Filesystem API and interface

CSC 343, Operating Systems

Topics covered in this lecture

- Filesystem API
- Internal and external interface
- Inodes and devices
- File descriptors
- File names

This slide deck covers chapters 39, 40 in OSTEP.

Purpose of a file system (1/2)

- Given: set of persistent blocks from a storage device
- Goal: manage these blocks efficiently. How?
  - Who has access?
  - What about initialization / bootstrapping?
  - Structural organization?
- Manages data (mostly) on nonvolatile storage
- Enables users to name and manipulate semi-permanent files (select execeutables and their data)
- Provide mechanisms to organize files and their metadata (e.g., owner, permissions, or type)

Purpose of a file system (1/2)

- Given: set of persistent blocks from a storage device
- Goal: manage these blocks efficiently. How?
  - Who has access?
  - What about initialization / bootstrapping?
  - Structural organization?

- Manages data (mostly) on nonvolatile storage
- Enables users to name and manipulate semi-permanent files (select execeutables and their data)
- Provide mechanisms to organize files and their metadata (e.g., owner, permissions, or type)

Purpose of a file system (2/2)

- Map bytes on disk to “file”
- Share files (concurrently?) among users and processes
  - Decide on locking granularity and binding operations
  - Semantics of operations like truncating in a shared world
- File caching: metadata and contents

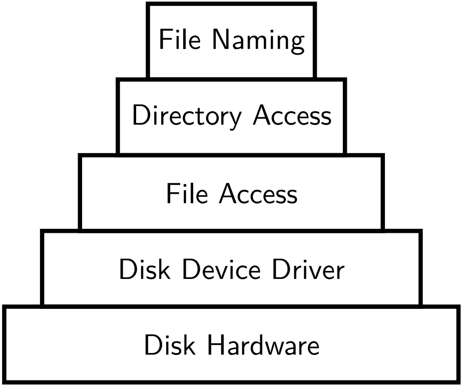
The `file` abstraction

- A file is a linear persistent array of bytes
  - Data operations: read or write
  - Metadata operations: create, delete, modify permissions/user/...
- Directory contains subdirectories
  - List of directories, files, inode mappings

The `file` abstraction

- Different perspectives
    - File name (human readable)
    - Inode and device number (persistent ID)
    - File descriptor (process view)

The I/O hierarchy



- Each level adds functionality (and complexity)

Different functionality

- ***Naming***
    - specifies name syntax and encoding
    - e.g., a URL; may be aware if a file is local/remote
- ***Directory access***
    - map name to file object
    - resolve a string to an object
- ***File access***
    - concerns file operations
    - e.g., create/delete/read/write

Different file philosophies

- Typed files: associate structure
  - System defines *all* possible file types (e.g., text document, source file, html file)
  - File type set at creation, file type specifies operations
- Untyped files: array of bytes
  - File is a sequence of bytes
  - System does neither understand nor care about contents
  - File operations apply to all files

- Modern systems settled on untyped files.
- Each file now has attributes or even extended attributes.
  - executable, append only, COW etc.
  - check manpage for `chattr`

Different file philosophies

- Typed files: associate structure
  - System defines *all* possible file types (e.g., text document, source file, html file)
  - File type set at creation, file type specifies operations
- Untyped files: array of bytes
  - File is a sequence of bytes
  - System does neither understand nor care about contents
  - File operations apply to all files

- Modern systems settled on untyped files.
- Each file now has attributes or even extended attributes.
  - executable, append only, COW etc.
  - check manpage for `chattr`

Desired file operations

- `create`: create a new file
- `unlink`: remove/destroy a file
- `open`: map a path to a file identifier
- `close`: close a file identifier
- `read`: read from the current file position
- `write`: write to the current file position
- `seek`: modify the file position
- `control`: various control operations such as changing permissions or user
- `createdir`: create a new directory
- `rmdir`: remove a directory
- `readdir`: return all files in a directory

Desired cost of file operations

- Sequential read/write is common
  - Design goal: `O(size of transfer)`
- Random access (seeking) is infrequent
  - Design goal: `O(log file length)`
- Constraints and observations
  - Many files are small
  - A few files are large
  - Most access is sequential, few accesses are at random positions
- Ideas for a clever data structure?

Desired cost of file operations

- Sequential read/write is common
  - Design goal: `O(size of transfer)`
- Random access (seeking) is infrequent
  - Design goal: `O(log file length)`
- Constraints and observations
  - Many files are small
  - A few files are large
  - Most access is sequential, few accesses are at random positions
- Ideas for a clever data structure?

The 3 views of a file

- ***Operating system:*** Inode and device id
    - Ids are unique and great and unambiguous
- User: file name
- Process: File descriptor

Managing files: inodes

- An `inode` contains metadata of a file
- Each file has exactly one associated inode
- Each inode is unique on a filesystem (not globally!)
- Inodes are recycled after reuse

- Note: multiple file names may map to the same inode
    - see "hard links"

Managing files: inodes

- An `inode` contains metadata of a file
- Each file has exactly one associated inode
- Each inode is unique on a filesystem (not globally!)
- Inodes are recycled after reuse

- Note: multiple file names may map to the same inode
  - see “hard links”

Table of inodes (1/2)

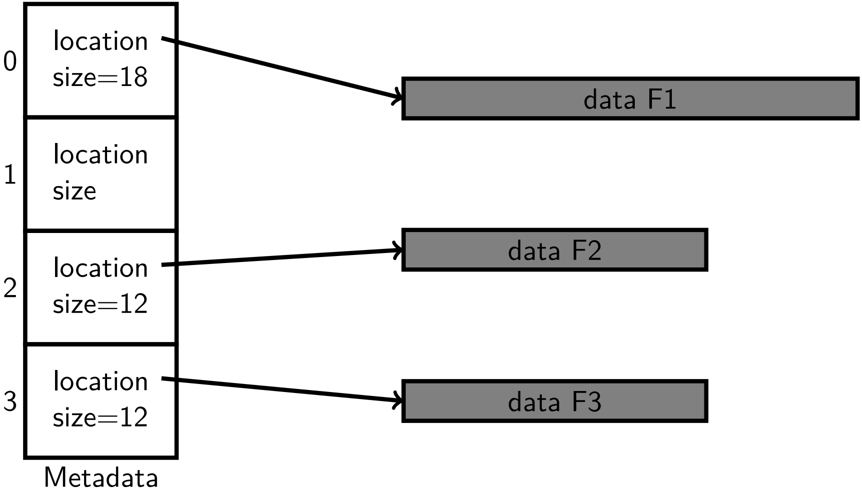


Table of inodes (2/2)

- Storage space is split into inode table and data storage
  - Initial prototypes used static sized table and data storage
- Files are statically allocated
- Need to remember inode number to access file content

- Idea: Use a dedicated place at the beginning of the storage medium, mostly initial blocks on the storage medium
  - A dedicated special file to store a mapping from file names to inodes

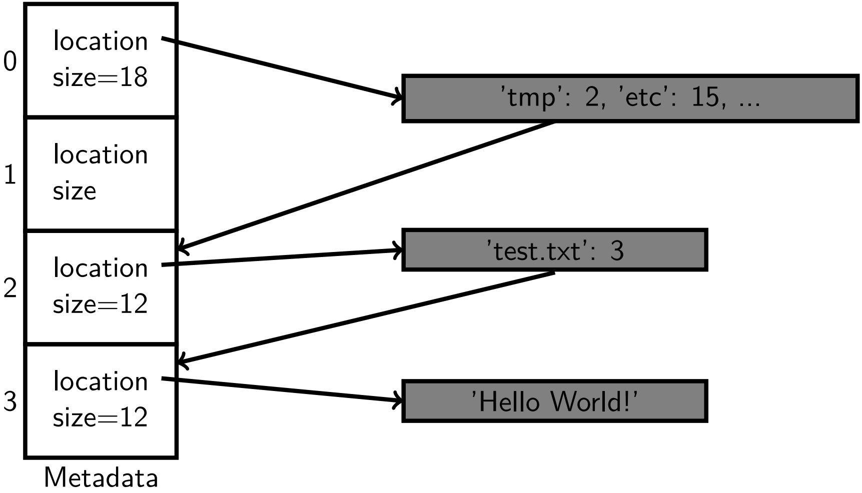
Table of inodes (2/2)

- Storage space is split into inode table and data storage
    - Initial prototypes used static sized table and data storage
- Files are statically allocated
- Need to remember inode number to access file content
- Idea: Use a dedicated place at the beginning of the storage medium, mostly initial blocks on the storage medium
    - A dedicated special file to store a mapping from file names to inodes

The 3 views of a file

- Operating system: Inode and device id
- ***User:*** file name
  - Humans are better at remembering names than numbers
- Process: File descriptor

From path to inode (1/2)



From path to inode (2/2)

- A special file stores mapping between file names and inodes
- Extend to hierarchy: mark if a file name maps to a regular file or a directory
  - Access to '/tmp/test.txt' in 3 steps: 'tmp', 'test.txt', contents
- What data should you store in the directory file (compared to the inode)?

Special directory entries: . and ..

```
$ ls -al
drwxr-xr-x 6 bob bob  4096 Nov 15 13:38 .
drwxr-xr-x 5 bob bob  4096 Nov  1 13:52 ..
-rw-r--r-- 1 bob bob  6830 Nov  1 13:52 00-intro.md
-rw-r--r-- 1 bob bob 16726 Nov  1 13:52 11-proc.md
-rw-r--r-- 1 bob bob 19316 Nov  1 13:52 12-sched.md
-rw-r--r-- 1 bob bob 14048 Nov  1 13:52 13-seg.md
-rw-r--r-- 1 bob bob 17495 Nov  1 13:52 14-pg.md
-rw-r--r-- 1 bob bob  3603 Nov  1 13:52 15-virt.md
```

- ‘.” maps to the current, ‘..” maps to the next higher directory

The 3 views of a file

- Operating system: Inode and device id
- User: file name
- **Process:** File descriptor
  - Keeps track of per-process state (e.g., read position or name, inode mapping)

File descriptor (1/4)

- The combination of file names and inode/device id are sufficient to implement persistent storage
- Drawback: constant lookups from file name to inode/device id are costly
- Idea: do expensive tree traversal once, store final inode/device number in a per-process table
  - Also keep additional information such as file offset
  - Per process table of open files
  - Use linear numbers (fd 0, 1, 2, . . .), reuse when freed

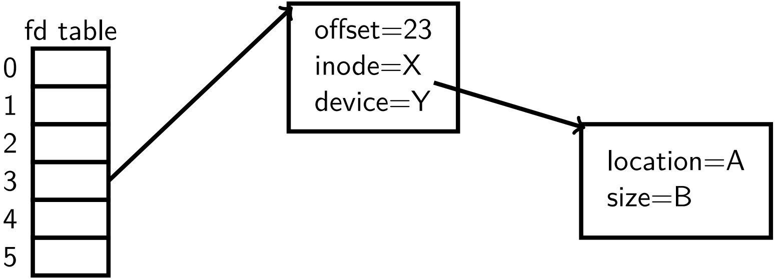
File descriptor (1/4)

- The combination of file names and inode/device id are sufficient to implement persistent storage
- Drawback: constant lookups from file name to inode/device id are costly
- Idea: do expensive tree traversal once, store final inode/device number in a per-process table
  - Also keep additional information such as file offset
  - Per process table of open files
  - Use linear numbers (fd 0, 1, 2, . . .), reuse when freed

File descriptor (1/4)

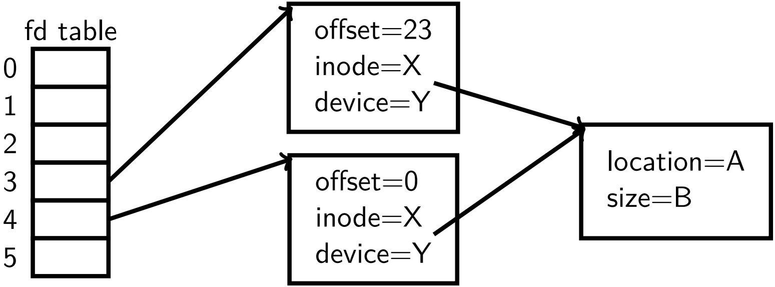
- The combination of file names and inode/device id are sufficient to implement persistent storage
- Drawback: constant lookups from file name to inode/device id are costly
- Idea: do expensive tree traversal once, store final inode/device number in a per-process table
  - Also keep additional information such as file offset
  - Per process table of open files
  - Use linear numbers (fd 0, 1, 2, . . .), reuse when freed

File descriptor (2/4)



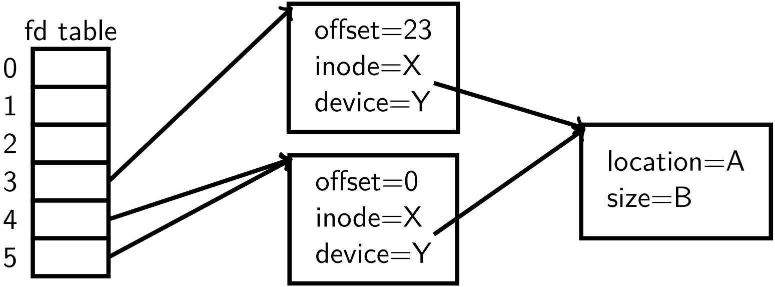
```
int fd1 = open("file.txt");  // returns 3
read(fd1, buf, 23);
```

File descriptor (3/4)



```
int fd1 = open("file.txt");  // returns 3
read(fd1, buf, 23);
int fd2 = open("file.txt");  // returns 4
```

File descriptor (4/4)



```
int fd1 = open("file.txt");  // returns 3
read(fd1, buf, 23);
int fd2 = open("file.txt");  // returns 4
int fd3 = dup(fd2);          // returns 5
```

File API

- `int open(char *path, int flag, mode_t mode)`
- `size_t read(int fd, char *buf, size_t nbyte)`
- `size_t write(int fd, char *buf, size_t nbyte)`
- `int close(int fd)`

Open translates a string name to an inode. OS allocates a file descriptor that points to that inode and returns the file descriptor table index. The path is only traversed once, the OS can cache inodes and each process keeps track of its open files.

File API: deletion

- There is no system call to delete files!
- Inodes are marked free if there are no more references to them (that's why they have a reference count)
- `unlink()` removes a file from a directory and reduces the reference count
- File descriptors are freed upon `close()` or when the process exits

Note, some programs create a temporary file, keep the file descriptor but unlink the file from the directory right after creation. This results in a private temporary file that is recycled when the process exits.

Sharing and concurrency is hard!

- Consider file permissions change *after* file is opened
- Consider a file is moved *after* it is opened
- Consider a file is deleted *after* it is opened
- Consider file owner changes *after* it is opened
- A process forks, what happens to open files (e.g., read positions)
- What happens when two processes write to the same file?
- . . .

The curious case of temp files

```
int main(int argc, char* argv[]) {
  int fd = open("test", O_CREAT | O_RDWR, 0600);
  // unlink("test"); <- what happens to fd here?
  char *data = "test";
  char rdata[64];
  write(fd, data, strlen(data));
  sleep(10);
  lseek(fd, 0, SEEK_SET);
  read(fd, &rdata, 64);
  rdata[63] = 0;
  printf("We read '%s'\n", rdata);
  close(fd);
  return 0;
}
```

Multiple file systems (1/3)

- Challenge: on a single system there are often multiple filesystems
  - Different partitions on the same disk
  - Multiple disks
  - DVD/BlueRay drive
  - USB stick
  - Network Attached Storage
  - Floppy disk
- How do you organize, manage, and display all these file systems?

Multiple file systems: Windows (2/3)

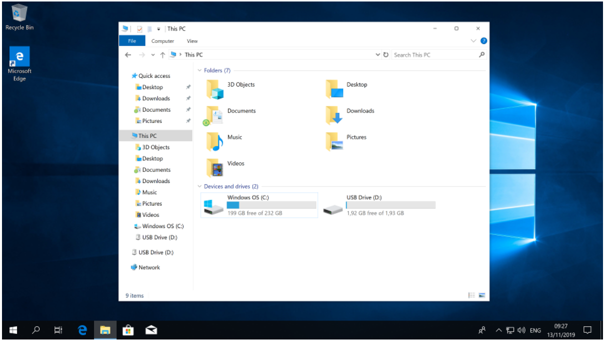
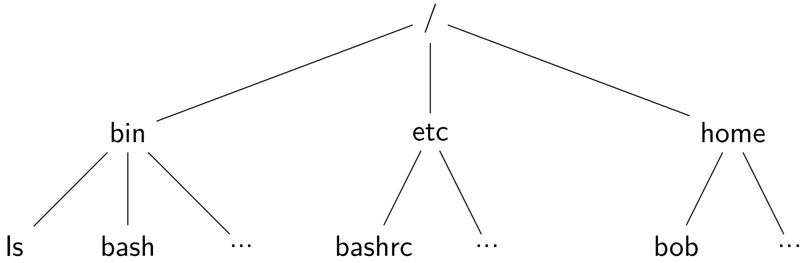


Figure 1: All file systems are mapped to a common root

- Assign a letter to each file system
  - A, B for floppy disks
  - C for main hard drive
  - ...

Multiple file systems: Unix (3/3)



- File systems can be mapped anywhere into *a single tree*
  - Any directory can be a mount point
  - Mounting a FS hides the files in the original directory
  - E.g., "home/* may be a different file system

Hard links and soft links

- Links are file pointers, i.e., they do not contain data themselves but reference another file
- Soft link: a directory entry points to a file that contains a file name, the OS resolves the file name when it is accessed
- Hard link: a directory that points to an existing file, increasing the reference counter
  - This is why the inode does not store the file name

Summary

- Filesystem API: handle interaction with the file system
- Internal and external interface
    - Internal: data structures handle large chunk of blocks
    - External: standardized interface
- Three ways to identify a file
    - File names (for humans)
    - Inodes and devices (on the disk)
    - File descriptors (for a process)
- Combine multiple file systems
    - Mount at the root (Windows)
    - Mount anywhere in the tree (Unix)