Project04 wiki

2017030055 윤상현

1. Design& implementation &test

우선 기본적인 login 시스템을 만들기에 앞서 기존의 xv6의 부팅 시스템을 분석하였습니다. Init.c 에서 sh.c 를 실행하여 부팅을 시작하게 되는데, 이 과정에서 sh.c 이전에 login.c 를 먼저 실행하여 로그인을 한 뒤에, sh.c 를 실행하는 것으로 login에 대하여 구현하였습니다.

```
main(void)
{
 // *initial=0;
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  myopen();
  ///printf(1,"aaaaaa\n");
 // *initial=1;
  for(;;){
    printf(1, "login:" );
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("login", argv);//open login instead of sh
      printf(1, "init: exec login failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

위 사진은 init.c의 main문입니다. 위와 같이 sh.c 가 실행되어야할 위치에 login.c 를 exec하도록 구현하였습니다. 이후에 login.c에서 로그인이 인증되면, 그곳에서 sh.c. 를 exec하게 됩니다. 또한 추가된 유저들은 userS라는 유저들의 정보를 모아둔 파일에 아이디와 비밀번호가 저장되어, 로그인 할때마다 해당 파일을 참조하여 로그인 정보를 인증 한 뒤, 실행됩니다.

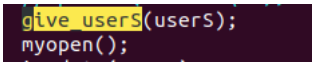
이를 통하여 xv6가 재부팅 되더라도, 새롭게 make가 되지 않는다면 기존에 addUser함수로 추가한 유저정보나, deleteUser을 통해 삭제한 유저정보는 시스템 상에 유지됩니다.

```
struct {
    int using_idx;//current using user's index in priv[i]
    char priv[11][2][17]; // [usernum][id or pwd][string]
    int num; //cur user number.
} user_info;
```

위 사진은 fs.c 에 선언한 user 구조체입니다. Priv는 유저의 개인정보(id, passward) 를 가져오는 배열로, 3차원 배열로 선언해주었고 [유저수] [아이디 or 비밀번호] [char 배열] 의 정보를

표현하기 위해 3차원으로 만들어 주었습니다. Using_idx는 '현재 사용중인 유저' 가 priv 배열상 몇번에 위치해 있는지 알려주는 변수로, get_now() 함수가 이를 사용하여 현재 사용중인 유저 정보를 얻게 됩니다. 이는 나중에 권한비교(현재 사용중인 유저의 권한 check) 에도 사용됩니다. 그리고 num 변수는 현재 priv내에 몇 명의 유저 정보가 저장 되어있는지 계속해서 갱신하는 변수로써, 유저가 add될 시 +=1, 유저가 delete 될 시에는 -=1 이 되어 유저수에 대한 정보를 유지합니다. 이와 같이 구현한 이유는, pirv는 큐, stack 과 같이 delete시 동적으로 크기가 조정되지 않기에 배열의 길이 정보가 유저의 수를 알려 줄 수 없고, add, delete역시 뒤에 쌓거나, 뒤에서 pop을 하는 방식이 아닌, O(n)의 search 통해 빈공간을 찾아 add하거나 지울 유저를 찾아 delete하는 방식이기 때문에 num 변수를 통하여 따로 개수 정보를 저장해 나갔습니다.

Myopen, addUser, deleteUser과 같은 함수들은 sys_ 함수에서 일차적인 과정들을 수행 후 fs.c 에 존재하는 함수로 넘어오게 되는데, 이때 userS에서 유저정보를 로드한 뒤, 넘어오게 됩니다.



위 사진과 같이 sys_ 함수 내에서 give_userS 를 통하여 유저 정보를 로드 시킨 뒤, 함수를 수행함으로써 기존의 유저정보를 통한 작업 수행이 가능해집니다.

1-1) User system

최초 시스템을 구축하는 system call 입니다. 만약 이전에 작업하던 정보가 있다면 불러오고, 아무런 정보가 없다면 root 정보를 생성하는 등 최초의 login system의 전반적인 영역을 담당합니다.

```
if((userS = namei("/userS"))==0){
  userS = create("/userS", T_FILE, 0, 0);
  strncpy(userS->owner, "root", 16);
  userS->permission = MODE_RUSR|MODE_WUSR;
  iupdate(userS);
  iunlockput(userS);
}
```

위 사진은 sys_open 함수의 일부분입니다. Xv6에 구현되어있는 namei함수를 통하여 userS의 정보를 불러옵니다. 만약 정보가 없다면 (최초 부팅 시가 이 경우에 해당합니다.) create을 통하여 file을 만들고 root 정보를 넣어줍니다. Sys_adduser, sys_deleteuser 함수 역시 위와같이 namei를 통하여 기존에 추가된 정보와 대조한 뒤, 동작하는 방식으로 선처리를 해준 뒤, 진행하였습니다. 또한 유저는 자신의 디렉토리를 가지기에, sys_addUser에선 유저 생성 시 해당 유저의 디렉토리를 아래와 같이 생성해줍니다.(해당 디렉토리의 owner는 유저 자신으로 설정해줍니다). 과제명세에 deleteuser는 별도의 처리를 하지 않는다고 명시되어있기 때문에 별도로 deleteuser를 통하여 삭제되는 유저의 디렉토리를 제거하거나 하지는 않습니다.

```
ip = create(username, T_DIR, 0, 0);
ip->permission = MODE_RUSR|MODE_WUSR|MODE_XUSR|MODE_ROTH|MODE_XOTH;
strncpy(ip->owner, username, 16);
iupdate(ip);
iunlockput(ip);
```

fs.c에선, sys_함수에서 받아온 유저정보들을 토대로 들어온 명령들을 수행하게 됩니다.

아래 사진은 매 부팅시 마다 initialize해주는 myopen 함수입니다.

```
void myopen(){

    memset(user_info.priv, 'x', sizeof(user_info.priv));
//  user_info.userS=userS;

    readi(userS, (char*) user_info.priv ,0,374); //userS>> add to priv
    for(int i=0;i<10;i++){//already exist information
        if(user_info.priv[i][0][0]!='x')user_info.num++;

    }
    if(user_info.num==0){ //root!
      strncpy(user_info.priv[0][1], "0000", 16);
      strncpy(user_info.priv[0][0], "root", 16);
      user_info.num++;//create! >> num+1


    }

    update(userS);
//  cprintf("iiii!\n");
//  return 0;
    initial=1;
}
```

위에서 설명했듯이, give_userS함수를 통하여 전역으로 선언된 inode 구조체인 userS에

유저정보가 담긴 정보가 들어오게 되고, 만약 유저정보가 비어있다면 root를 가지고 최초정보를 생성, 비어있지 않다면 기존 유저정보를 load해줍니다. 이후 update함수를 통해 이 변경사항을 반영해주는데, 이 함수는 뒷부분에서 설명하겠습니다.

아래 사진은 addUser 함수입니다.

```
int addUser(char* username,char* password){//find empty space and add user.
    int idx=find_empty();

    if( idx >=10||idx==-1){
        cprintf("error: you cannot add more user\n");
        return -1;
    }
    for(int i=0;i<10;i++){
        if(strncmp(user_info.priv[i][0],username,16)==0){
            cprintf("error: alreay registered\n");
            return -1;
        }

    }
    strncpy(user_info.priv[idx][0], username, 16);
    strncpy(user_info.priv[idx][1], password, 16);
    update(userS);
    user_info.num++;

    return 0;
}
```

Find_empty() 함수는 단순히 priv 배열을 search하여 처음으로 찾아진 빈공간의 index를 리턴해주는 함수입니다. 만약 빈공간이 없으면 -1을 리턴해주게됩니다. Add를 할 수 없는 예외사항들을 체크해준 뒤, priv 배열에 추가할 유저 정보들을 strncpy 함수를 통하여 저장합니다.

이후 마찬가지로 update 함수를 통해 수정된 유저정보 사항들을 반영해줍니다. 이후

User의 수를 1 늘려줍니다.

아래 사진은 deleteUser 함수 입니다.

```
if(strncmp(username,"root",16)==0){
    cprintf("error: you cannot delete root\n");
    return -1;
}
for(int i=0;i<10;i++){
    if(strncmp(username,user_info.priv[i][0],16)==0){//find user to delete
        memset(user_info.priv[i],'x',sizeof(user_info.priv[i]));

        update(userS);
        user_info.num--;
        return 0;
    }
}
cprintf("error: nothing erased\n");
return -1;
```

먼저 root를 지우려고 한다면 , return -1 통해 요청을 거부해줍니다.

그리고 cmp 함수를 통하여 내가 지우려는 유저가 존재하는지 find를 해준 뒤,

Memset을 통하여 해당 유저의 정보들을 전부 'x' 로 초기화 해줍니다. 그리고

Update 함수를 통하여 정보를 반영한 뒤, user의 수를 1 감소시킵니다.

```
$ userdelete_test
[Delete user]
Username: a
Delete user successful!
$
```

위 사진과 같이 기존에 존재하는 유저가 입력으로 들어온 경우 search에 성공하여 delete해줍니다.

아래 사진은 update 함수입니다.

```
//priv[][][] >> 11,2, 17
    int max = ((MAXOPBLOCKS-1) / 2) * 512;

    int p=(char*)(user_info.priv),i,idx=0;

    int ed=374;//11*2*17
    while(idx<ed){
        if(ed-idx>max)
            ed=max;
         i=writei(userS,p+idx,idx,374);

         if(i>0){
             idx+=i;
            }
         else{
            cprintf("update error!!\n");
             break;
         }
    }
    return 1;
```

이 함수는 adduser, deleteuser, myopen 함수의 helper function으로써, 변경된 유저정보를

유저정보를 저장하는 file에 write해줍니다. 이처럼 give_userS함수를 통하여 userS 파일에서 저장 된 유저정보를 load 해오고, fs.c 에서 변경한 유저정보들을 update 함수를 통하여 다시 userS 파일에 write 함으로써, xv6가 재부팅 되더라도 기존의 유저정보들을 유지한채 동작할 수 있게 됩니다.

제가 설명한 것처럼 adduser, deleteuser 를 통하여 변경한 유저정보가 올바르게 상호작용하는지 확인하기 위하여 최초 로그인시 'a' 'b' 유저를 adduser을 통해 생성 한 뒤, deleteuser을 통해 a 유저만 제거한 뒤, 재부팅 뒤에 로그인 시도를 해보았습니다.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
login:User: a
password: a
error: no such username exist
wrong information!
User: b
password: b
$
```

위 사진과 같이, 삭제된 a 유저는 그러한 유저가 없다는 문구가 뜨고, 추가된 b유저는 재부팅 되었음에도 올바르게 정보가 유지되어 로그인이 성공한 모습입니다.

지금까지는 유저정보의 수정에 관련된 fs.c 함수들이었고, 이제 이 정보를 토대로 동작하는 login, logout 함수에 대하여 설명하겠습니다.

```
int login(char* username,char* password){
    for(int i=0;i<10;i++){
        //cprintf("user: %s, psw: %s\n",user_info.priv[i][0],user_info.priv[i][1]);
        if( strncmp(username,user_info.priv[i][0],16)==0 ){
            if(strncmp(password,user_info.priv[i][1],16)==0) {//id, psw match!
                user_info.using_idx=i;
                return 0;
            }
            cprintf("error: wrong password\n");
            return -1;
        }
    }
    cprintf("error: no such username exist\n");
    return -1;

}

int logout(){
    user_info.using_idx=0;
    return 0;
}
```

login에선, login.c로부터 입력된 로그인 정보(유저 정보) 를 받아오고, 해당 유저가 존재하는지 priv 배열을 돌며 serach합니다. 만약 존재한다면, '현재 사용중인 유저' 로 세팅해준뒤, return 0을 통해 login 성공을 알립니다. Logout은 비교적 심플합니다. 현재 사용중인 유저를 0으로 초기화 후, 성공을 알립니다.

아래 사진들은 로그인과 관련된 몇가지 테스트 입니다.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
login:User: root
password: 1234
error: wrong passward
wrong information!
User: root
password: 0000
$
```

Root의 최초 비밀번호는 0000으로 설정되었기 때문에 올바르지 않은 비밀번호 입력시에 login에 실패합니다.

```
$ useradd_test
[Add user]
Username: j
Password: j
error: you cannot add more user
error: addUser
Add user failed!
```

이전에 a~l 의 이름을 가진 9명을 유저를 addUser을 통해 추가하여 root포함 10명의 유저가 등록되어있기 때문에 이후의 add는 fail합니다.

1-2) File mode, permission

다음 파트는 유저모드와 권한에 관한 구현입니다. 운영체제는 본래 각 파일간의 관계를 모두 파악하고 1대1 대응으로 파일간의 권한에 대해 알아야하지만, 이것은 너무 무겁기에

크게 3가지로 file을 분류합니다. 자신을 생성한 owner와 owner를 제외한 other, 그리고 gruup 이 있지만, 이번 과제에선 owner와 other 2가지 분류만 진행하기에 2개의 케이스에 대한 read, write execute 권한을 각 파일과 디렉토리에 부여합니다.

Root는 최초의 파일 시스템에서 모든 파일 및 디렉토리의 owner이고, 주어진 chmod_test 함수에서 77 과 같이 owner, other 에대한 권한을 변경할 수 도 있습니다.

이러한 별도의 동작이 없다면, open 시스템 콜을 통해 만들어지는

file의 경우 최초에 MODE_RUSR|MODE_WUSR | MODE_ROTH 의 권한을 부여받는데, 이는 owner가 read, write할 수 있으며, other은 read만 가능함을 나타냅니다.

그리고 mkdir로 만들어지는 디렉토리의 경우, 최초에 MODE_RUSR | MODE_WUSR

|MODE_XUSR | MODE_ROTH | MODE_XOTH 의 권한을 가지는데, 이는 owner가 read, write, execute 할 수 있고, other는 read와 execute만 가능하다는 권한을 나타냅니다.

저는 이러한 권한을 체크하기 위해 get_modeBit 함수를 구현하였습니다.

```
int get_modeBit(struct inode* ip, int own_mode,int oth_mode){

//return apropriate bit depends on who(owner or others) is doing what(r, w, exe)
    if(initial==0){
        return 1;
    }

    if(strncmp(ip->owner,get_now(),16)==0||strncmp("root",get_now(),16)==0){//if owner, give mode

        if(own_mode&(ip->permission)==0)//extract needed bit, if 0> 0 else that bit is 1
            return 0;
        return 1;

    }

    if((int)(oth_mode&(ip->permission))==0){

        return 0;//if other, give mode  bit for other
    }

    return 1;



}
```

인자로 들어오는 ip는 inode를 통해 받아온 file 정보이고, 뒤에 추가로오는 own_mode, oth_mode 는 각각 ip와 현재 유저의 관계를 분석하여 만약 owner 라면 owner에 대한 권한 bit를, other 이라면 other에 대한 권한 비트를 리턴합니다. 예를 들어 execute를 하기 위한 권한 체크를 하고 싶다면, 인자로 실행할 파일 혹은 디렉토리를 ip 로, MODE_XUSR(owner의 실행)를 own_mode 로, MODE_XOTH(other의 실행) 을 oth_mode 인자로 받고 됩니다.

만약 owner(혹은 root) 라면 ip의 permission을 MODE_XUSR 와 &연산해주어 해당 영역의 비트만 뽑아내는데, 만약 해당 영역의 bit가 0이라면 0, 1이라면 0이 아닌 값이 남게됩니다.

이를 통하여 해당영역의 비트를 찾을 수 있게되고, return해줌으로써 이 함수에서 현재 주어진 상황에서 적절한 권한 체크를 가능하게 해줍니다.

이 함수를 통해 권한 체크를 한 파트는 과제명세에 나온대로 namex, exec,create, sys_open, sys_chdir, sys_unlink 입니다.

```
int bit=get_modeBit(ip,MODE_WUSR,MODE_WOTH);//write ok?
if(bit!=1){
    iunlockput(ip);
    return 0;


}
```

위 사진과 같이, 체크하고 싶은 파트에서 내가 필요한 비트를 extract 해준 뒤, 만약 1이라면 권한이 존재하는 것이므로 그대로 continue하고, 만약 0이라면 종료해줍니다.

아래 사진은 권한을 변경해주는 chmod 함수입니다.

```
if(strncmp(ip->owner, get_now(), 16)!=0&&strncmp("root", get_now(), 16)!=0){
    cprintf("error: only owner and root can do chmod\n");


    iunlockput(ip);

    end_op();
    return -1;
}



if( (strncmp(ip->owner, get_now(), 16)!=0) && (strncmp("root", get_now(), 16)!=0) ){
    iupdate(ip);//only owner and root  can dp chmod!
    iunlockput(ip);
    end_op();
    return -1;
}
ip->permission=mode;
iupdate(ip);
iunlockput(ip);
end_op();
```

Chmod가 가능한 곳은 root 혹은 mod를 바꿀 대상의 owner만 가능하기에 만약 둘 모두에 해당 되지 않는다면, chmod가 fail됩니다. 그리고 permission을 입력으로 들어온 mode로 변경을 해줌으로써 대상의 권한을 변경해주게 됩니다.

아래는 mode, 권한에 관련된 몇가지 테스트 케이스 들입니다.

우선 ls 결과창입니다.

```
$ mkdir subdir1
$ ls
.              drwxr-x root 1 1 512
..             drwxr-x root 1 1 512
README         -rwxr-x root 2 2 2286
cat            -rwxr-x root 2 3 16524
echo           -rwxr-x root 2 4 15380
forktest       -rwxr-x root 2 5 9688
grep           -rwxr-x root 2 6 18744
init           -rwxr-x root 2 7 15988
kill           -rwxr-x root 2 8 15412
ln             -rwxr-x root 2 9 15264
ls             -rwxr-x root 2 10 18860
mkdir          -rwxr-x root 2 11 15508
rm             -rwxr-x root 2 12 15484
sh             -rwxr-x root 2 13 28304
stressfs       -rwxr-x root 2 14 16400
usertests      -rwxr-x root 2 15 67504
wc             -rwxr-x root 2 16 17264
zombie         -rwxr-x root 2 17 15084
login          -rwxr-x root 2 18 16552
useradd_test   -rwxr-x root 2 19 15628
userdelete_tes -rwxr-x root 2 20 15532
chmod_test     -rwxr-x root 2 21 16040
console        grwxrwx  3 22 0
userS          -rw---- root 2 23 374
subdir1        drwxr-x root 1 25 32
```

유저정보가 저장되어있는 userS는 root를 owner로 가지는 최초의 파일시스템에 의해 생성된 파일로, 최초의 권한은 owner인 root만이 read, write 가능합니다. 이 파일을 가지고 권한 테스트를 진행해보았습니다.

```
$ useradd_test
[Add user]
Username: tst
Password: tst
Add user successful!
$ logout
User: tst
password: tst
$ cat userS
cat: cannot open userS
$
```

위와같이 tst 유저를 추가해준뒤 해당 유저로 로그인 후 userS를 cat하면, 해당 파일을 read(open 에서) 할 수 없다는 문구와 함께 cat에 실패합니다. 이는 최초에 userS의 권한 설정이 other가 read 할 수 없게 설정되었기 때문입니다.

```
$ chmod_test 77 a
chmod successful
$ chmod_test 77 userS
chmod successful
$ logout
User: a
password: a
$ cat userS
rootx0000xaxaxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

하지만 위와 같이 other에도 모든 권한을 허용해주면, userS 파일에게 other로 인식되는 a유저도 해당 파일을 읽어들일 수 있게 됩니다. userS 의 내용에 대해 간략이 설명하면

User1xpasswardxUser2xpassward 처럼 'x' (디폴트 값)를 통하여 각 유저간의 구분, 그리고 id와 passward간의 구분을 해주었습니다.

1-3) Ls modification

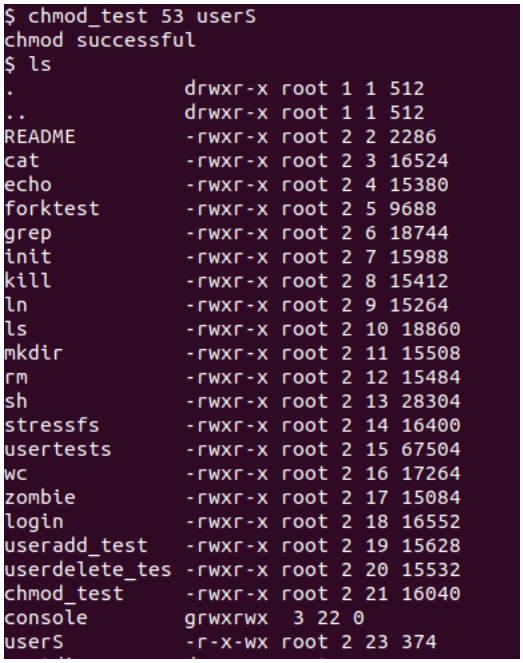
```
//project4
char permission[10] = "drwxrwx";//project4
if(st.type == 1){
  permission[0] = 'd';
}else if(st.type == 2){
  permission[0] = '-';
}else{
  permission[0] = 'g';
}

or(int i = 0; i < 6; i++){
  int idx = 6-i;
  if((st.permission & (1<<i))==0)permission[idx] = '-';
}
```

ls에서 drwxrwx 와 같이 파일인지 directory인지, 그리고 owner과 other에 대한 read write execute 권한을 표시해주도록 ls.c 를 변경해주었습니다. Permission 배열에 기본적으로

drwxrwx의 값을 세팅해주고, 주어진 파일에 따라 값을 변경해줍니다.

만약 디렉토리면 맨 앞의 character을 'd'로, file 이면 '-'로 변경해줍니다. 그외의 케이스는 필요 없지만 이론 수업에서 배운내용대로 남은 케이스는 group을 의미하는 g로 변경해주게 되었습니다. 그리고 해당 파일의 권한 bit 를 체크하여 만약 해당 권한의 비트가 0이라면(없다면) '-'로 해당영역을 변경하여 권한 없음을 표시하였습니다.

Ls가 올바르게 권한을 표시하나 확인을 하기위해, chmod 를 통해 userS의 권한을 변경한 뒤, ls 명령어를 수행해보겠습니다.(원래의 ls 결과는 앞의 chmod 사진을 참고해주세요.)



53 은 owner의 bit 값이 5 즉 101 을 의미하는 것이고, 이는 read, execute권한이 존재한다는 것을 의미합니다. 그리고 뒤의 3은 other의 권한이 011, 즉 write, execute권한이 있다는 것을 의미합니다. 또한 userS 는 파일이기에 맨앞에 '-'로 T_file(파일) 임을 명시하기에

Ls 를 통하여 나온 권한 결과 값이 -r-x-wx 가 되고, 이것이 ls 명령어를 통하여 올바르게 출력됨을 확인 할 수 있습니다.

2) Trouble shooting

최초에 가장 어려움을 겪었던 부분은 바로 정보의 동기화입니다. Xv6의 전체적인 코드가 매우 크기에 cscope를 활용하여도 완벽하게 주어진 코드들의 수행순서를 확인 하는 것이 어려웠고, 제가 의도한 타이밍에 정보가 갱신되게 만드는 것에 어려움이 많이 있었습니다. 그 예 중 하나가 바로 user정보가 load되는 타이밍이었는데요, userS 파일에서 아직 유저정보를 전부 읽어오지 않았음에도 get_modeBit 함수가 수행되어 올바르지 않은 결과가 수행되었습니다. Printf로 디버깅하는 과정에서 우연히 printf가 들어가면 올바른 정보가 들어가는 것을 보게 되었고, 이를 통해 수행 속도의 차이로 인한 문제일 수 도 있겠다는 생각도 하게 되었습니다.

그래서 제가 생각한 방법은 initial 변수를 통하여 user 정보의 load 가 완료되었는지 check 해주는 것이었습니다. initial 변수는 user정보를 담은 userS가 load되어 동기화되었다면 1로 바뀌게 됩니다. 이를 바꿔주는 시점은 myopen 함수가 끝나는 부분으로, 이를 통하여 load가 완료 되었을 때(올바른 권한정보가 들어온 시점부터)만 권한 체크를 통한 동작 제한을 수행하는 것으로 문제를 해결하였습니다. 또한 inode관련 함수들을 사용하는 것에도 어려움이 있었습니다. 다행이 과제 명세에서 어느정도의 설명은 적혀 있어 어느정도 해결할 수 있었지만, ilock을 걸어놓지 않는 다거나, lock을 걸고 푸는 것을 잊어버리는 등의 실수들은 매우 크게 반영되어 디버깅시에 오류를 찾는것에서 많은 어려움이 있었습니다. 이 경우, cscope를 통해 경로를 타고 가서 ilock에 도달함으로써 제가 실수한 부분들을 찾아낼 수 있었습니다. 또한 기존의 과제들과 달리, 명확한 테스트케이스가 주어지지 않았기에, 어떻게 제가 구현한 코드가 올바르게 동작할 수 있는지 테스트하는 것 역시 어려움이 있었습니다. 큰 틀에서 결국

권한의 테스트가 가장 어려웠고, 이를 수행하는 것은 결국 read, write, execute 의 수행으로 나눠지기에 어떤 call 들이 read, write execute를 수행하는지 cscope를 통하여 찾고 그 중 제 관점에 직관적인 것들을 통하여 test 함으로써 어느정도 제가 구현한 코드를 체크해볼 수 있었습니다.