Project4

2018008877 이상원

Project4 Implementing User System in xv6

xv6의 file system에는 user의 개념이 존재하지 않는다. 따라서 linux에서의 root 계정을 사용하는 것처럼 어떠한 파일이든 읽고 쓰고 실행하는 것이 가능하다. 이번에는 xv6에 user system을 추가하여 login과 logout 을 구현하고 file의 권한에 따라 읽고 쓰고 실행하는 것을 제한할 수 있도록 할 것이다. 또한 새로운 user를 추가하고 기존 user를 삭제하는 system call과 file의 권한을 owner 혹은 root user가 바꿀 수 있도록 chmod system call을 구현할 것이다.

Design & Implementation

User Account

user의 정보를 저장하고 관리하기 위해서 *account.c* 라는 파일을 새로 만들었다. 그리고, user account information 을 관리하기 위하여 utable 이라는 자료구조를 만들었다.

```
// account.c
struct {
  int current_user;
  uint cnt;
  char user[10][2][MAXUSERNAME];
} utable;
```

`utable.user` 는 총 10개의 user account의 username과 password를 저장하는 array 이다. 이때, `utable.user[utable.current_user]` 가 현재 login 된 user이다. `utable.current_user` 가 -1 이라면 로그인되지 않았다는 뜻인데, 이것이 필요한 이유는 부팅을 하고 최초 user process를 실행할 때까지 exec이나 open 등의 system call을 사용해야 하는데 내부에서 권한을 확인해야 하기 때문이다. 마지막으로 `utable.cnt`은 총 몇 개의 계정이 있는 지 나타낸다.

login/logout

init user table

먼저, login을 구현하기 위해서는 최초의 user process가 만들어 진 이후 login을 담당하는 program을 실행시켜 주어야 한다. 기존에 xv6에서는 init process가 실행이 되면 console을 열거나 생성하고, shell 을 실행한다. 하지만, 이제는 *main.c* 에서 `userinit()` 이 실행되기 전 `initUtable()` 함수를 호출하여 `utable` 을 초기화 해 주어야 한다.

```
// account.c
int
initUtable(void)
{
  int i;
  utable.current_user = -1;
  for(i = 0; i < 10; i++){
    memset(utable.user[i], 0, MAXUSERNAME);
  }
  return 0;
}
```

이후 init.c 의 main 함수에서 console을 열고 `setuser` system call을 호출한다.

set user table

- `sys_setuser(void)` 의 동작 과정은 다음과 같다.
  1. “/account” 경로의 `inode` 주소를 namei를 통해 가져온다. 이미 존재한다면 4번으로 이동한다.
  2. create를 통해 “/account” 경로에 file을 만든다.
  3. owner를 root로 설장하고, permission은 `MODE_RUSR | MODE_WUSR` 로 설정한다. (owner 와 permission 에 관해서는 후술할 것임.)
  4. `setuser()` 를 호출한다.
- `setuser(struct inode* account)` 의 동작 과정은 다음과 같다.
  1. `readi` 를 통해 `inode` 로부터 `utable.user` 를 읽어온다.
  2. 만약 아무 내용도 존재하지 않는다면 (최초 상태라면), root 계정을 만들어 주고 `writei` 를 통해 file에 반영한다.

setuser system call 이 정상적으로 마치면, “/account” file의 내용이 `utable` 에 반영된다. 이후 *init.c* 의 for문 내부에서 `exec(”sh”, argv)` 를 `exec(”verify”, argv)` 로 바꾸어 준다. 즉, shell이 아니라 verify를 위한 program을 실행시켜준다.

verify program

verify.c 에서는 user로부터 username 과 password를 입력받아서 `verify` system call을 호출하여 login 한다.

```
int
verify(char* username, char* password)
{
  uint i;
  for(i = 0; i < 10; i++)
  {
    if (!strncmp(utable.user[i][0], username, MAXUSERNAME) &&
        !strncmp(utable.user[i][1], password, MAXPASSWORD)){
      break;
    }
  }
  if(i >= 10)
    return 1;
  utable.current_user = i;
  return 0;
}
```

Run shell program

정상적으로 login 을 마치면, `exec("sh", argv)` 를 통해 shell을 실행한다. 그리고 shell program 이 종료될 때까지 wait 한다.

Logout

shell 의 내장 명령어를 추가하여 logout을 구현하였다.

```
// sh.c
for(i = 0; i < 6; i++){
  if(buf[i] != logoutChar[i])
    break;
}
if(i>=6){
  logout();
  break;
}
```

내부에서 logout system call을 호출하는데, 단순히 `utable.current_user` 을 -1로 만드는 작업을 한다. 이후 shell program을 종료하고, verify program 은 다시 loop를 돈다.

Add or Delete Account

sys_addUser

동작 순서는 다음과 같다.

1. 현재 user가 root 계정인지 확인한다.
2. "/account" 의 inode 주소를 `namei` 를 통해 가져온다.

3. 이후 `addUser()` 를 호출한다.
4. `iupdate` 를 통해 수정사항을 반영한다.
5. 만약 새로 추가 된 username 의 directory가 존재하지 않으면 `create()` 를 통해 새로 만들어준다. 이때, owner는 해당 user이고, permission은 MODE_WOTH를 제외한 모두를 갖는다.

addUser

동작 순서는 다음과 같다.

1. 만약 현재 “/account” 에 10명의 user 정보가 존재한다면, 더 이상 생성할 수 없다.
2. 만약 중복된 user가 존재한다면 생성할 수 없다.
3. 1,2 에 해당하지 않는다면, utable에서 빈 공간을 확인한 후 정보를 채워 넣는다.
4. 이후 `writei`를 통해 반영한다.

sys_deleteUser

동작 순서는 다음과 같다.

1. 현재 user가 root 계정인지 확인한다.
2. “/account” 의 inode 주소를 `namei` 를 통해 가져온다.
3. 이후 `deleteUser()` 를 호출한다.
4. `iupdate` 를 통해 수정사항을 반영한다.

deleteUser

동작 순서는 다음과 같다.

1. 해당하는 username 의 user가 존재하는지 확인한다.
2. 이후 `utable.user[해당 index]` 의 username 과 password 부분을 초기화 해준다.
3. `writei` 를 통해 반영해 준다.

File Mode

가장 먼저, (d)inode 의 구조체를 수정하였다. 또한 dinode 구조체의 크기를 512 의 약수로 맞춰주기 위해서 `NDIRECT`를 7로 바꾸어 주었다.

```
struct dinode {
  ...
  uint permission;
```

```
struct inode {
  ...
  uint permission;
```

```
  char owner[16];
};
```

```
  char owner[16];
};
```

그리고, *fs.h* 에 mode 에 관한 bit를 정의해 주었다.

```
#define MODE_RUSR 32 // owner read
#define MODE_WUSR 16 // owner write
#define MODE_XUSR 8 // owner execute
#define MODE_ROTH 4 // others read
#define MODE_WOTH 2 // others write
#define MODE_XOTH 1 // others execute
```

또한, 최초 file system을 구성할 때, 이 program이 사용되는데, 여기서도 file mode와 owner를 설정해 주어야한다. filemode는 -rwxr-x 로 하였고 owner은 root 계정으로 설정해 주었다.

```
uint
ialloc(ushort type)
{
  ...
  din.permission = xint(MODE_RUSR|MODE_WUSR|MODE_XUSR
               |MODE_ROTH|MODE_XOTH);
  strncpy(din.owner, "root", 16);
  ...
}
```

Check permission

```
int
getPermission(struct inode* ip, uint accessMode)
{
  int current_user;
  uint permission, othersPermission, myPermission;
  char *currentUserName;

  if(ip->type == T_DEV)
    return 1;
  if((current_user = getCurrentUser()) < 0)
    return 1;

  permission = ip->permission;
  myPermission = (permission >> 3) & 0b111;
  othersPermission = (permission) & 0b111;
  currentUserName = getUserName(current_user);

  int isMine =!strncmp(ip->owner, currentUserName, MAXUSERNAME)
           || !strncmp("root", currentUserName, MAXUSERNAME);
```

```
    return accessMode & ((isMine) ? myPermission
                         : othersPermission);
}
```

`getPermission()` 이라는 함수를 만들어서 인자로 들어온 `inode` 에 대해 특정 accessMode로 접근할 수 있는 지 확인할 수 있도록 하였다. 간단하게 과정을 설명하자면, 현재의 user가 root 계정이거나 `inode` 의 owner 라면 MODE_?USR 에 해당하는 bit를 확인해 주어야 하고, root 계정도 아니고 owner도 아니라면 MODE_?OTH 에 해당하는 bit를 확인해 주어야 한다. 따라서 `myPermission` 은 `(permission >> 3) & 0b111` 에 해당한다. 하지만 이때 주의해야할 점은 최초 부팅 시에도 getPermission 을 통해 권한을 체크한다면 문제가 생길 수 있기 때문에, 현재 user가 존재하지 않는다면, 권한 체크를 하지 않도록 분기 처리를 해 주었다.

이 `getPermission()` 을 통해 권한을 체크하는 곳으로는 `namex()`, `sys_open()`, `sys_mkdir()`, `sys_chdir()`, `sys_unlink()`, `create()` 가 있다. 예외적으로 `sys_chmod()` 의 경우에는 파일의 권한으로 결정되는 것이 아닌 root 계정이거나 owner이어야지만 호출을 할 수 있기 때문에 `getPermission()` 을 사용하지 않는다.

Change Mode

바로 위에서 서술했듯이 Change Mode를 담당하는 system call chmod 는 file의 owner 또는 root 계정만 호출할 수 있는 system call이다.

sys_chmod

1. 해당하는 pathname의 `inode` 를 `namei` 를 통해 가져온다.
2. `chmod()` 를 호출한다.
3. 변경사항을 `iupdate()` 를 통해 반영한다.

chmod

1. 현재 user가 root 계정인지 혹은 `inode` 의 owner인지 확인한다. 만약 맞다면, `ip→permission` 을 인자로 들어온 mode로 바꿔준다.

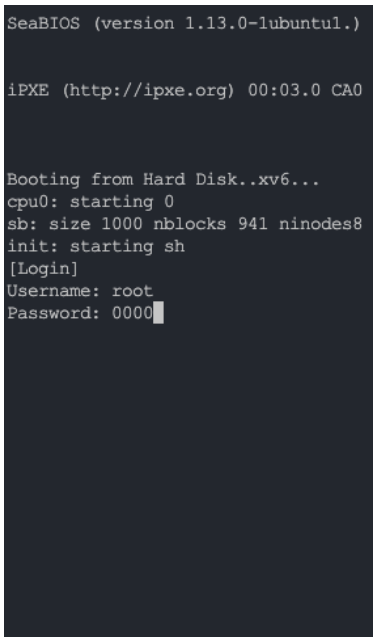
Modification of ls

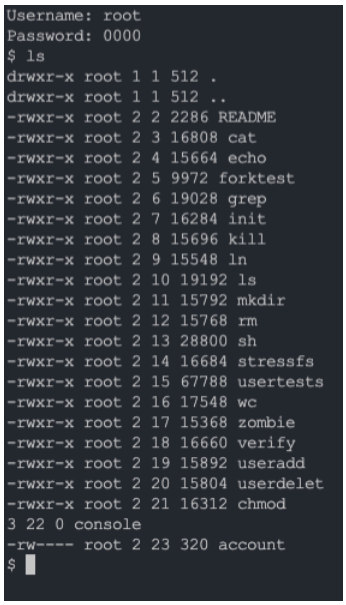
ls 의 내부에서 `stat()` 을 호출한다. `stat()` 은 내부에서 `fstat()` system call 을 호출하고 이 본체는 `filestat()` 이라는 함수에 있다. `filestat()` 은 내부에서 `stati()` 를 호출하는데 stat 구조체에 inode의 내용을 옮겨담아 ls 에서 정보를 볼 수 있도록 한다. 하지만 ls에서 `stat()` 을 호출할 때 open system call을 호출하는데, 만약 read 권한이 존재하지 않으면 파일을 열 수 없도록 하였다.

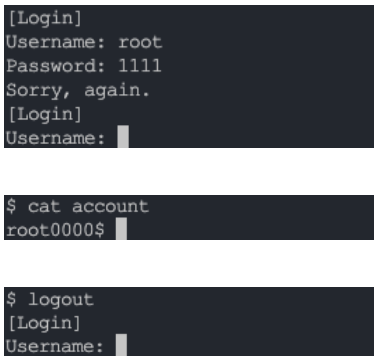
또한 file의 type이 T_DEV라면 기존의 xv6 에서처럼 출력하였고, T_FILE이나 T_DIR 라면, permission → owner → type → ino → size → fmtname 순서대로 출력하였다. 또한 `ls 특정디렉토리` 라는 명령어를 입력하였을 때, 실행권한이 존재하지 않는다면, 내부의 파일 목록을 볼 수 없다.

Result

login & logout

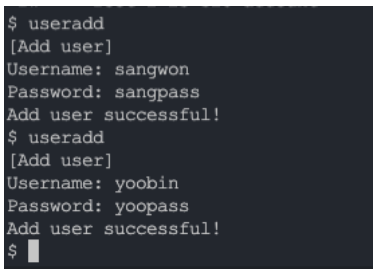


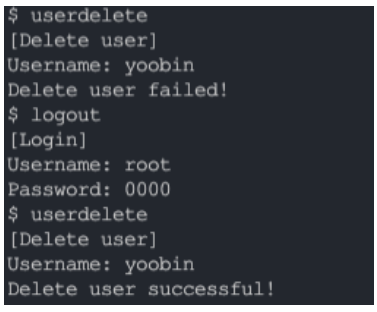


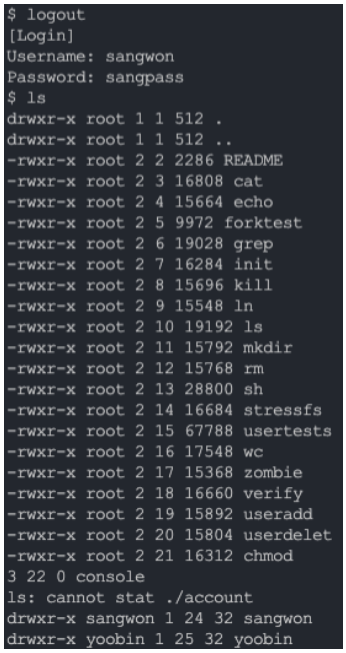


최초 부팅 후, 로그인 화면이 나오고, root 계정으로 로그인하면, 로그인이 완료된다. 만약, 계정의 정보가 틀리다면 우측 사진처럼 다시 로그인 화면이 나온다. 유저 정보는 우측 하단 사진에 보이듯이 account file에 기록된다. 또한 logout을 입력하면 다시 로그인 화면이 나온다.

useradd & userdelete

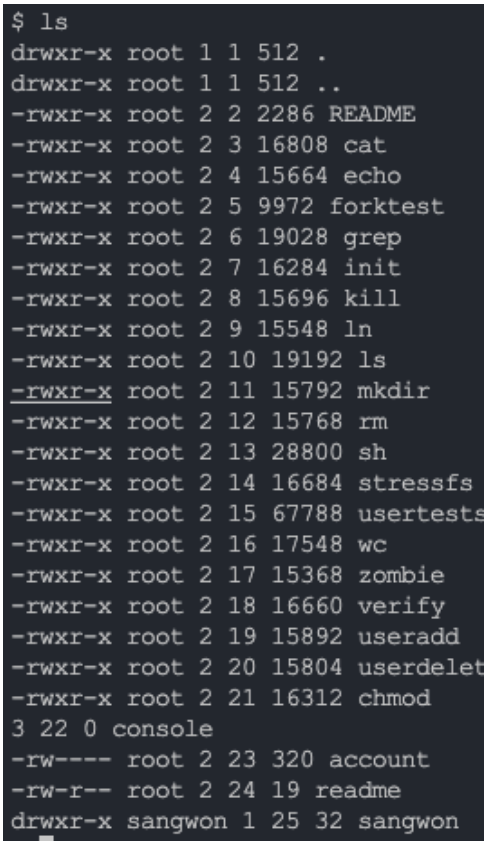


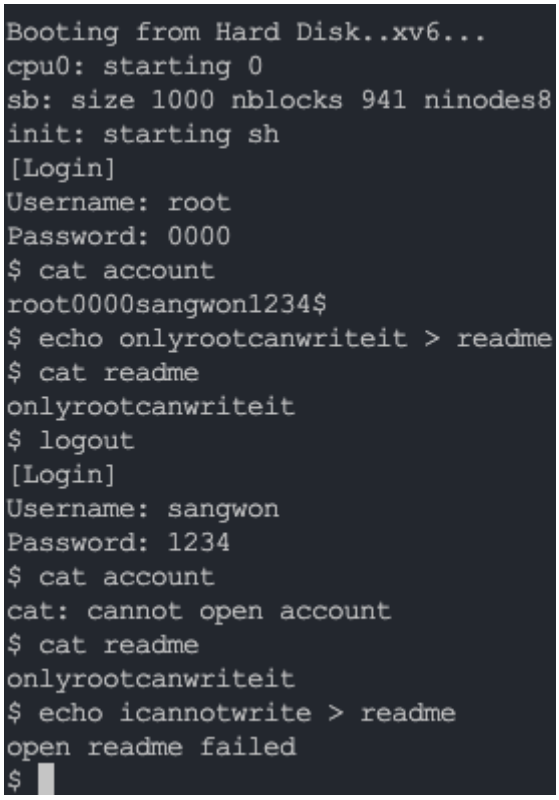




root 계정에 로그인 된 채로, useradd command를 입력하면 user를 추가할 수 있다. 그리고 새롭게 만든 user로 로그인을 할 수 있는데, ls 명령어를 입력했을 때, read 권한이 없으면 `stat()` 을 호출할 수 없다. 또한 만들어진 user의 directory가 생성되는 것을 확인할 수 있다. root 계정이 아닌 user로 로그인 한 후 userdelete를 하면, user 삭제가 불가능한 것을 확인할 수 있다.

File Mode

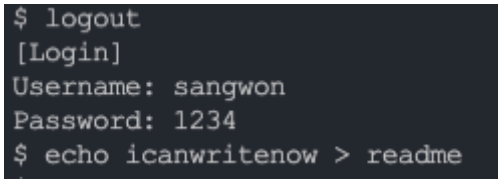


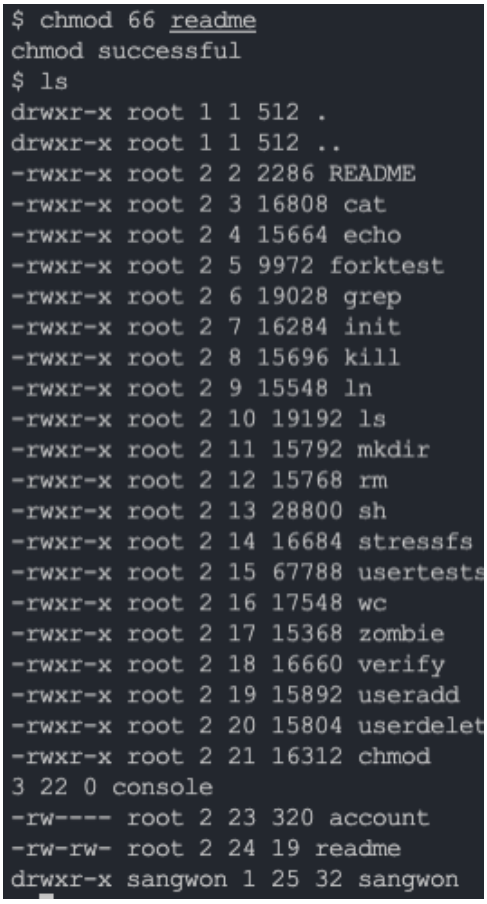


root 계정은 user 들의 계정 정보가 담긴 account file을 읽을 수 있다. 왜냐하면 account의 권한은 -rw—— 으로 root의 경우 읽고 쓸 수 있기 때문이다. 또한 root 가 echo를 통해 readme라는 파일을 만들었고, 이 파일의 권한은 -rw-r— 이다.

일반 계정의 경우 account file에 대한 어떠한 권한도 없기에 파일을 열 수 없다. readme 파일의 경우는 읽기 권한은 존재하기 때문에 cat을 통해 읽는 것은 가능하지만 작성하는 것은 불가능하다.

Change Mode





이제 root 계정이 `chmod 66 readme` 라는 command를 입력하면, others 에 대한 읽기 권한과 쓰기 권한이 활성화 된다. 따라서 일반 계정으로 로그인하고, write 했을 때 작성이 가능한 것을 확인할 수 있다.

Trouble Shooting

`inode` 를 다루는 것에서 많은 문제를 겪었다. lock 을 잡는 것 뿐 아니라 sequence 도 중요하기 때문에 잘 분석하는 것이 중요했다.