, 가장 마지막에 있는 child의 바로 상위 parent 이름을 return한다.

8) char* get_parent_path(char* path)

파일 경로 char* path를 parsing하여, 가장 마지막의 child 이름을 제외한 바로 상위 parent의 경로를 return한다. 이 함수의 결과값을 search_snode_path() 함수의 parameter로 사용하여 새 child를 insert할 위치를 찾을 수 있다.

(3) datalist 와 관련된 함수

1) data* make_data(int ino)

inode number를 parameter로 가진다. 동적 메모리 할당을 통해 data 공간을 할당해 주고 만들어진 data를 return한다.

2) void insert_data(data* newNode)

make_data를 통해 만들어진 data를 linked list에 삽입해 연결한다. Datalist가 비어 있는지 확인하여 비어있으면 datalist.head와 연결하고, 비어있지 않으면 list의 가장 끝인 datalist.tail과 연결한다.

3) void delete_data(data* ndata)

삭제할 data를 parameter로 받아와 삭제한다. datalist와의 연결을 없앤 후 free()를 통해 지워준다.

4) data* search_data(int ino)

inode number를 parameter로 가진다. Current 포인터를 linked list의 head부터 하나씩 이동시키다가 찾는 inode number와 일치한 값을 가지는 data를 찾으면 return 한다.

(4) SSF의 기능과 직접 관련된 함수

1) int ssf_getattr(const char* path, struct stat* stbuf)

경로 char* path에 존재하는 파일의 metadata를 가져와 stat struct인 stbuf에 저장한다. 해당하는 파일이 존재하지 않을 경우 ENOENT 에러를 return한다.

2) int ssf_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi)

경로 char* path에 존재하는 디렉토리에 속한 파일의 정보를 읽어 buf에 저장한다. 해당하는 디렉토리가 존재하지 않을 경우 ENOENT 에러를 return하고, 해당 파일이 디렉토리가 아닐 경우 ENOTDIR 에러를 return 한다.

3) int ssf_mkdir(const char *_path, mode_t mode)

경로 char* _path와 mode를 parameter로 가지며 해당 경로에 새로운 디렉토리를 생성한다. 해당하는 path에 이미 같은 이름의 파일이나 디렉토리가 있으면 생성할 수 없다.

4) int ssf_create(const char* _path, mode_t mode, struct fuse_file_info* fi)

경로 char* path에 지정한 mode로 새로운 파일을 생성한다. 만들려는 파일과 동일한 이름을 갖는 파일 또는 디렉토리가 있는 경우, EEXIST 에러를 return하며 원래 존재하던 파일과 디렉토리를 유지한 채 function call이 종료된다.

5) int ssf_chmod(const char* path, mode_t mode)

경로 char* path와 바꿔줄 mode_t mode를 parameter로 가지며 해당 경로의 파일의 권한을 변경한다. 해당 파일이 존재하지 않을 경우 ENOENT 에러를 return 한다.

6) int ssf_open(const char *_path, struct fuse_file_info *fi)

parameter로 받아온 경로 char* path를 복사한 뒤 해당 경로에 있는 파일의 에러를 체크한다. 해당 경로에 파일이 존재하지 않으면 ENOENT 에러를 return하고, 해당 파일이 디렉토리일 경우 EISDIR 에러를 return 한다.

7) int ssf_release(const char* path, struct fuse_file_info* fi)

파일을 닫는다.

8) int ssf_rmdir(const char* path)

경로 char* path를 parameter로 받아와 해당 경로에 있는 디렉토리를 삭제한다. 삭제할 경로에 디렉토리가 없으면 ENOENT 에러를 return하고, 디렉토리가 비어있지 않을 경우 ENOEMPTY를 return 한다.

9) int ssf_unlink(const char* path)

경로 char* path를 parameter로 받아 해당 경로의 파일을 삭제한다. 삭제할 파일이 없을 경우 ENOENT 에러를 return 한다.

10) int ssf_rename(const char* old, const char* new)

원본 파일의 이름 char* old와 바꿀 이름 char* new를 parameter로 가진다. 이름을 바꿔줄 파일이 존재하지 않는 경우 ENOENT 에러를 return 한다.

11) int ssf_read(const char *path, char* buf, size_t size, off_t offset)

열린 파일로부터 내용을 읽어와 buf에 저장한다. 경로 char* path에 파일이 없는 경우 ENOENT 에러를 return 한다.

12) int ssf_write(const char* path, const char* buf, size_t size, off_t offset, struct fuse_file_info *fi)

열린 파일에 buf에 있는 내용을 읽어와 쓴다. 경로 char* path에 파일이 없는 경우 ENOENT 에러를 return 한다.

4. 실제 작동 모습

(1) File/Directory

1) 파일 생성 touch

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example$ ./ssf mnt
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example$ cd mnt
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ touch file1
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ touch file2
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
file1  file2
```

Figure 8 touch

파일 생성은 명령어 touch를 통해 기능한다. touch명령어는 ssf_getattr(), ssf_create(), ssf_utime()을 호출하며 빈 파일을 생성한다. 생성된 파일을 확인하기 위해 사용한 명령어 ls는 ssf_readdir()를 호출한다. 같은 이름의 파일이나 디렉토리가 존재하는 경우, 별다른 에러 메시지는 출력하지 않고 원본 파일을 유지한 채 종료된다.

2) 디렉토리 생성 mkdir

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ mkdir dir1
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
dir1  file1  file2
```

Figure 9 mkdir

디렉토리 생성은 명령어 mkdir를 통해 기능한다. ssf_mkdir()가 호출되며 빈 디렉토리를 생성한다. 같은 이름의 파일이나 디렉토리가 존재하는 경우, 에러 메시지를 출력하고 종료된다.

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ cd dir1
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1$ touch file3
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1$ ls
file3
```

Figure 10 디렉토리 안에서의 touch

cd 명령어를 통해 디렉토리에 진입하여 touch 명령어를 사용하면 해당 디렉토리 내에 빈 파일이 생성된다.

3) 다중 디렉토리 생성

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ cd dir1
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1$ mkdir dir2
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1$ cd dir2
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1/dir2$ mkdir dir3
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1/dir2$ touch file4
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1/dir2$ cd ..
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1$ cd ..
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ tree
.
├── dir1
│   ├── dir2
│   │   ├── dir3
│   │   └── file4
│   └── file3
├── file1
└── file2

3 directories, 4 files
```

Figure 11 다중 디렉토리 생성

디렉토리 생성과 진입을 반복하면 다중 디렉토리 구조가 만들어진다. 파일 구조를 한 눈

에 보기 위해 리눅스에서 제공하는 tree 확장기능을 사용하였다.

4) 파일 복사 cp

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ cp file1 file5
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
dir1  file1  file2  file5
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ cat file5
hello!
```

Figure 12 cp

cp 명령어를 통해 파일을 복사할 수 있다. cp [복사할 파일] [생성할 파일]의 형식으로 기능하며, 새로운 파일을 생성하고 동시에 원본 파일의 내용이 새로 생성되는 파일에 복사된다.

(2) File/Directory Delete

1) 파일 삭제 rm

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
dir1  file1  file2  file5
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ rm file2
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
dir1  file1  file5
```

Figure 13 rm

파일 삭제는 rm 명령어를 통해 기능한다. rm은 ssf_getattr(), ssf_unlink()를 호출하며 원하는 파일을 삭제한다.

2) 디렉토리 삭제 rmdir

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1/dir2$ ls
dir3  file4
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1/dir2$ rmdir dir3
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt/dir1/dir2$ ls
file4
```

Figure 14 rmdir

디렉토리 삭제는 rmdir 명령어를 통해 기능한다. ssf_rmdir()가 호출되며 빈 디렉토리를 삭제한다. 디렉토리가 비어있지 않을 경우 error가 발생한다.

(3) File Write vi

1) 빈 파일에 쓰기

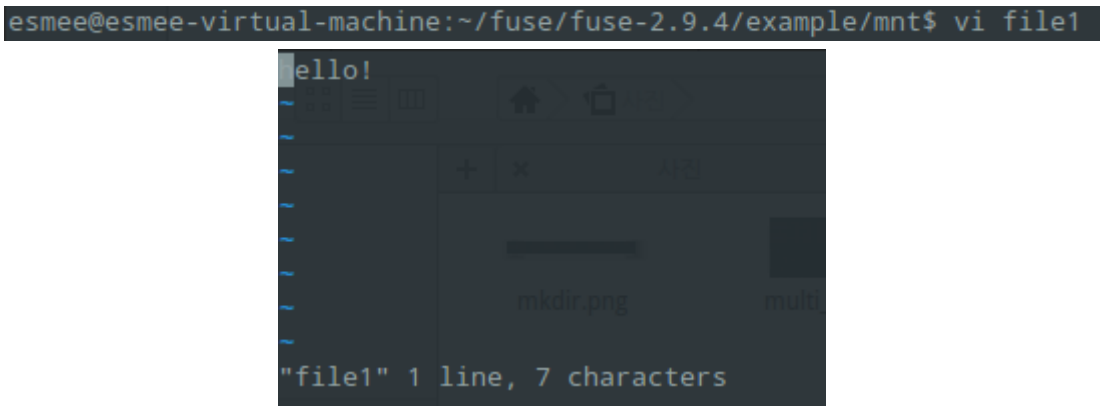

Figure 15 빈 파일에 쓰기 - vi

이미 만들어져 있는 빈 파일에 쓰기는 vi 명령어를 통해 기능한다. vi editor는 ssf_read(), ssf_getattr(), ssf_write()를 호출하며, 파일의 내용을 쓰고 저장할 수 있다.

2) 새로운 파일을 생성하여 쓰기

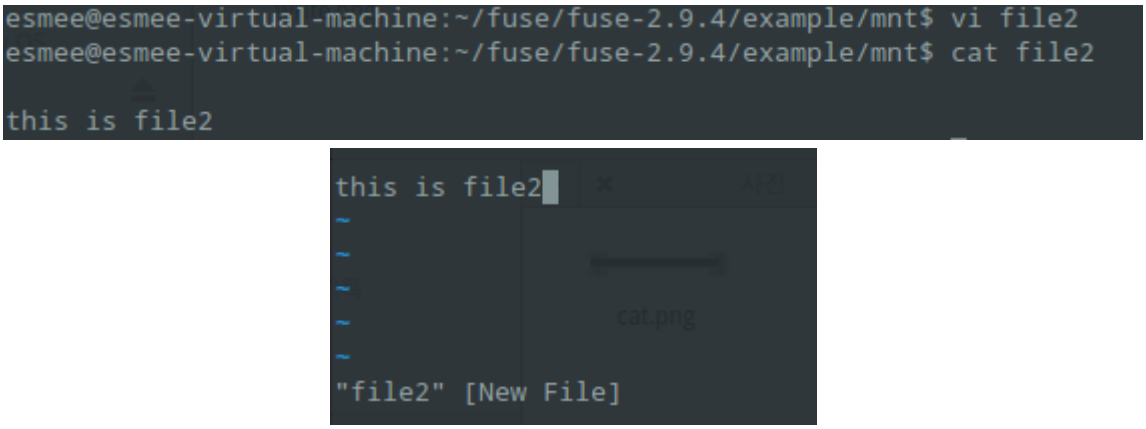

Figure 16 새 파일에 쓰기 - vi

이미 생성된 파일이 아니라 새로운 파일을 생성함과 동시에 내용을 쓸 수도 있다. Touch 명령어가 아니라 vi [생성한 파일] 의 형식으로 파일을 생성하며 바로 vi편집기로 진입해 내용을 입력할 수 있다.

(4) File Read cat

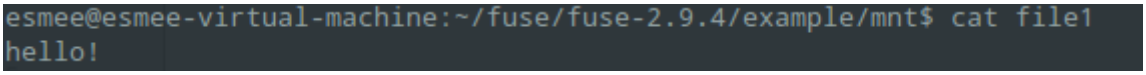

Figure 17 cat

파일은 cat 명령어를 통해 내용을 읽어올 수 있다. cat 명령어는 ssf_open(), ssf_read(), ssf_getattr(), get_release()를 호출하며 파일의 데이터를 화면에 출력한다.

(5) 추가 기능

1) File Rename mv

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
dir1  file1  file2  file5
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ mv file2 file_2
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls
dir1  file1  file5  file_2
```

Figure 18 mv

필수기능 이외에 추가로 구현한 기능이다. mv 명령어를 통해 기능하며 mv [원래 이름] [바꿀 이름]을 입력하면 해당 파일의 이름이 바뀐다.

2) Permission control chmod

```
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls -al
total 4
drwxr-xr-x 26 root  root     0 12월  3 22:57 .
drwxr-xr-x  5 esmee users 4096 12월  3 22:57 ..
-rw-r--rw-  1 root  root  4096 12월  3 23:10 .file2.swp
drwxrwxr-x  4 root  root     0 12월  3 22:58 dir1
-rw-rw-rw-  1 root  root     7 12월  3 22:57 file1
-rw-rw-rw-  1 root  root     7 12월  3 23:03 file5
-rw-rw-rw-  1 root  root    15 12월  3 23:09 file_2
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ chmod 0777 file1
esmee@esmee-virtual-machine:~/fuse/fuse-2.9.4/example/mnt$ ls -al
total 4
drwxr-xr-x 26 root  root     0 12월  3 22:57 .
drwxr-xr-x  5 esmee users 4096 12월  3 22:57 ..
-rw-r--rw-  1 root  root  4096 12월  3 23:10 .file2.swp
drwxrwxr-x  4 root  root     0 12월  3 22:58 dir1
-rwxrwxrwx  1 root  root     7 12월  3 22:57 file1
-rw-rw-rw-  1 root  root     7 12월  3 23:03 file5
-rw-rw-rw-  1 root  root    15 12월  3 23:09 file_2
```

Figure 19 chmod

필수 기능 이외에 추가로 구현한 기능이다. chmod 명령어를 통해 기능하며 파일의 권한을 변경한다. 위에서는 file1의 권한이 변경됨을 확인할 수 있다.

3) Error Check

에러가 발생하는 경우 해당하는 에러를 return한다. 다음은 파일 시스템을 구현하며 확인

한 에러들이다.

- ENOENT : 존재하지 않는 파일에 어떤 작업을 시도할 경우 발생한다.
- ENOTDIR : 디렉토리가 아닌 파일에 대해 디렉토리를 대상으로 하는 작업을 시도할 경우 발생한다.
- EEXIST : 이미 존재하는 파일을 다시 생성하려고 시도할 경우 발생한다.
- EISDIR : 디렉토리에 대해 rm 같은 디렉토리를 대상으로 하지 않는 작업을 시도할 경우 발생한다.
- ENOTEMPTY : 내부에 파일이 존재하는 디렉토리를 삭제하려고 시도할 경우 발생한다.