**Project 3: User System in xv6**

ELE3021-11797

2016024875 손정우

1. **디자인 Design**
   1. User Accounts
      * As suggested in the project specification, what I focused on implementing user functions is as follows.
        + Input/output method of the file that stores the username and password list
        + Change the order of process execution after the init process
        + Handling exceptions that can occur when processing strings
      * In the case of a file that stores a list of usernames and passwords, I named it 'shadow' with reference to the UNIX system. I thought it was very tricky to work with file I/O at the inode level below, not at the file descriptor level, so I devised a whole array of user information.
      * In the case of the init process, sh is originally executed. However, the process was changed to implement the presented login function and user function. Therefore, the changed execution order is init -> login -> sh, where login is the user program for login prompt.
      * Besides, a variable that stores the userid of the process execution is added to the struct proc, and through this, the useradd() and userdel() permissions are checked. String handling fails at the moment of neglect. Therefore, the program was designed with caution for possible exceptions.
   2. Login / Logout & User add / delete
      * When implementing user creation, deletion, and login functions, I focused on minimizing disk I/O.
      * I thought that continuous disk I/Os can significantly drain the performance of the operating system. Therefore, when the system boots, the user information is structured and managed like a ptable.
        + The problem that can occur here is that data inconsistency between memory and disk can occur. Also, if lock is not managed like ptable, structure data can also have integrity problems.
      * The following solutions were applied to ensure data integrity.
        + Disk I/O occurs only when data in memory is changed.
        + The structure for user information also protected by the lock.
   3. File Mode / chmod / Modification of ls
      * It is designed to display the owner and file mode of a file by adding variables to the inode, dinode, and stat structures for the file mode.
      * The main thing to note when processing file mode is the console. For T\_DEV type files, the initial file mode is set to crwxrwx, and it is designed to return -1 when chmod tries to change the file mode.
      * The modified stat structure and fstat function allow the file mode and owner to be displayed when the ls program is executed.
2. **구현 Implementation**
   1. Overview
      * The following files have been modified: defs.h, init.c, proc.c, proc.h, sh.c, syscall.c, syscall.h, ulib.c, user.h, usys.S
        + Defs.h, syscall.c, syscall.h, user.h, and usys.S have been modified to add new system calls.
        + To implement the functions required in the project specification, the variable char userid[16] was added to the proc structure existing in proc.h.
        + In the case of init.c, the program to exec has been changed to login, and the phrase welcoming users and whoami and logout commands have been added to sh.c.
        + fork2() and wrapper functions have been added to proc.c.
      * The following files were added: PRJ03\_syscall.c, login.c
        + login.c has been created to implement the login prompt.
        + In PRJ03\_syscall.c, system calls and wrapper functions such as inituser(), useradd(), userdel(), usercheck(), and chmod() are implemented.
   2. User Accounts

The structure created to create and manage users in xv6 is as follows.

|  |
| --- |
| #define NUM\_USER 10  #define USER\_FIELD 32  struct {  struct spinlock lock; // lock  struct inode \* ptr; // inode ptr for updating the user info  char u[NUM\_USER][USER\_FIELD]; // array that stores user info  int status[NUM\_USER]; // status of array u  } utable; |

When xv6 is booting, the init process calls the inituser() system call. The role played by this function is to initialize the utable and return load\_shadow(). load\_shadow() is a function that reads shadow files from disk. The following is pseudo code of this function.

|  |
| --- |
| int  load\_shadow(char \*buf)  {  Variable declaration;    if((ip = dirlookup(dp, name, 0)) == 0){  If the file does not exist, then create;  } else {  Read the inode of the file;  }  Data read using the inode of the ‘shadow’ file through the readi() function is stored in buf;  } |

When logging in, sh and child processes must also have the privileges of the logged in user. Accordingly, the fork() system call was modified to create fork2(). fork2() is the same as fork(), but sets the executor of the newly created process as a string received as an argument. By calling this system call in login.c, upon successful login, sh is executed as the name of the user who successfully logged in.

* 1. Login / Logout

The system call added for the login function is usercheck(). This function returns 1 if the information of the input user is correct and 0 if it is not.

|  |
| --- |
| int  usercheck(char \*username, char \*password)  {  int i;  for(i = 0; i < NUM\_USER; i++){  if(!strcmp(&(utable.u[i][0]), username) && !strcmp(&(utable.u[i][0]) + USER\_FIELD/2, password)) {  return 1;  }  }  return 0;  } |

In the case of login.c, it works in a large frame as shown below, and the prompt() function processes the user information input by the user and puts it as an argument of the usercheck() function to check whether or not login is possible and returns.

|  |
| --- |
| int  main(int argc, char \*argv[])  {  int pid, wpid;  int flag = 0;  for(;;){  while((flag = prompt()) < 1) {  switch(flag) {  case -1:  printf(1, "login: you forgot to type ID or PW\n");  break;  case 0:  printf(1, "login: ID does not exist or wrong password\n");  break;  default:  printf(1, "login: unknown error occured\n");  break;  }  }  char \*args[] = { "sh", id, 0 };  pid = fork2(id);  if(pid < 0){  printf(1, "login: fork failed\n");  exit();  }  if(pid == 0){  exec("sh", args);  printf(1, "login: exec sh failed\n");  exit();  }  while((wpid=wait()) >= 0 && wpid != pid)  printf(1, "zombie!\n");  }  } |

* 1. User add / delete

For the implementation of useradd() and userdel(), the existing create() and sys\_mkdir() system calls were referenced. Since both useradd() and userdel() are system calls where user information is updated, not only the value of utable in memory is modified, but also the file on disk is updated.

Among them, useradd() was implemented as follows. Userdel() is implemented similarly to useradd() and is omitted.

|  |
| --- |
| int  useradd(char \*username, char \*password)  {  Variable declaration;  Check if the user who called the system call is root;  Check if there is space to create additional users;  Check if the entered username and password are correct;  acquire(&utable.lock);  Store user information in utable;  Copy utable.u’s data to utemp;  release(&utable.lock);  begin\_op();  Write data from utemp to shadow file through writei() function;  Create user folder;  end\_op();  return 0;  } |

* 1. File Mode

To implement file mode, char userid[16] and int mode are added to dinode, inode, and stat structures. At this time, mkfs.c also changed the NDIRECT value from 12 to 7 to correspond to the changed structure. Note that to cope with this change in fs.c, not only the ialloc() function but also the iupdate() and ilock() functions need to be added with code to change the userid and mode values.

As suggested in the project specification, file permission check could also be implemented by creating a new part to check the permissions related to file mode in namex(), exec(), create(), sys\_open(), sys\_chdir(), and sys\_unlink().

* 1. Change Mode

I implemented chmod system call by finding the inode pointer of the file received from the wrapper function sys\_chmod() through namei() and changing it in chmod().

|  |
| --- |
| int  chmod(struct inode \*ip, int mode)  {  ip->mode = mode;  iupdate(ip);  return 0;  }  int  sys\_chmod(void)  {  char \*pathname;  int mode, ret;  struct inode \*ip;  if(argstr(0, &pathname) < 0 || argint(1, &mode) < 0)  return -1;  begin\_op();  if((ip = namei(pathname)) == 0) {  end\_op();  cprintf("chmod: invalid path\n");  return -1;  }  ilock(ip);  if((strcmp(myproc()->userid, "root") != 0) && (strcmp(myproc()->userid, ip->userid) != 0)){  cprintf("chmod: privilege error (only owner of the file or root!)\n");  iunlock(ip);  end\_op();  return -1;  }  if(ip->type == T\_DEV) {  cprintf("chmod: cannot change T\_DEV (e.g. console)\n");  iunlock(ip);  end\_op();  return -1;  }  ret = chmod(ip, mode);  iunlock(ip);  end\_op();  return ret;  } |

* 1. Modification of ls

The format of the output of ls is as follows:

File name | Owner | File mode | inode number | File size

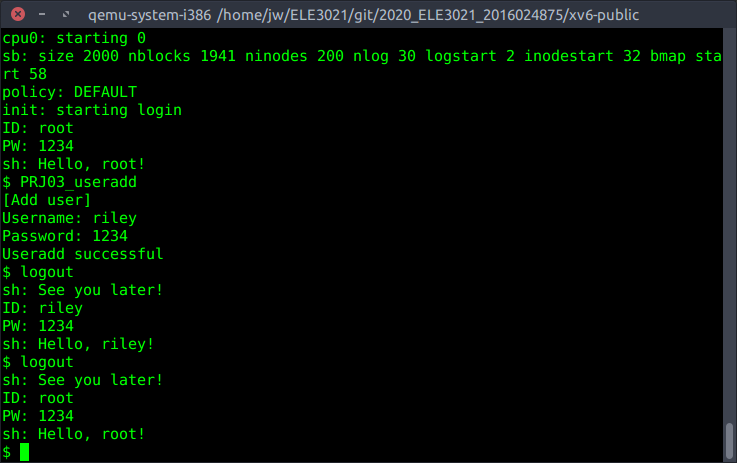
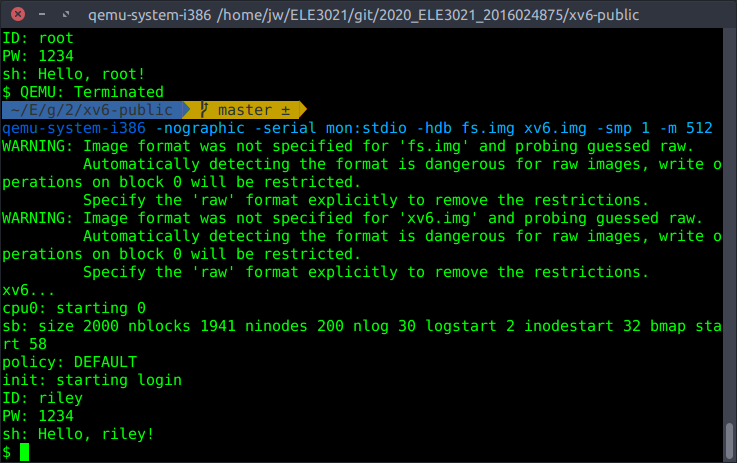
I removed st.type in the ls result, since File mode has file type in its expression.

The stati() function was modified to pass the necessary information to ls. In addition, the following functions were added to ls.c to visualize file mode information.

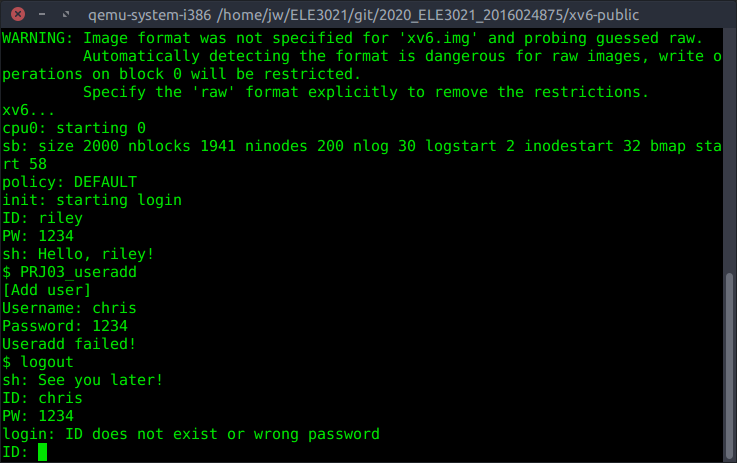
|  |
| --- |
| char\*  auth(char \*buf, short type, uint mode)  {  for(int i = 0; i < 7; i++)  buf[i] = '-';  buf[7] = '\0';  switch(type){  case T\_DIR:  buf[0] = 'd';  break;  case T\_FILE:  buf[0] = '-';  break;  case T\_DEV:  buf[0] = 'c';  break;  default:  printf(2, "ls: unknown file type\n");  exit();  }  if(mode & MODE\_RUSR) buf[1] = 'r';  if(mode & MODE\_WUSR) buf[2] = 'w';  if(mode & MODE\_XUSR) buf[3] = 'x';  if(mode & MODE\_ROTH) buf[4] = 'r';  if(mode & MODE\_WOTH) buf[5] = 'w';  if(mode & MODE\_XOTH) buf[6] = 'x';  return buf;  } |

1. **실행결과 Test Results**

* Test Environments: Ubuntu 16.04.4 on VMware Workstation 15.5.2 (Allocated 2 cores and 2GB of memory)
  1. User Accounts

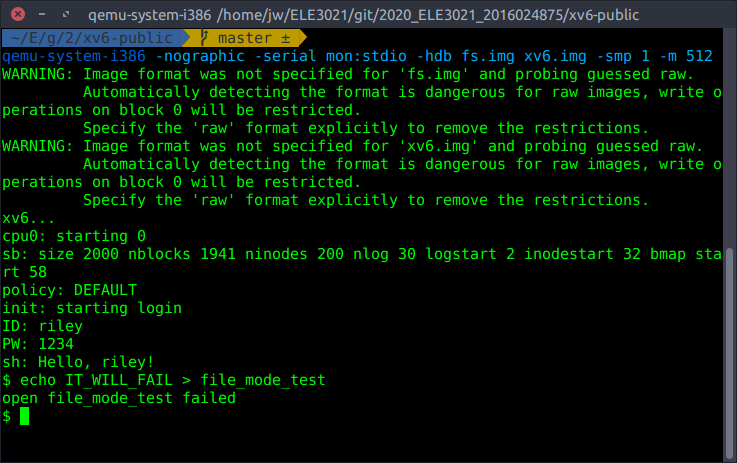
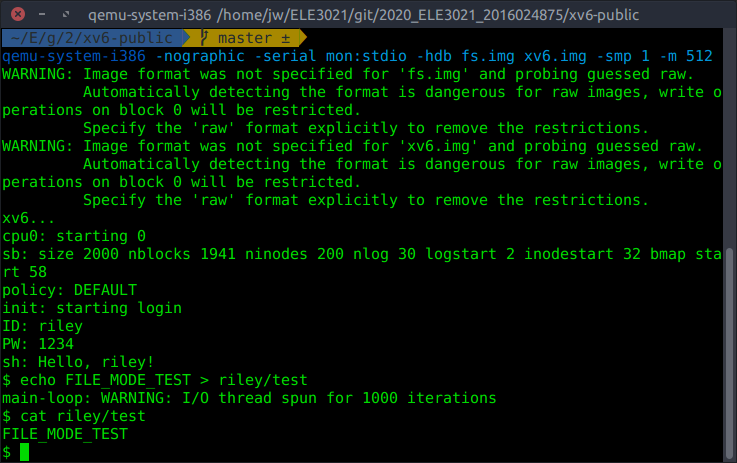
 

When useradd() is called by logging in with the root account, you can see that the user riley is registered normally.

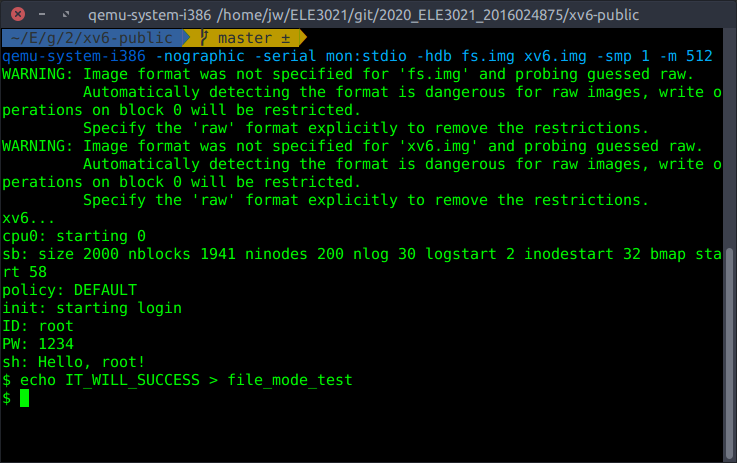
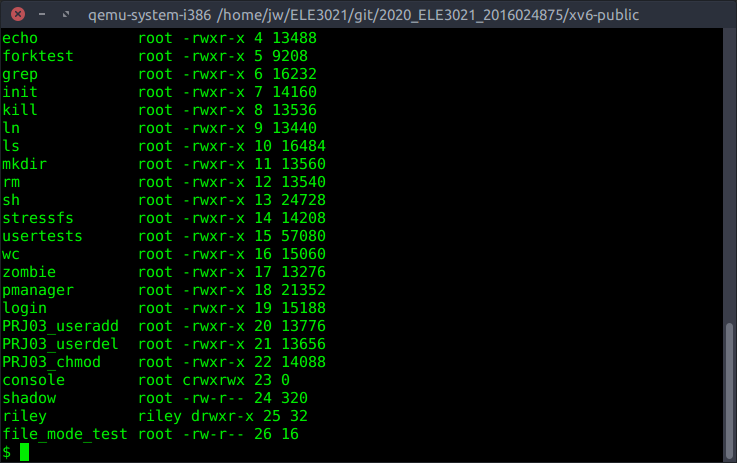


However, if you try to register the user chris with the user account riley, you will see that it fails.

* 1. File Mode

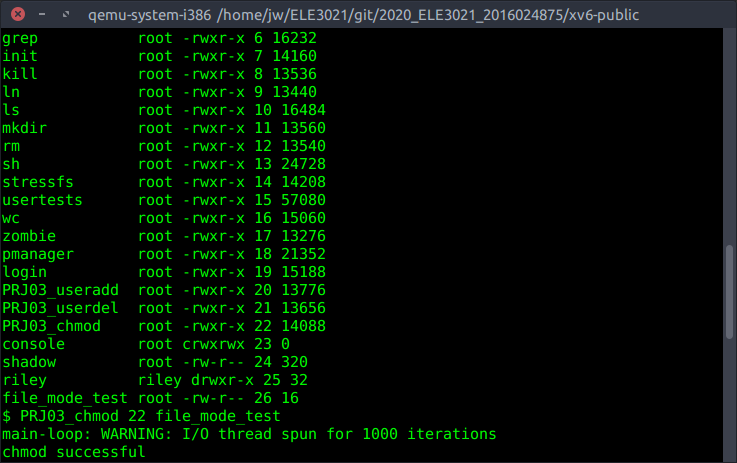
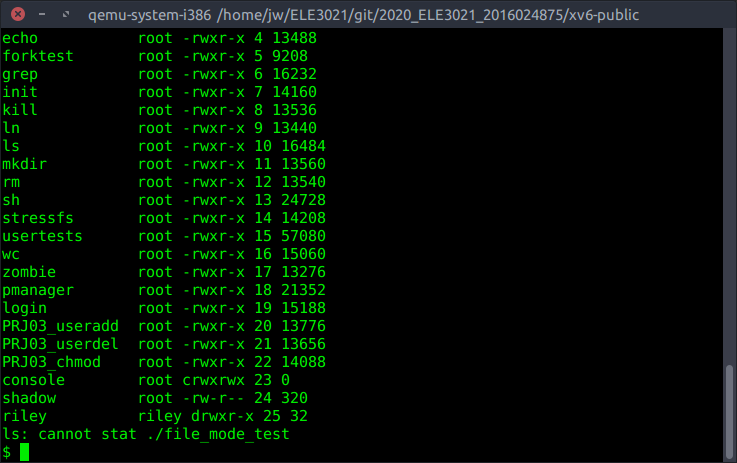
 

f you log in with a user ‘riley’, you cannot create files in the root directory. However, you can see that you can write files in your own directory ‘riley’.

However, if you log in with the root account, you can see that files can be created in the root directory.

* 1. chmod and ls

You can see that ls and chmod() also work fine through the above attempt.

1. **트러블슈팅 Trouble Shooting**

* Failed to build fs.img & balloc panic occurs when trying to create a new file while running xv6

In the case of these two problems, the file system implemented in fs.img was full of capacity. To solve the problem, the test user programs used in the previous project were removed from the Makefile, and the FSSIZE value of param.h was increased to 2000.

* chmod() system call gets an incorrect inode pointer

It was a problem caused by incorrect use of ilock() and iunlock() functions. The problem was solved by redesigning the system calls and wrapper functions.

* sched lock panic occurs when useradd() and userdel() are executed

There was a problem managing the lock of the newly declared utable. I could resolve this problem by adjusting the location of utable.lock in the useradd() and userdel() functions.