

CS236 Operating Systems Lab

Spring 2025-26

Lab 11: file system voodoo

xv6 is a teaching operating system (developed at MIT) — an OS meant for teaching design and implementation concepts of an OS.

To install xv6 on Ubuntu, look up the details described in **Appendix A**.

The Holy Grail for xv6

- [xv6-risc book](#)
- [xv6-risc src booklet](#)

The xv6 source tree is organised to mirror OS subsystems:

- **kernel/** → process management, memory, file system, traps
- **user/** → user programs
- **mkfs/** → file system creation tool

Each directory corresponds to a logical OS component, enabling systematic exploration and modification. **You will mostly interact with the **kernel/** and **user/** directory.**

Submission Guidelines

Create a patch with:

```
Shell
> ./patch.sh
```

Rename the generated **lab11.patch** to **<rollno>_lab11.patch** and submit via Moodle.

1. The file to block journey

Implement the following xv6 system call —

```
void fstrace(int fd, int arg);
```

The call should print out the following statistics given a file descriptor,

[pid]

<fd> <address of file struct> <offset in file struct> <inode number>

<size> <reference count of inode>

<arg> <index of block in inode> <block address on disk>

<block in block cache>

- - -

if **arg** is -1 use the current offset in the corresponding file struct

and if **arg** ≥ 0 and **arg** $<$ **filesize**, interpret arg as file offset and print the statistics relative to the mentioned arg-offset. To get filesize you might need to query the inode. Print the whole output with a single printf call, so that the output doesn't get mixed due to multiple (concurrent) calls.

Note:

1. xv6 supports different file types — pipe, device, inode.
This question concerns itself inode-backed files, defined as file type **FD_INODE**
2. The in-memory inode pointer (in fact all/most file system objects) should be used in a mutual exclusion manner — check usage of the inode lock.
3. The xv6 **bmap** function, given the block number in inode, provides the corresponding block address, but also allocates a new block if the block number isn't allocated yet. For our purpose, we only need to query the block address and no allocation should be done.
4. These files might be of your interest: **kernel/sysfile.c**, **kernel/fs.c**, **kernel/bio.c**. Feel free to create relevant helper functions there, take help of existing functions to learn how to use specific locks.

Sample outputs (detailed description given in the corresponding C file)

- check if anything is in the cache without any read/writes

```
Shell
$ p1a
pid: 3
fd: 3
file struct address: 0x000000008001fbc0
offset in file struct: 0
inode number: 10
file size: 40272B
```

```
ref count of inode: 1
arg offset: 0
index of block in inode: 0
block address on disk: 301
block is in cache?: no
```

- opening the same file twice creates two file structs, both pointing to the same inode, which can be seen via the inode's ref count.

```
Shell
$ p1b
pid: 3
fd: 3
file struct address: 0x000000008001fbc0
offset in file struct: 10
inode number: 10
file size: 40272B
ref count of inode: 2
arg offset: 10
index of block in inode: 0
block address on disk: 301
block is in cache?: yes
pid: 4
fd: 3
file struct address: 0x000000008001fbe8
offset in file struct: 10
inode number: 10
file size: 40272B
ref count of inode: 2
arg offset: 10
index of block in inode: 0
block address on disk: 301
block is in cache?: yes
```

- using duplicated fds, both of them point to the same file struct, affecting read/writes using these fds.

```
Shell
$ p1c
pid: 3
fd: 3
file struct address: 0x000000008001fbc0
offset in file struct: 10
inode number: 10
file size: 40272B
ref count of inode: 1
arg offset: 10
index of block in inode: 0
block address on disk: 301
block is in cache?: yes
pid: 3
fd: 4
file struct address: 0x000000008001fbc0
offset in file struct: 10
inode number: 10
file size: 40272B
ref count of inode: 1
arg offset: 10
index of block in inode: 0
block address on disk: 301
block is in cache?: yes
```

- opening the same file twice again, the read/writes using one fd shouldn't affect those through the other fd.

```
Shell
$ p1d
pid: 3
fd: 3
file struct address: 0x000000008001fbc0
offset in file struct: 0
inode number: 10
file size: 40272B
ref count of inode: 2
arg offset: 0
index of block in inode: 0
block address on disk: 301
block is in cache?: yes
```

```
pid: 3
fd: 4
file struct address: 0x000000008001fbe8
offset in file struct: 10
inode number: 10
file size: 40272B
ref count of inode: 2
arg offset: 10
index of block in inode: 0
block address on disk: 301
block is in cache?: yes
```

- check your argument!

```
Shell
$ p1e
arg beyond file size
```

try to experiment with more such test cases using the fstrace syscall.

2. to sudo or unsudo, that is the question?

The aim of this exercise is to add two features to xv6 —

- (i) user level — **sudo** (the admin) and **unsudo** (the normal user)
- (ii) per file read/write privileges when in sudo mode and when in unsudo mode

By default, the **sudo** mode has read/write privileges for all files, while in **unsudo** mode each file can have a specific read/write permission. Also, by default, all files are created with sudo mode.

Implement/update the following user programs and required system calls —

```
sudo                // switch to sudo user
unsudo             // switch to unsudo user
chmod pathname R/W 0/1 // change R/W permissions for unsudo
                        // pathname specifies the file
ls                 // print the R/W permissions in unsudo mode for each entry
```

File operations – read and write should now check the user level and then check the file permissions to allow or disallow the operation.

Notes:

1. The disk inode structure size should be a power of 2 (divide a block exactly).
2. The directories leading to a file must be given permissions (/ or . should be given R and/or W permissions)
3. User level is a kernel wide property not a per process
4. Transactions must be used while writing back permissions into the disk inode (check usage of begin_op and end_op functions)
5. Add a new make target which does not depend on **fs.img** to test for persistence of user-level permissions (by default **make qemu** creates a new **fs.img**).

```
C/C++
qemu-nofs: check-qemu-version $K/kernel
          $(QEMU) $(QEMUOPTS)
```

Sample output

0 indicates no permission and 1 indicates permission in the unsudo mode for the read and write permissions.

E.g., chmod has been used to set read permission of the binary **sudo** for the unsudo mode.

Shell

\$ ls

1	0	.	1	1	1024
1	0	..	1	1	1024
0	0	README	2	2	2425
0	0	file2.txt	2	3	24
0	0	cat	2	4	38296
0	0	echo	2	5	37080
0	0	forktest	2	6	18424
0	0	grep	2	7	41848
0	0	init	2	8	37464
0	0	kill	2	9	37008
0	0	ln	2	10	36800
0	0	ls	2	11	40488
0	0	mkdir	2	12	37072
0	0	rm	2	13	37056
0	0	sh	2	14	61016
0	0	stressfs	2	15	37928
0	0	usertests	2	16	194936
0	0	grind	2	17	53592
0	0	wc	2	18	39328
0	0	zombie	2	19	36336
0	0	logstress	2	20	39088
0	0	forphan	2	21	37848
0	0	dorphan	2	22	37272
1	0	sudo	2	23	36424
0	0	whoami	2	24	36616
0	0	unsudo	2	25	36440
0	0	chmod	2	26	37304
0	0	console	3	27	0

Appendix A

Follow these steps for installing xv6 on a Ubuntu-based Linux machine:

1. Prerequisites

- a. First, update your system:

sudo apt update

- b. Install the required packages:

**sudo apt install -y git build-essential qemu-system-riscv64
gcc-riscv64-unknown-elf**

- **build-essential**

For essential packages like libc, gcc, make, etc.

- **qemu-system-riscv64**

Emulates a RISC-V machine in software.

Your computer's real CPU speaks x86-64, but xv6 is written for RISC-V.

QEMU **pretends to be a RISC-V computer**, so the OS can run even though the hardware is different.

- **gcc-riscv64-unknown-elf**

It runs on your x86-64 machine but produces **RISC-V binaries**, not x86-64 ones.

We need this because xv6 must be compiled for the **CPU architecture that QEMU is emulating**.

2. Verify toolchain installation

riscv64-unknown-elf-gcc --version

qemu-system-riscv64 --version

If both commands work, your environment is ready.

3. Clone the xv6 RISC-V Repository

git clone <https://github.com/mit-pdos/xv6-riscv.git>

cd xv6-riscv

4. Build xv6

make

Run the command to build the OS. If compilation succeeds, you are ready to run xv6.

5. Run xv6 on QEMU

Start the xv6 operating system:

make qemu

Appendix B

A list of riscv ISA items and xv6 items that intersect with the xv6 system call implementation — For details of these and more, look up Chapter 4 of the xv6 book.

None

ecall

sret

stvec

sepc

scause

sstatus (SIE and SPP)

sscratch

satp

csr

csrrw

a0

TRAMPOLINE

TRAPFRAME

uservec

usertrap

userret

prepare_return

syscall

copyin

copyout

copyinstr

argint

argstr