OS 4375/Lab 7/ xv6 system/Sonam S Tshering/

Task 1a. Look at the ls code in user/ls.c and explain how the code works in terms of the library and system calls it makes. Show the output from the ls command and interpret the different columns.

The ls code in user/ls.c implements the command utility ls that is used to list the contents of the directory. The code includes header files like types.h, stat.h and fs.h from kernel and user.h from user. These files provide definitions and declarations for types, file systems and user-level functions. The first function 'fmtname' formats the name of a file and pads the name with spaces to ensure that it has more than the fixed length 'DIRSIZ+1' and returns the newly formatted name. The second function 'ls' takes the path as an argument and list the contents of the directories of that specific path. It opens the directory using 'open and retrieves the information using 'fstat'. The third 'main' function is where it calls the 'ls' function for the current dir if arg<2 or else it calls ls for the arg provided.

```
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6/user$ ls
_cat         cat.sym       initcode.asm  mkdir.o     sh.d          uptime.c
_echo        echo.asm      initcode.d    mkdir.sym   sh.o          uptime.d
_forktest    echo.c        initcode.o    printf.c    sh.sym        uptime.o
_grep        echo.d        initcode.out  printf.d    sleep.asm     uptime.sym
_grind       echo.o        kill.asm      printf.o    sleep.c       user.h
_init        echo.sym      kill.c        ps.asm      sleep.d       usertests.asm
_kill        forktest.asm  kill.d        ps.c        sleep.o       usertests.c
_ln          forktest.c    kill.o        ps.d        sleep.sym     usertests.d
_ls          forktest.d    kill.sym      ps.o        stressfs.asm  usertests.o
_matmul      forktest.o    ln.asm        ps.sym      stressfs.c    usertests.sym
_mkdir       grep.asm      ln.c          pstest.asm  stressfs.d    usys.S
_ps          grep.c        ln.d          pstest.c    stressfs.o    usys.d
_pstest      grep.d        ln.o          pstest.d    stressfs.sym  usys.o
_pstree      grep.o        ln.sym        pstest.o    time.d        usys.pl
_rm          grep.sym      ls.asm        pstest.sym  time1.asm     wc.asm
_sh          grind.asm     ls.c          pstree.asm  time1.c       wc.c
_sleep       grind.c       ls.d          pstree.c    time1.c.save  wc.d
_stressfs    grind.d       ls.o          pstree.d    time1.d       wc.o
_time1       grind.o       ls.sym        pstree.o    time1.o       wc.sym
_uptime      grind.sym     matmul.asm    pstree.sym  time1.sym     zombie.asm
_usertests   init.asm      matmul.c      rm.asm      ulib.c        zombie.c
_wc          init.c        matmul.d      rm.c        ulib.d        zombie.d
_zombie      init.d        matmul.o      rm.d        ulib.o        zombie.o
cat.asm      init.o        matmul.sym    rm.o        umalloc.c     zombie.sym
cat.c        init.sym      mkdir.asm     rm.sym      umalloc.d
cat.d        initcode      mkdir.c       sh.asm      umalloc.o
cat.o        initcode.S    mkdir.d       sh.c        uptime.asm
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6/user$
```

Task 1b. Look at the mkdir code in user/mkdir.c and explain how the code works in terms of the system call it makes and the OS helper functions used by that system call. Use mkdir and some other commands (e.g., echo) to create a new directory and a couple of files in it. Run ls again and explain the output.

Mkdir.c also has header files to provide definitions and declarations for types, file systems and user-level functions. The main function being the entry point of the code, it checks if the number of command line arguments, or argc <2 which means no directory names were given. In that case, it prints out a message and exits the code with 0. Then it iterates through the provided

directory names, argv[i] and tries to create each directory using **mkdir (**sys call used to create new directory) system call. After creating directories successfully, it exits with 1.

```
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6$ mkdir my-dir
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6$ ls
LICENSE   README  kernel  my-dir
Makefile  fs.img  mkfs    user
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6$ ls -l
total 1032
-rw-r--r-- 1 sstshering sstshering    1174 Sep 19 13:04 LICENSE
-rw-r--r-- 1 sstshering sstshering    4793 Dec 15 11:25 Makefile

-rw-r--r-- 1 sstshering sstshering    2226 Sep 19 13:04 README
-rw-r--r-- 1 sstshering sstshering 1024000 Dec 13 02:28 fs.img
drwxr-xr-x 2 sstshering sstshering    4096 Dec 15 11:25 kernel
drwxr-xr-x 2 sstshering sstshering    4096 Dec 13 03:33 mkfs
drwxr-xr-x 2 sstshering sstshering    4096 Dec 15 12:00 my-dir
drwxr-xr-x 2 sstshering sstshering    4096 Dec 15 11:25 user
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6$ cd my-dir
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6/my-dir$ ls
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6/my-dir$ echo "This is
a new file" > file.txt
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6/my-dir$ ls
file.txt
sstshering@DESKTOP-20BNCM1:~/OS-riscv-xv6/my-dir$
```

Task 1c. Look at the ln code in user/ln.c and explain how the code works in terms of the library and system calls it makes. Use ln to create one or more links to a file you created in Task 1b. Run ls again and explain the output.

The ln.c also has header files to provide definitions and declarations for types, file systems and user-level functions. The main function checks if argc!=3, which means incorrect usage, and prints usage message and exits with error code. Then the **link** (used to create a new hard link to an existing file) system call is used to create a hard link between the old file and the new link, which is depicted by argv[1] and argv[2]. After the link has been created, the code exits with 0 = success, and if <0, error.

```
sstshering@DESKTOP-20BNCM1:~$ ln file.txt l
sstshering@DESKTOP-20BNCM1:~$ ls
OS-riscv-xv6  file.txt  linkfile.txt  mydir
sstshering@DESKTOP-20BNCM1:~$
```

Task 2a. Look at the code in kernel/sysfile.c for the fstat() system call and in kernel/file.c for the filestat() helper function and explain how the codes work.

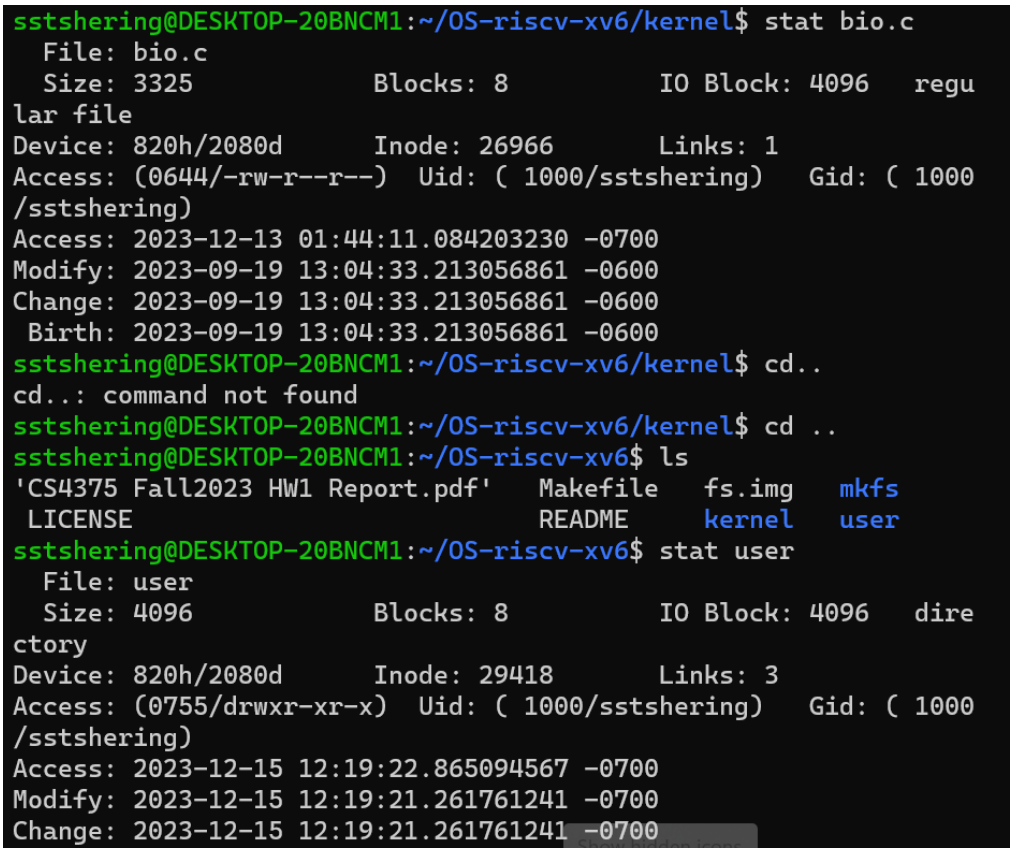
The **fstat()** system call in sysfile.c is responsible for retrieving file status information. It fetches the **argfd** (file descriptor) as an argument and checks if its valid and then

fetches **argaddr** (user space addr) for the **struct stat**. It then calls **the filestat()** to retrieve the file info and returns the result of this call, otherwise -1 on error/failure.

The **filestat**() sys call on file.c checks if the file type is supported, if so it locks the related inode and retrieves the file info using the **stati (**copies relevant info from inode to the struct stat) function and then unlocks the inode. It then copies the file info to the user space.

Task 2b. On a Linux system, look at the stat man page. Run the stat command on a regular file and on a directory and explain the output.

The stat command on file and directory gives output with the file name, size, blocks and all other related information regarding the file/ directory.



Task 2c. In your hw7 branch, implement an fstat command for xv6 that takes the name of a regular file or a directory as an argument and outputs the information for that file. The information output should include the filename, file type, file size, inode number, and number of links in an easily understandable format.

```
uint64
sys_fstat(void)
{
  char path[MAXPATH];
  struct file *f;
  struct stat st;

  if (argstr(0, path, MAXPATH) < 0)
    return -1;       You, 19 hours ago • hw7

  if (filestat(f, (uint64)&st) < 0)
    return -1;

  printf("File: %s\n", f);
  printf("Type: %s\n", (st.type == T_FILE) ? "Regular File" : "Directory");
  printf("Size: %d bytes\n", st.size);
  printf("Inode: %d\n", st.ino);
  printf("Links: %d\n", st.nlink);

  return 0;
}
```