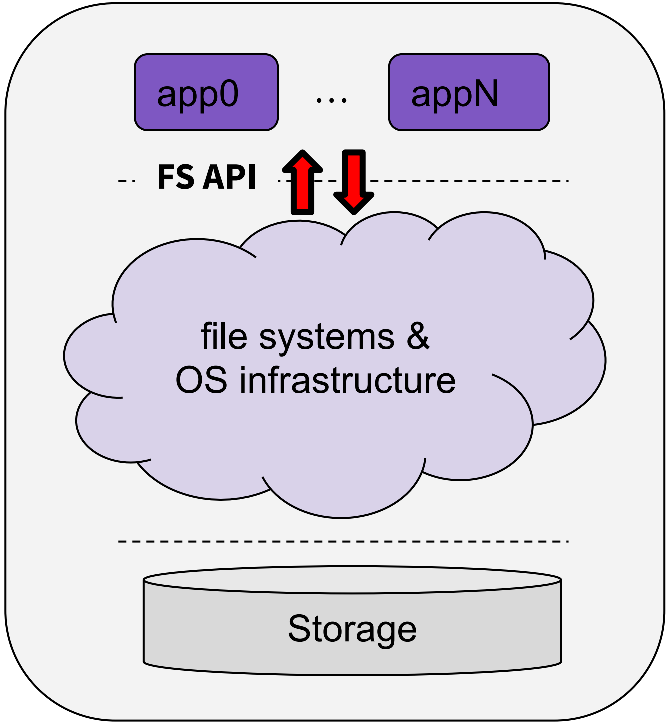
File System API

CS333 :: Storage Systems

Williams College

File Systems are Mediators

- An FS is a part of the operating system that mediates access to storage and implements the file abstraction for applications
- The File System API provides a standard way to:
  - Manage identifiers & namespaces
  - Enforce permissions
  - Access and modify contents
  - Express guarantees for specific behaviors



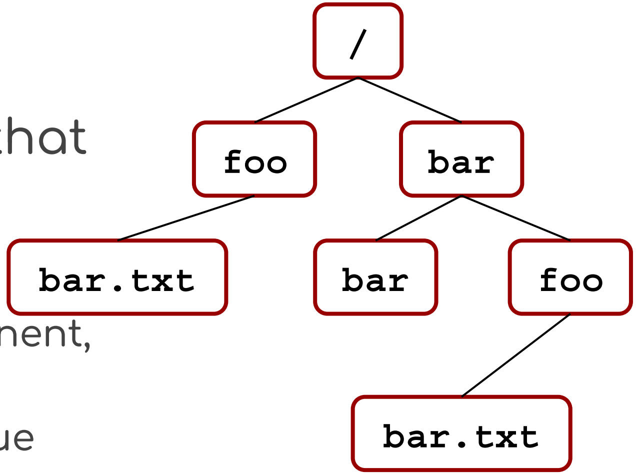
"File" is an Overloaded Term

- As a colloquial term:
    - A file is some "unit" of persistent data that we can refer to by name
- As an abstraction:
    - A file is "*a linear array of bytes, each of which you can read or write*" — (ch. 39)
    - Files are organized into a hierarchy using directories (a type of file), where
        - "Data files" are mutable & byte-addressable
        - "Directory files" are formatted listings of files that form a tree
- As a data structure:
    - Each OS process has its own array of "open files", and the file data structure keeps track of some in-memory state to facilitate interacting with open files
        - current offset (where the next read or write will start)
        - access mode (a *subset* of the legal operations that a process is allowed)

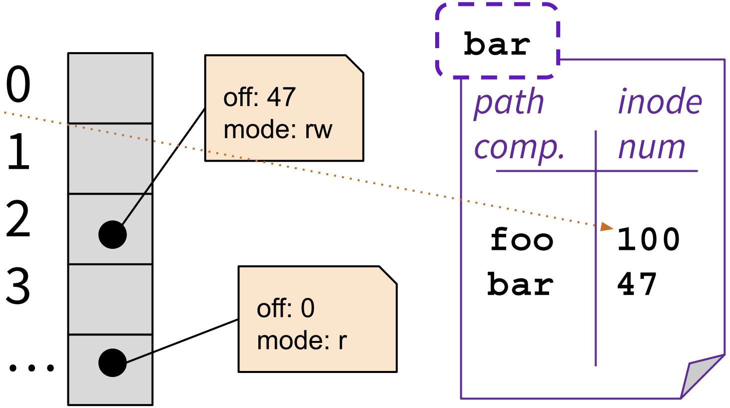
Three Key Identifiers

There are three types of identifiers that describe files, one for each "type"

- Path name (high-level)
    - Concatenation of each path component, separated by '/'
        - Components need not be unique



- Inode number (low-level)
    - Unique object identifier
    - Provides useful "layer of indirection"
- File descriptor index
    - Unique per-process index into open file table
        - Allocated when file is open
        - Recycled when files are closed



Indirection/Index nodes

An inode is a data structure that most closely resembles the idea of "file contents"

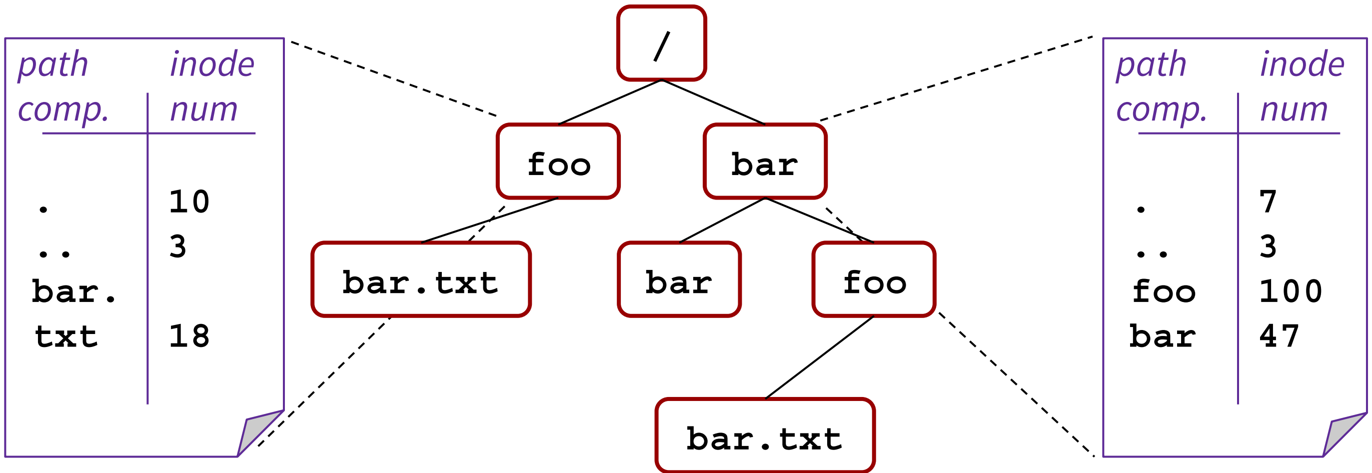
- has a size, permissions, access times, an array of "blocks", etc.

Human-readable path names each refer to an inode, and the OS typically starts FS requests by translating from a human-readable name/high-level (path) to a FS-specific/ow-level name (inode num)

```
$ stat /home/bill/foo.txt
$ File: foo.txt
  Size: 268             Blocks: 8           IO Block: 131072 regular file
  Device: 3ah/58d       Inode: 15007945     Links: 1
  Access: (0640/-rw-r-----)  Uid: (10255/  bill)   Gid: (10255/  bill)
  Access: 2019-09-11 11:09:13.986065000 -0400
  Modify: 2019-09-11 11:16:29.113886000 -0400
  Change: 2019-09-11 11:16:29.113886000 -0400
```

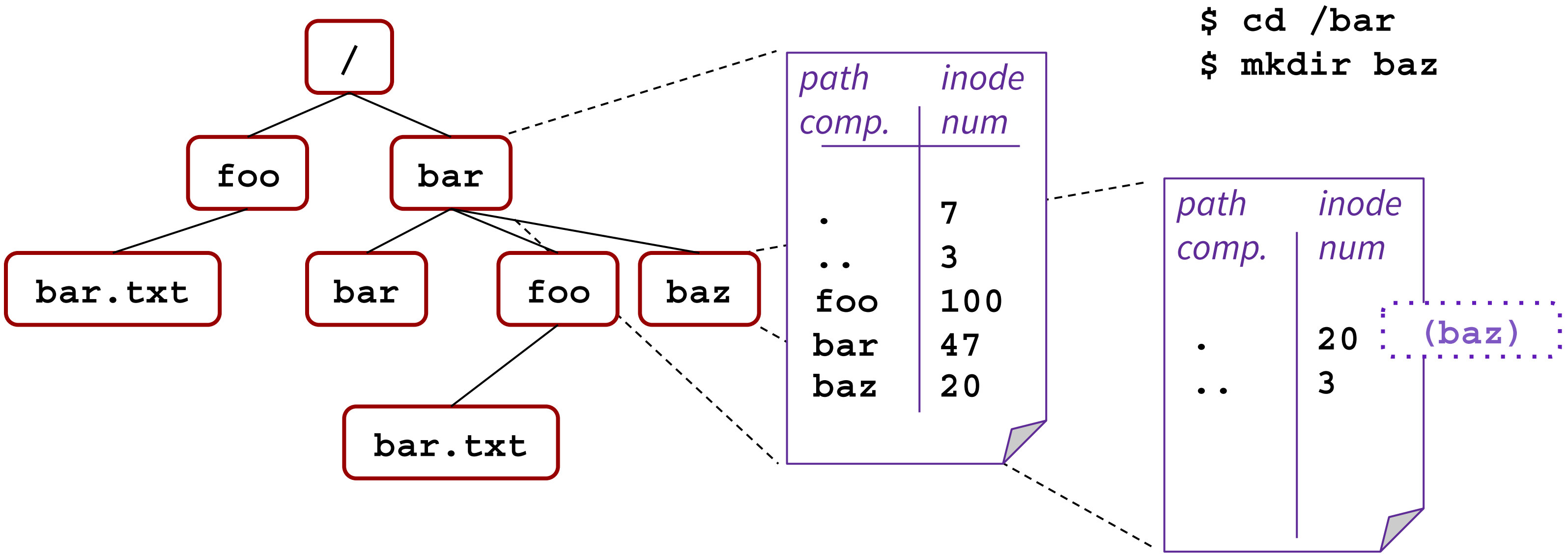
The Namespace Hierarchy

- Our files form a tree, rooted at '/'
  - (We will not get into how we initialize an "empty file system" yet)
- Directories are "special" files in the sense that they have a particular structure and set of directory-specific operations
  - directories contain a listing of children
  - for each child pathname, they store its associated inode number



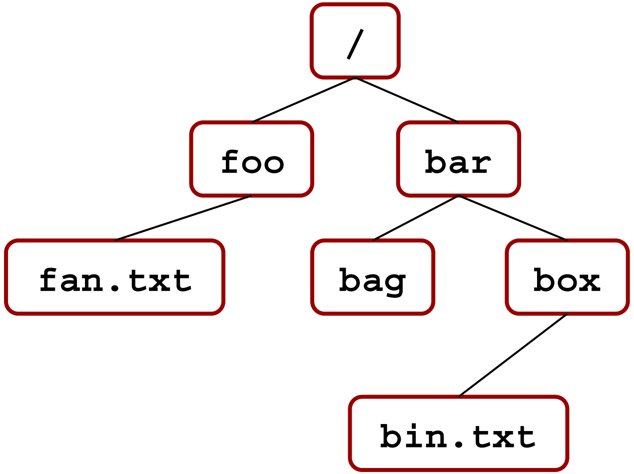
Directories

- To create a directory, we use **mkdir**, which:
  - Creates a new directory file that contains only '.' (this dir) and '..' (parent dir)
    - we use these "dot" entries to navigate up and down the tree
  - Modifies parent directory with new entry's (path, inode num) pair



Directories

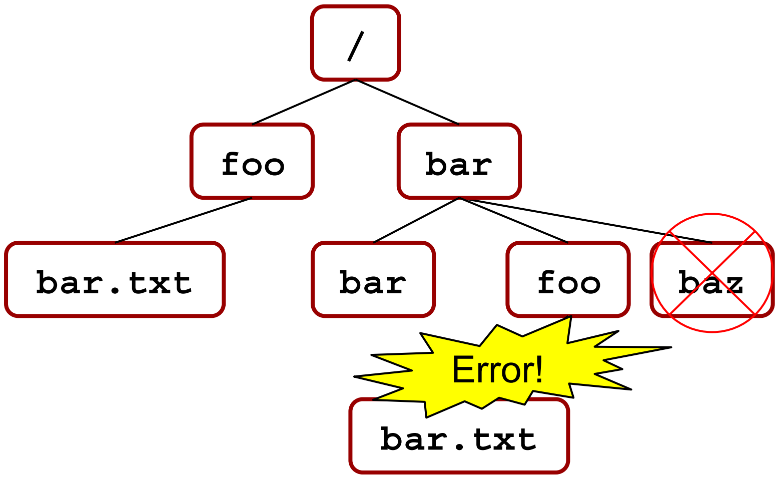
- To list a directory's contents, we use **`readdir`**
  - walks through items in directory file, and for each child's (path, inum) pair, emits details (populates a "**`struct dirent`**")




```
struct dirent {
 char d_name[256]; // filename
 ino_t d_ino; // inode number
 off_t d_off; // offset to the next dirent
 unsigned short d_reclen; // length of this record
 unsigned char d_type; // type of file
};
```

Directories

- To delete a directory contents, we use **`rmdir`**
  - Directory must be empty other than . and ..




```
$ cd /bar
$ rmdir foo
  rmdir: failed to remove 'foo':
  Directory not empty

$ rmdir baz  # succeeds!
```

Processes Open Files

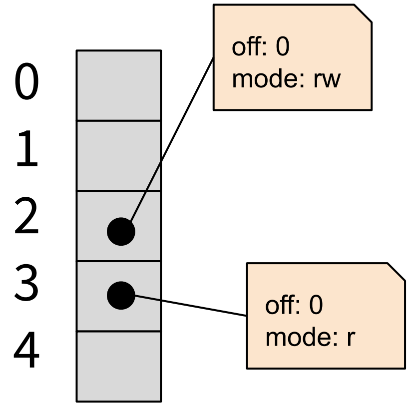
- `open` gives us a way for processes to interact with files
  - In its *simplest form*, `open` takes two arguments: a path name and flags
    - The path is traversed to find its associated inode
    - The flags specify an access mode (O_RDONLY, O_WRONLY, or O_RDWR)

- The file system checks the permissions of the inode*, and if the flags are a subset of the legal capabilities, then open returns a file descriptor
  - (A file descriptor is an index into the processes open file table, see next slide)

File Descriptor Table

Each process has a table that tracks its open files

- File descriptors are integer table indexes
- Table entries store information about a process's interaction with a file. Importantly:
    - The access mode for this particular interaction
    - The current offset
- Thus, the same file can be opened multiple times, with reading and writing happening at different offsets




```
int fdA = open("foo", O_RDRW);
int fdB = open("foo", O_RDONLY);
```

(`fdA` == 2, `fdB` == 3 in above example)

Accessing/Modifying Data

- On success, the **`read`** and **`write`** system calls advance the offset associated with a specific file descriptor
  - Done for convenience, so next read/write picks up where the last left off
- You can also advance a file descriptor's offset manually using **`lseek`**

```
int fd = open("foo", O_RDRW); // offset is at 0
read(fd, buf, 100); // advances offset to 100
read(fd, buf, 100); // advances offset to 200
lseek(fd, 32, SEEK_SET); // offset is set to 32
```

IMPORTANT: functions can fail (or partly fail). We need to check all return values and understand the different modes of success/failure

Accessing/Modifying Data

- The **pread** and **pwrite** system calls DO NOT advance the offset associated with a specific file descriptor
  - Instead, must provide starting offset as part of function call

```
int fd = open("foo", O_RDRW); // offset is at 0
read(fd, buf, 100); // advances offset to 100
read(fd, buf, 100); // advances offset to 200
lseek(fd, 32, SEEK_SET); // offset is set to 32

int fd2 = open("foo", O_RDRW); // offset is at 0
pread(fd, buf, 100, 0); // reads 100 bytes from offset 0
pread(fd, buf, 100, 100); // reads 100 bytes from offset 100
// offset was never changed, so it is still 0
lseek(fd, 32, SEEK_SET); // offset is set to 32
```

Paths + Inodes = Indirection

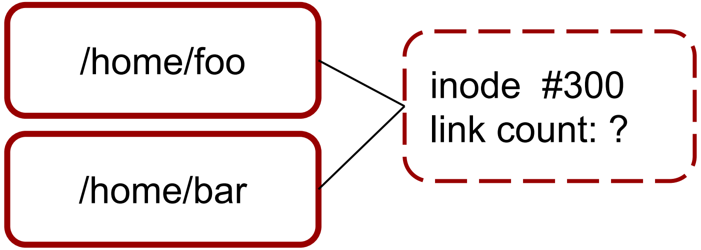
High-level and low-level names provide a "layer of indirection"

- Each path name "points" to one inode, but not necessarily 1-to-1

"Pointing" multiple paths to the same inode can be done with **`link`**

- Changes made using one path are reflected in the other since they share the same underlying inode (low-level name)
- Deleting (**`unlink`**) one pathname does not alter the other

```
creat("/home/foo", S_IRUSR|S_IWUSR); //link count=1
link("/home/foo", "/home/bar"); //link count=2
unlink("/home/foo"); //link count=1
```



Links - Hard and Soft

We can also use indirection at the "path" level rather than the "inode"

A symbolic link is a special type of file that has contains a path that is "resolved" instead:

- When I ask to open a symbolic link, the FS instead tries to open the path stored inside the symbolic link file
    - If that path doesn't actually exist, this is called a "dangling link"
- Note: we can create infinite loops if we're not careful (or not nice)
    - The OS prevents this by giving up after a fixed number of tries

```
bill@unix:~-> ls -l /bin/python3
lrwxrwxrwx 1 root root  /bin/python3 -> python3.8*
```

Indirection Gives Fast Renames

The **`rename`** system call removes a pathname from the directory hierarchy at one location and inserts it into the directory hierarchy at another

- Renaming a file is atomic
  - Either the rename happens completely or it doesn't happen at all
  - Intermediate state is never visible: the file always exists at exactly one location
- Renaming a directory moves all children with it
  - This is is possible because directories map path components to inode numbers rather than absolute paths to inode numbers

Rename is a source of considerable trickery and abuse. Many applications take advantage of its atomicity.

Caching improves Performance

File are byte-addressable, but our media often is not

- Caching can help!
  - When we write data, the FS may cache it in RAM so that future writes can be aggregated in a single I/O
  - We don't have to worry about caching—it happens automatically

Are there downsides to caching? Maybe...

- Potential downside 1: Duplication of work
  - What if applications already do their own caching?
    - open the file with the O_DIRECT flat
- Potential downside 2: Data loss on a crash
  - We need a way to tell the FS we *really* need our data to be written
    - The `fsync` function tells the FS to write all uncommitted data immediately

Important FS API functions

Covered in this video

- open/creat
- close
- read/pread
- write/pwrite
- fsync
- rename
- link/unlink
- mkdir
- readdir
- rmdir
- lseek

Not covered but important:

- stat/fstat
- mount/unmount