Chapter 39 Files and Directories

So far we have covered **virtualization** of

- The CPU —> process
- Memory (volatile) —> the address space

Expand virtualization to **persistent storage (non-volatile memory)**

- keep stored content even after power loss
- Hard disk drive (HDD)
  - Block addressable
- Solid-state storage device (SSDs)
  - Block addressable
- Non-volatile memory (NV memory):
  - Optane (3D X-point)
  - byte addressable

31.1 Files and Directories

Two key abstractions for virtualization of storage

1. File
   - Array of bytes
   - Low-level name: inode identifier
   - OS does not know the structure of the file
   - OS responsibility:
     - Persistently store data and
     - Retrieve data again (without corruption)
   - File name parts: bar.txt
     - Bar and txt (separate period .)
     - Just a convention; not enforced by the OS or file system

2. Directory

- Has structure
- Also has low-level name: inode number
- Contains entries (user-readable name, low-level name)
  - (“Foo”, 10)
- Each entry in a directory, either
  - A file
  - Other directory
- Placing directories in other directories
  - Can build arbitrary directory tree

- Root directory: /
  - Separtor: /
  - Sub-directories
  - The file system provides unified way to access
    - Files on disk
    - USB drives
    - CD-roms
- Absolute path names
  - /foo/bar/fizz.txt

39.3 Creating Files

To create a file, use the open system call:

int fd = open(“foo”, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR |S_IWUSR)

Second param:

- O_CREAT - create file if does not exist
- O_WRONLY - file can only be written to
- O_TRUNC - if file already exists, truncate it to size zero (remove existing content)

Third param:

- S_IRUSR - make file readable by owner (current user)
- S_IWUSR - make file writable by owner

Return:

- fd - file descriptor
  - Integer
  - Private per process
  - Use fd to read or write the file (assuming you have permission)
  - The fd as just a handle or pointer that gives you power to perform certain operations
  - Fd: as pointer to an object of type file
  - Call methods to access the file: read() and write()

Go has a similar function in the “os” package:

- os.OpenFile()

File descriptors managed by OS (per-process basis; kept in proc struct on Unix/xv6 kernel)

```
struct proc {
    …
    struct file *ofile[NOFILE];        // open files
    …
}
```

NOFILE = max # of files per-process

Each entry in array — just a pointer to a “file” struct.

Commands:

- strace (Linux)
- dtruss (macOS) — but must be enabled from recovery mode

Note on the open() call:

- O_RDONLY - file is opened as read-only by cat
- Returns success (positive value = fd = 3

Q: Why 3?

Already has three files open: stdin = 0, stdout = 1, stderr = 2

You can see these fds in other calls to read(3,…) and write(1, …)

read(fd, content, size) = size of content

write(fd, content, size) = size of content

write(1, …) write content to stdout/screen

cat tires to read more from the file:

read(3, “…”, size) = 0

So this returns 0, meaning that it has read the entire file

cat calls

close(3)

39.5 Reading and Writing, But Not Sequentially

Now we want to read or write to a specific offset within a file.

```
off_t lseek(int fd, off_t offset, int whence);
```

- fd - file descriptor
- Offset — position/location within the file
- Whence - how seek is performed
  - SEEK_SET - offset is set to offset in bytes
  - SEEK_CUR - offset is set to current location + offset bytes
  - SEEK_END - offset is set to size of file + offset bytes

xv6:

```
struct file {
    int ref;              // reference count
    char readable;        // opened as readable
    char writable;        // opened as writable
    struct inode *ip;     // points to underlying file
    uint off;             // current offset
}
```

Open(): offset initialized to zero

read(…, 100); - 100 bytes at a time

read(…, 100) returns 0 — done

close(fd)

OFT = Open File Table

Two entries (10, 11)

Two file descriptors (3, 4), but pointing to the same “file”

Current offset for each OFT entry is updated independently

Process uses lseek() to reposition current offset before reading (or writing)

- set current offset to 200
- Read next 50 bytes
  - Updates the current offset to 250

39.7 Writing Immediately with fsync()

For performance reasons:

- buffer writes in memory
- At later time (e.g. after 5 seconds), issue write ops to storage device
- Appears to calling application that write are quick

Issue: data may be lost

- rare case, e.g. due to machine crash after write() call, but before the write to disk takes place

This guarantee is called “eventual” durability and is “acceptable” for some systems.

However, many systems require strong durability guarantees.

- e.g. database management system (DBMS)

To support such applications

- fsync(int fd)
- Force all dirty (not yet written) data to be stored on disk
- Returns when all writes are complete (can take a bit of time)
- Application can safely move on knowing the data has been persisted

Note:

- may also need to fsync() the surrounding directory that contains the file foo
- If foo newly created

- Ensure that file foo is durable part of the directory

39.8 Renaming Files

% strace mv foo bar

Calls: rename(“foo”, “bar”) = 0

Guarantee of rename(old, new):

- atomic with respect to system crashes
  - File will either be named “old” or “new”
  - No in-between state can arise

Ex. File editor (e.g. emacs); editing file foo.txt

- inserting a new line of text or code in the middle of the file
- To guarantee the new (saved) file has original contents plus the new line, do this:

```
int fd = open(“foo.txt.tmp”, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
write(fd, buffer, size);         // write out new version of file
fsync(fd);
close(fd);
rename(“foo.txt.tmp”, “foo.txt”);
```

Editor code:

- write new version of the file under a temporary name (foo.txt.tmp)
- Force it to disk with fsync()
  - Application is now certain that the new file metadata and contents are on disk
- Rename temporary file to original file’s name: foo.txt
- Atomically swapping new file into place; (deleting old version) — atomic file update

39.9 Getting Information About Files

File system keeps lots of info about each file it is storing:

- such data about files is called metadata
- Get metadata for a file, use
  - stat()
  - fstat() system calls

The stat struct contains key info:

- file size
- inode#
- Ownership info
- When the file was access/modified

Linux: % stat file

macOS: % stat -x file

39.10 Removing Files

% strace rm foo

unlink(“foo”) = 0

% rm -rf /

39.11 Making Directories

% strace mkdir foo

mkdir(“foo”, 0777) = 0

When created, the directory is considered empty.

Empty directories has two entires:

- “.” (dot) - refers to itself
- “..” (Dot-dot) - refers to the parent directory

39.17 Making and Mounting a File System

Q: How to assemble a full directory tree from many underlying file systems?

- First make file systems: mkfs
- Then mounting them to make their contents accessible: mount

Goal: To make a file system accessible within a *uniform file system tree.*

Mount:

- Take exiting directory as target *mount point*.
- “Paste” (or map) new file system onto directory tree at that point.

Example:

- ext3 file system
- One device partition: /dev/sda1
  - Content of device file system: directories: meling, lamport
- Want to mount this file system at mount point: /home/users

% mkdir -p /home/users

% mount -t ext3 /dev/sda1 /home/users

% ls /home/users/

meling lamport

% ls /usr/local/

Mount command:

- apfs - Apple file system (optimized for SSDs)
- Devfs - device file system
- On linux: (check on Unix machines…)
    - ext3: hard disk file system
    - **proc**: file system for accessing information about processes
    - tmpfs: file system for temporary files
    - nfs: network file system