6.181: Filesystems (pt. 1)

Adam Belay <abelay@mit.edu>
Why do we need filesystems?

• Durability across restarts and crashes
• Naming and organization
• Sharing data between processes and users
What makes them interesting?

• Crash recovery
• Performance + concurrency
• Sharing + security
• Powerful abstractions (e.g., proc, afs, 9P, pipes, etc.)
xv6 FS software layers

0: System calls
1: Names + FDs
2: Inodes
3: Inode cache + Buffer cache
4: Log
5: Virtio disk driver

Focus for today
High-level design choices in system calls

• Objects: Use files (not virtual disks or databases)
• Content: Use byte arrays (not structured)
• Naming: Human-readable (not ID numbers)
• Organization: Name hierarchy
• Synchronization: None (no locking, no versions)
  • link() and unlink() can change names concurrently w/ open()
0: System call layer

```c
fd = open("x/y", flags);   // creates a fd
write(fd, "abc", 3);      // writes 3 bytes
link("x/y", "x/z");       // creates a link
unlink("x/y");           // removes x/y
write(fd, "def", 3);      // writes 3 more bytes
close(fd);                // closes the fd
```

File x/z contains abcdef
xv6 FS software layers

0: System calls
1: Names + FDs
2: Inodes
3: Inode cache + Buffer cache
4: Log
5: Virtio disk driver

Focus for today
1: Name layer

- Path names are organized as a tree
- No cycles, but multiple names can refer to the same file (i.e., via link())
- Processes share the namespace
- But each process has a *current working directory* (CWD)
- Absolute path: /x/y
- Relative path: x/y
1: FD Layer

• Each process has its own FD number namespace

• Each FD identifies a file created by open()
  • By convention STDIN (0), STDOUT (1), STDERR (2)
  • Lowest available FD number is allocated during open()
  • Survives even if the file is unlinked (i.e., deleted)

• A file is an object that you can read and write to like a stream
Interacting with a file

- FDs access file as an array of bytes, very similar to an address space
- Each FD has a “cursor” to the file
Interacting with a file

- FDs access file as an array of bytes, very similar to an address space
- Each FD has a "cursor" to the file
- read() advances the cursor

```c
#include <unistd.h>

void read(int fd, void* buf, int size)
{
    // Read from file
}
```
Interacting with a file

- FDs access file as an array of bytes, very similar to an address space.
- Each FD has a "cursor" to the file.
- read() advances the cursor.
- write() does too.

```
read(fd, buf, 8)
write(fd, buf, 8)
```
Some files are special

- e.g., a pipe()
- Usage: int fds[2]; pipe(fds);

```
write(fd[1], ...)
```

Buffer

```
read(fd[0], ...)
```
# xv6 FS software layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>System calls</td>
</tr>
<tr>
<td>1:</td>
<td>Names + FDs</td>
</tr>
<tr>
<td>2:</td>
<td>Inodes</td>
</tr>
<tr>
<td>3:</td>
<td>Inode cache + Buffer cache</td>
</tr>
<tr>
<td>4:</td>
<td>Log</td>
</tr>
<tr>
<td>5:</td>
<td>Virtio disk driver</td>
</tr>
</tbody>
</table>

*Focus for today*
2: Inode layer

- **Inode**: Records the details of a file
  - Tracks the size of the file and where on the disk the data is stored
  - Has a link count (and open FD count) to figure out when to free
  - Deallocation deferred until link + open count is zero

- **I-number**: Refers to an inode, similar to an FD
  - Uniquely identifies a position on disk
Where is data stored?

• On a **persistent** storage medium
  • Data doesn’t go away under loss of power

• Common storage mediums
  • **HDDs**: High capacity, slow, inexpensive
  • **SSDs**: Lower capacity, faster, more expensive
  • More choices on the horizon

• Disks accessed in fixed-sized units (like pages)
  • Called **sectors**, historically 512 bytes
Performance characteristics

• Applies to both HDDs and SSDs
• Sequential access much faster than random
• Big reads/writes much faster than small ones
• Both facts influence FS design
Disk blocks

• Typically, multiple sectors are combined to form blocks
  • e.g., a 4KB block is 8 sectors
• Needed to reduce book-keeping and seek overhead
• xv6 uses two sector blocks
• Every block has a block number
  • think of it like an address that identifies the location on the disk
xv6 FS software layers

0: System calls
1: Names + FDs
2: Inodes
3: Inode cache + Buffer cache
4: Log
5: Virtio disk driver

Focus for today
3: Inode + buffer cache

• Problem: Disk accesses are slow and random
• Idea: Store copies of inodes and blocks in RAM
• Works well because the same data is often accessed many times
• e.g., the same inodes and blocks are accessed each time a file is read
• No need to access the disk if a copy is available!
On-disk layout

1: Superblock
2: Log head
3: Log blocks
32: Inodes
45: Free block bitmap
46: Actual data
xv6 provides mkfs program

• Generates this layout for a new (empty) FS
• The layout is static for the lifetime of the FS
• What is metadata?
  • Everything other than file content
  • Super block, inodes, bitmap, directory content
On-disk inode layout

- Type: Free, file, directory, or device
- Nlink: number of links
- Size: the size of the file in bytes
- Addrs: addresses of data blocks (array)
- Example: Find file’s byte at 4000
  - $4000 \div \text{BSIZE} (=1024) = 3$; Look at $3^{rd}$ addr entry

<table>
<thead>
<tr>
<th>Block #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>410</td>
<td>124</td>
<td>124</td>
<td>52</td>
</tr>
</tbody>
</table>
|         |     |     |     |     |...
Problem: Inode is fixed size!

- How can we fit large files into addr?
- Idea: Use indirect block: a full block of more addr

[Diagram showing indirect block structure]
How to turn i-number to inode?

• i-number functions as an index to a disk block
• But have to skip log metadata
• Each inode is 64 bytes long
• Inode(i-number) = 32*BSIZE + 64*i-number
What about directories

• Represented much like a file
  • But users can not directly write contents

• Content is an array of dirents

• Each dirent:
  • i-number (of the file in the directory)
  • 14-byte file name
  • dirent is free (unused) if inum == 0
On-disk structure is tree-based

• Layer 1: Directory tree
• Layer 2: Inodes
• Layer 3: Blocks

Allocation pools: Inodes and Blocks
Example: Writing a file
Concurrent in FS

• xv6 has modest goals
  • Parallel read/write of different files
  • Parallel pathname lookup
• Disk also operates concurrently (e.g., intr)
• Even these pose interesting challenges
Conclusion

• File system maintains address space-like view of disk blocks
• Uses trees (like a page table) for naming and tracking disk blocks
• Next lecture: more details of xv6 and logging