Project 3 : Implementing User System in xv6

2018008395
박정호

1. User Account

A. Implementation

i. Loading User Information

기존의 xv6는 부팅 후 init 프로세스가 실행이 되면 console을 열고 그 이후에 sh.c를 실행한다. 여기서 난 console을 연 직후 loaduserinfo()라는 system call을 호출해서, 유저 정보를 저장한 파일을 열었다. 하지만, 이렇게 연 파일은 커널 내부에서만 쓰이기 때문에 open system call처럼 fd layer에서 관리하지 않고, inode layer에서 관리했다. 어차피 유저 id와 password는 각각 최대 15글자이고, 유저 수는 10을 넘기지 않는다는 게 보장이 되므로, 딱 320바이트 (유저 10명 * (id 16바이트 + password 16 바이트)) 만 I/O를 하는 것으로도 충분히 관리가 가능했다. 읽어온 데이터는 모두 usertable이라는 자료구조의 info라는 배열에 저장했다. usertable의 구조는 다음과 같다.

```
struct {
  char info[10][2][16]; //array for username and password
  int now; //now log in-ed user
  int cnt;
  struct inode* _userlist;
}usertable;
```

ii. Add User and Delete User

useradd()와 userdel은 비교적 간단하게 구현했다.

위의 usertable.cnt가 현재 존재하는 user 계정의 개수인데, 이 값을 가지고 현재 유저가 최대인지 아닌지를 체크했고, 대상 user가 있는지는 단순히 전체 usertable을 순회하면서 확인했다. useradd()의 경우, 아직 해당 user가 없다면 빈 칸을 찾아서 해당 칸에 user 정보를 추가했고, userdel()의 경우 해당 user가 있다면 그 칸의 정보를 모두 0으로 채워서 지웠다.

useradd()의 경우 username과 같은 이름의 디렉토리를 만들어야 했는데 이는 sys_mkdir의 구현을 그대로 가져왔다. 대신 mkdir 이전에 namei로 해당 이름의 파일이 존재하는지 먼저 체크했다.

iii. Login and Logout

기존의 init 프로세스는 sh.c를 직접 실행했지만, 이번에는 login 기능을 위해서 umanager라는 프로그램을 대신 실행하도록 했다. 이 프로그램은 단순히 무한루프를 돌면서 username과 password를 입력받고, 입력받은 값으로 login system call을 호출한다.

login system call은 입력받은 값이 usertable.info에 있다면, 그 값으로 usertable.now를 바꾸고, 0을 return한다. 만약 해당 정보가 usertable에 없다면 바로 -1를 return한다. umanager에서는 이 return값을 가지고 login의 성공/실패 여부를 알려준다. 성공한다면 바로 sh가 실행된다.

logout은 sh에서 처리하는데 logout이라는 문자열을 입력받을 경우, logout system call을 호출하고, 이후에 exit()한다. 여기서 logout system call은 usertable.now를 0으로 맞추고(0은 root 계정이다. 사실 -1로 하려 했지만, 혹시라도 발생하는 예외를 막기 위해 umanager도 root가 실행한다는 가정으로 진행했다. 실제로 umanager에서는 login 외의 시스템 콜이 전혀 호출되지 않기 때문에 딱히 문제는 없다.) 바로 return한다.

usertable.now의 존재 이유는 이후 file mode에서 현재 유저가 누구인지 파악하기 위해서이다. 실제로 getnowuser라는 함수는 이 usertable.now의 값을 가지고 현재 user의 username을 return한다.

B. Test

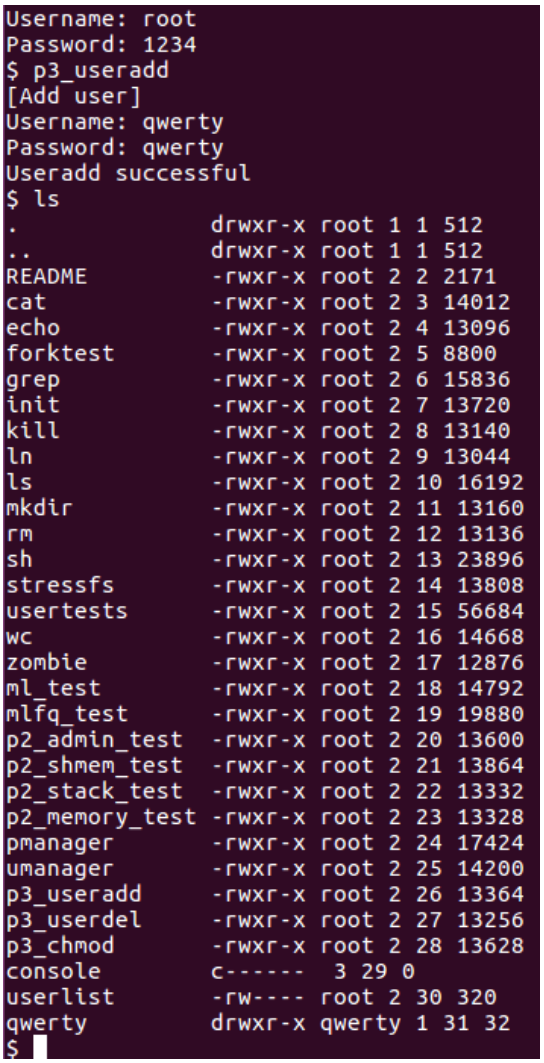
i. Login / Logout

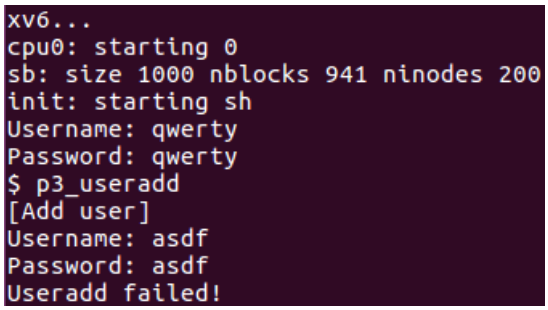
```
pch@pch-VirtualBox:~/os_practice/xv6-public$ qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512
WARNING: Image format was not specified for 'fs.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'xv6.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Username: qwerty
Password: qwerty
Wrong Login Information
Username: root
Password: 1234
main-loop: WARNING: I/O thread spun for 1000 iterations
$ logout
Username:
```

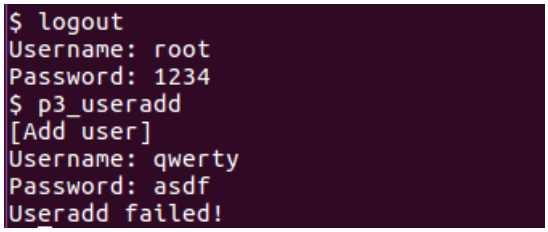
테스트를 위해 빌드를 새로 하고 진행했음을 밝힌다.

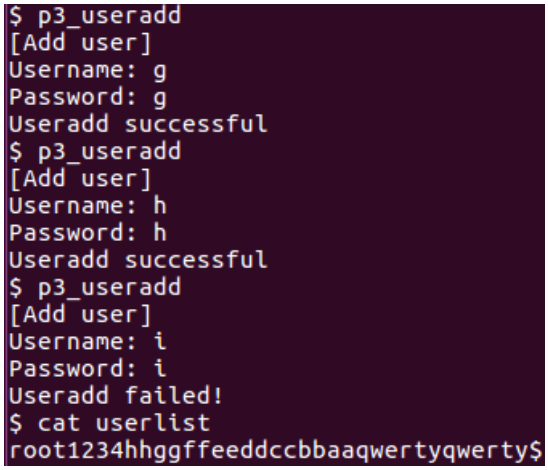
즉, 지금 userlist 파일은 비어있다. 여기서 qwerty와 같은 username은 login이 거부되고, root만이 login이 되는 것을 확인할 수 있다. 또한 sh에서 logout을 입력할 경우 다시 umanager의 login 단계로 넘어오는 것을 볼 수 있다.

ii. p3_useradd









위의 테스트에서 root 로그인 상태에서 시작했음을 밝힌다.

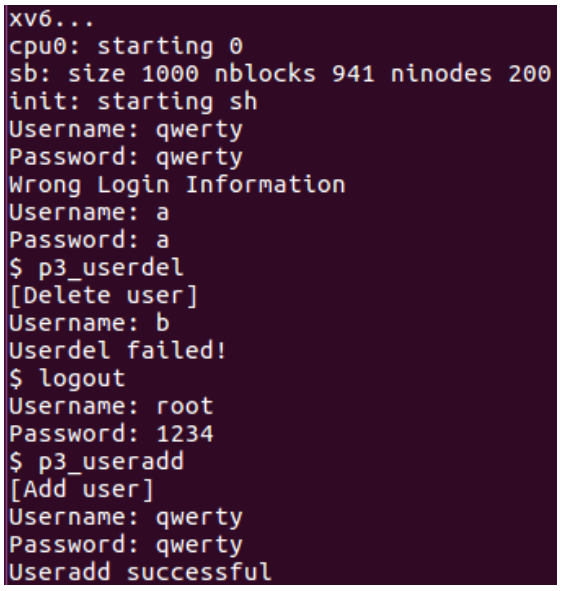
왼쪽 스크린샷에서 확인할 수 있듯이, qwerty라는 유저를 더하면 qwerty라는 디렉토리가 추가되는 것을 볼 수 있다.

오른쪽 첫번째 스크린샷은 재부팅 후의 테스트인데, qwerty 유저로 로그인이 되는 것을 확인할 수 있고, 여기서 p3_useradd를 실행할 경우 root 유저가 아니기 때문에 실패하는 것을 확인할 수 있다.

또한 오른쪽 두번째 스크린샷에서 확인 가능하듯이, 이미 존재하는 username으로 유저를 추가하려고 하면 p3_useradd가 실패하는 것을 볼 수 있다.

오른쪽 세번째 스크린샷은 현재 root와 qwerty라는 유저가 있는 상황에서 a부터 i까지 유저 추가를 시도하는 테스트인데, h까지는 10명의 유저라서 추가가 되지만, i는 11번째 유저라 추가가 거부되는 것을 확인할 수 있다. 또한 userlist를 출력해보면 root와 qwerty, 그리고 a부터 h까지의 유저 정보가 저장되어 있음을 확인할 수 있다.

iii. p3_userdel



위의 테스트 이후 재부팅 후 실행했음을 밝힌다.

왼쪽 스크린샷은 root 계정에서 다양한 상황에서 p3_userdel을 실행하는 테스트인데, 우선 qwerty를 지우게 될 경우 성공적으로 userlist에서 qwerty 유저의 정보가 지워지는 것을 볼 수 있다. 그렇지만, root를 지우려 시도하거나, 이미 존재하지 않는 유저인 qwerty를 지우려 하면 p3_userdel이 실패하는 것을 볼 수 있다.

오른쪽 스크린샷은 다시 xv6을 재부팅하고 p3_userdel을 실행해보는 테스트인데, 우선 이미 지워진 qwerty 계정으로는 이제 로그인이 되지 않는 것을 볼 수 있다. 또한 root가 아닌 a 유저가 p3_userdel을 수행할 경우 실패하는 것을 볼 수 있다. 그리고 다시 root로 로그인해서 qwerty라는 유저를 추가할 경우 다시 추가가 성공적으로 되는 것을 볼 수 있다.

C. Trouble Shooting

i. fd가 아닌 inode를 통한 파일 I/O

처음에 파일 쓰기가 제대로 안되어서 조금 애를 먹었었다.

이는 inode 전용함수인 ilock(), iupdate(), iget(), iput() 등의 이해가 부족해서 생긴 일이었는데, 이를 사용하기 위한 규칙을 제대로 지켜서 사용하니, 제대로 파일이 추가되었다.

ii. 유저에 대한 이해

유저를 처음에는 마치 프로세스처럼 여러 명이 존재할 수 있겠다고 생각했다. 하지만 이 또한 과제 이해가 부족했던 것으로, umanager에서 한번 sh를 실행하면 그 프로세스가 끝날때까지 wait하므로, 사실상 유저는 동시에 한명만 로그인이 가능하다는 것을 이해했고, 따라서 proc 구조체에 user 정보를 추가하는 것이 아닌, 커널의 전역변수로 usertable이라는 자료구조를 두어서 아예 별개로 다루도록 했다.

2. File Mode

A. Implementation

i. inode와 dinode 구조체의 변경

이번 과제의 file mode는 결국 파일 소유자 여부에 따라서 권한이 달라진다. 따라서 파일마다 그 소유자에 대한 정보와 파일의 권한 정보가 추가되어야 했다. 따라서 소유자의 이름과, 권한 정보를 dinode와 inode에 추가했고, 이를 처리하는 부분을 iupdate와 ilock에 추가했다. 변경된 dinode의 구조는 다음과 같다.

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;           // Size of file (bytes)
  uint permission;
  char owner[16];       //owner of this inode
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

여기서 추가된 바이트는 총 20바이트, 따라서 dinode의 크기가 512의 약수가 아니게 된다. 이렇게 될 경우 mkfs.c의 assert를 통과하지 못할 것이기 때문에 NDIRECT의 값을 7로 조정했다. 이렇게 할 경우 총 dinode의 크기가 64바이트가 되어 assert를 통과할 수 있게 된다. 물론 추가된 정보는 inode에도 똑같이 추가했다.

ii. mkfs.c에서의 구현 변경

최초의 파일 시스템 구성에도 이 file mode 정보를 더해야 했다. 이는 ialloc()에서 inode를 할당할 때 permission과 owner를 미리 정해서 추가하는 것으로 해결했다. Permission은 61, 즉 other write만 제한하고, owner는 "root"로 설정해주었다.

iii. File Mode설정

사실 기본 사항은 과제 명세 파일에서 "Guideline" 부분에 설명된 대로 구현했다. namex(), sys_open(), sys_unlink(), sys_mkdir(), create(), sys_chdir()에서 권한을 체크하도록 구현했다. 여기서는 그 권한 체크를 어떻게 하는지만 설명하겠다.

chkimode()라는 함수로 그 권한 체크를 진행했는데, 이 함수는 ip라는 inode 포인터 하나와 mode라는 int형 변수 하나를 받는다. 이 함수는 다음과 같이 동작하는데, 우선 userlist가 load되지 않은 상태라면(이는 전역변수로 userloaded라는 int형 변수를 두어서, userlist를 load할 때 1로 바꾸는 식으로 구현했다.), 부팅 중으로 판단하고 무조건 1을 return한다. 즉 무조건 권한을 열어주는 것이다. 만약 이미 userlist가 load되었다면 ip의 owner와 현재 username을 비교해서 같다면 mode에 8을 곱한다.(owner와 other의 권한은 딱 8배 차이가 난다.) 이후 ip의 permission과 mode를 & 연산해서, 해당 권한 부분만 구한 다음 return했다. 권한이 있었다면 0 이외의 값일 것이고, 권한이 없다면 0일 것이다.

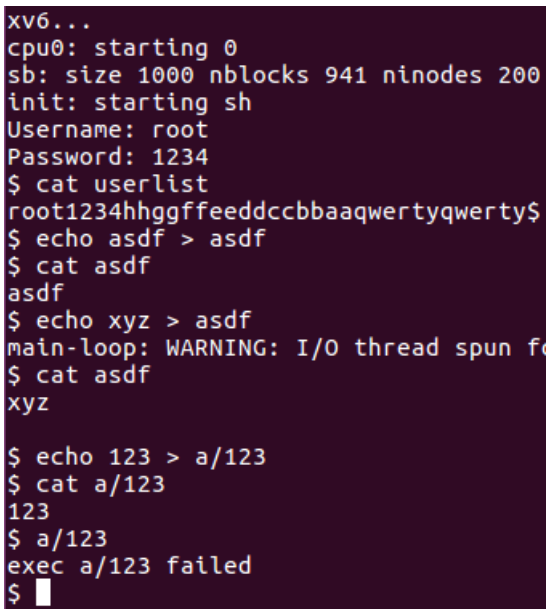
iv. Change File Mode

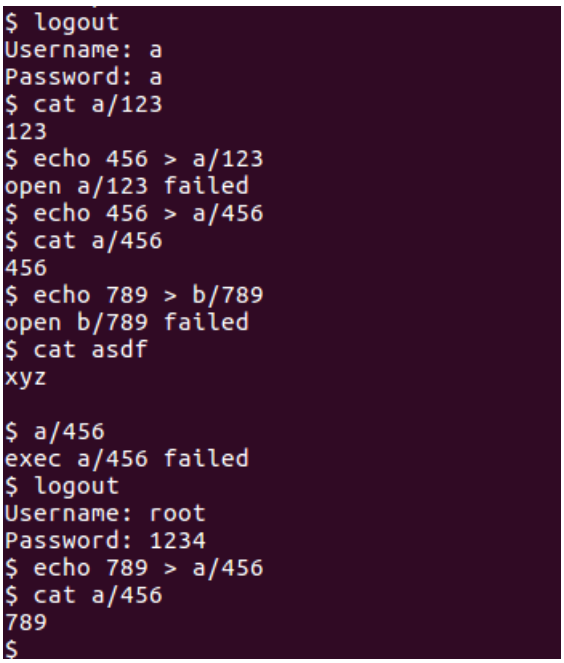
chmod()라는 시스템 콜의 구현이었는데, 내부 함수 없이 sys_chmod()만 구현해서 이 기능을 만들었다. 구현 자체는 굉장히 간단한데, 위에서 chkimode와 비슷하게, ip의 owner와 현재 username을 비교해서 같다면 ip의 permission을 변경하고, 다르다면 바로 -1을 return하도록 했다.

여기서 chkimode는 사용하지 않았는데, chkimode는 owner 여부만 따지지 않고, 그 세부 권한까지 고려하기 때문에 이 시스템 콜에서는 사용하지 않았다.

B. Test

i. File Mode의 적용





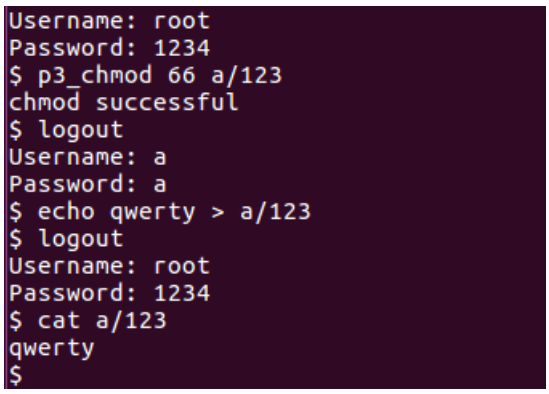
우선 앞의 테스트가 수행된 상태 그대로에서 진행함을 밝힌다.

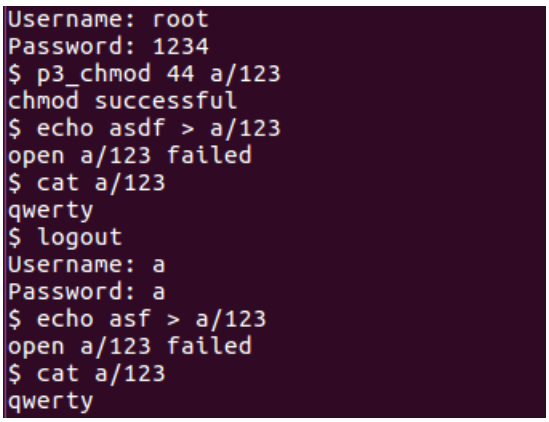
root 계정으로 로그인한 후, asdf라는 파일을 root 디렉토리에 만들면 당연히 소유자가 root이기 때문에 허용이 되는 것을 볼 수 있으며 여기서 내용을 덮어쓰는 것 또한 asdf의 소유자가 root이기 때문에 허용이 되는 것을 볼 수 있다.

또한 root 계정은 다른 유저의 파일과 디렉토리에 대해서도 소유자와 같은 권한을 갖기 때문에 a가 소유하는 디렉토리인 a에도 123이라는 파일을 추가할 수 있음을 볼 수 있다. 하지만 open이 만드는 파일은 기본적으로 execute 권한이 제한되어 있기에 실행하는 것은 실패하는 것을 볼 수 있다.

이제 root는 로그아웃하고, a로 로그인해서 a/123을 읽으면 이는 허용이 된다. 하지만, 이 파일에 덮어쓰기하는 것은 불가능한 것을 볼 수 있다. 파일의 소유자가 a가 아니기 때문이다. 또한 root가 아니기 때문에 타 유저의 디렉터리에 파일을 추가하는 것도 불가능하다. 그러나 root는 a가 생성한 파일에도 덮어쓰기가 허용되고 있음을 볼 수 있다.

ii. chmod system call - File 접근 권한

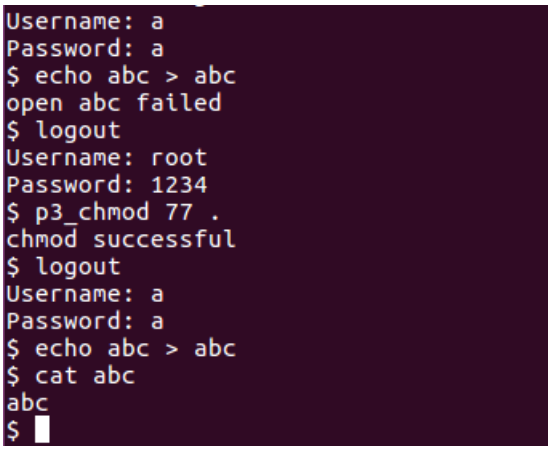




앞의 File Mode 테스트에서, a/123은 root 소유의 파일이기 때문에 유저 a는 읽을 수만 있고 수정할 수는 없었다. 이제 이 파일의 other 권한을 MODE_WOTH까지 주게 되면, echo로 파일을 쓸 수 있는 것을 확인할 수 있다.

또한 이후에 a/123의 권한을 읽을 수만 있도록 바꿔두면, echo로 인해 a/123의 소유자가 된 a도, 모든 파일의 소유자인 root도 echo로 파일 내용을 덮어쓰지 못하고, 오직 읽을 수만 있는 것을 볼 수 있다.

iii. chmod system call - Directory 접근 권한



앞의 File Mode 테스트에서도 봤지만, root 이외의 유저는 원래 root 디렉토리에 파일을 추가할 수 있는 권한이 없었다. MODE_WOTH의 기본값이 디렉토리의 경우 0이기 때문이었는데, root 디렉토리의 권한을 root 유저가 최대 허용치인 77로 맞추게 되면, 일반 유저인 a도 root 디렉토리에 새로운 파일이 추가되는 것을 볼 수 있다.

C. Trouble Shooting

파일의 권한을 지정하고 그 권한에 따라 접근을 제한할 경우, 유저 정보가 만들어지기 전인 부팅 단계에서는 상당히 큰 문제가 생길 수 있었다. 실제로 필자의 경우, 부팅이 끝나지 않거나, init exit이 발생하는 등 여러가지 난관을 만났다.

따라서 난 유저 정보가 만들어지기 전의 접근 권한을 모두 열어두는 것으로 이를 해결했다. 또한, T_DEV의 경우도 모든 권한을 열어두었다.

3. Modification of ls Program

A. Implementation

ls는 결국 command line argument로 받은 경로를 가지고 그 파일의 정보를 보여주는 프로그램이다. 여기서 인자로 넘긴 경로가 디렉토리라면 그 디렉토리 내의 directory entry 들의 정보를 모두 보여주는데, 이번 과제에서 수정해야 할 부분은 이 정보를 콘솔에 나타내는 print문과, 그 정보를 가져오는 함수인 fstat()이었다.

사실 콘솔에 나타내는 것은 어렵지 않다. 각 파일의 permission 값과 type 값을 가지고 문자열을 파싱하면 되는 문제인데, 간단하게 코드로 보여주자면 다음과 같다.

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

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

그럼 사실상 구현에 있어서 중요한 건 fstat()이다. fstat()은 매크로로 sys_fstat()으로 대체되고, 이 함수는 filestat()을 호출한다. 그리고 결국 filestat()에서 stati()를 호출해서 inode 내의 정보를 struct stat에 옮기는데, 여기서 permission과 owner값까지 옮기도록 설정했다. 물론 struct stat에 permission과 owner라는 field를 추가해야 했다. stat의 구조는 다음과 같다.

```
struct stat {
  short type;  // Type of file
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
  uint permission;
  char owner[16];
};
```

B. Test

i. Before chmod system call

우선 이전 테스트가 진행된 상태 그대로 진행함을 밝힌다.

```
Username: root
Password: 1234
$ ls
.               drwxrwx root 1 1 720
..              drwxrwx root 1 1 720
README          -rwxr-x root 2 2 2171
cat             -rwxr-x root 2 3 14012
echo            -rwxr-x root 2 4 13096
forktest        -rwxr-x root 2 5 8800
grep            -rwxr-x root 2 6 15836
init            -rwxr-x root 2 7 13720
kill            -rwxr-x root 2 8 13140
ln              -rwxr-x root 2 9 13044
ls              -rwxr-x root 2 10 16192
mkdir           -rwxr-x root 2 11 13160
rm              -rwxr-x root 2 12 13136
sh              -rwxr-x root 2 13 23896
stressfs        -rwxr-x root 2 14 13808
usertests       -rwxr-x root 2 15 56684
wc              -rwxr-x root 2 16 14668
zombie          -rwxr-x root 2 17 12876
ml_test         -rwxr-x root 2 18 14792
mlfq_test       -rwxr-x root 2 19 19880
p2_admin_test   -rwxr-x root 2 20 13600
p2_shmem_test   -rwxr-x root 2 21 13864
p2_stack_test   -rwxr-x root 2 22 13332
p2_memory_test  -rwxr-x root 2 23 13328
pmanager        -rwxr-x root 2 24 17424
umanager        -rwxr-x root 2 25 14200
p3_useradd      -rwxr-x root 2 26 13364
p3_userdel      -rwxr-x root 2 27 13256
p3_chmod        -rwxr-x root 2 28 13628
console         c------  3 29 0
userlist        -rw---- root 2 30 320
qwerty          drwxr-x qwerty 1 31 32
a               drwxr-x a 1 32 64
b               drwxr-x b 1 33 32
c               drwxr-x c 1 34 32
d               drwxr-x d 1 35 32
e               drwxr-x e 1 36 32
f               drwxr-x f 1 37 32
g               drwxr-x g 1 38 32
h               drwxr-x h 1 39 32
i               drwxr-x i 1 40 32
asdf            -rw-r-- root 2 41 5
xyz             -rw-r-- a 2 44 4
abc             -rw-r-- a 2 45 4
```

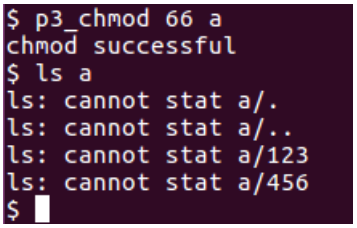
왼쪽의 스크린샷은 root 디렉토리의 모습인데, 앞의 테스트에서 전체 권한을 열어둔 root 디렉토리 이외의 모든 파일의 권한이 명세 상의 기본값으로 초기화된 것을 볼 수 있다. 또한, useradd에서 추가되는 디렉토리와 echo로 만들어진 파일들도 역시 기본값으로 권한이 설정되어 있음을 볼 수 있다.

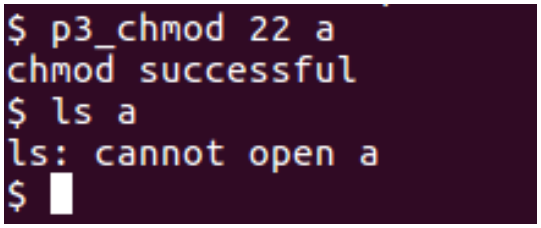
아래의 스크린 샷은 a 유저가 만들어질 때 같이 생성된 디렉토리인 a를 가지고 ls를 실행한 모습이다. 위의 p3_chmod로 인한 권한 변경 상황까지 잘 적용된 모습을 볼 수 있다.

a 디렉토리의 MODE_XUSR가 허용되어 있기 때문에 현재는 ls로 디렉토리 내의 파일 정보를 확인할 수 있음을 알 수 있다.

```
$ ls a
.               drwxr-x a 1 32 64
..              drwxrwx root 1 1 720
123             -r--r-- a 2 42 7
456             -rw-r-- root 2 43 4
```

ii. After chmod system call





이제 a 디렉토리의 파일 모드를 변경하고 ls를 실행해보겠다.

우선 execute 권한을 제거해서 유저가 디렉토리 내의 파일 정보를 읽을 수 없도록 하면, 왼쪽 스크린샷처럼 cannot stat ~~라는 메시지가 나온다. 그 다음 read 권한까지 제거하면 유저가 아예 디렉토리를 열 수 없게 되므로, cannot open ~~라는 메시지를 받게 된다. 이는 기존의 file mode들이 제대로 디렉토리에서도 동작하고 있음을 확인시켜준다.