/../15/410

File System (Interface)
Mar. 29, 2006

Greg Hartman

Dave Eckhardt

Bruce Maggs

Contributions from Rahul Iyer

Synchronization

Today

- Chapter 10, File system interface
  - Ok to skip: remote/distributed (10.5.2!!)

Synchronization

Two interesting papers about disks

- http://www.seagate.com/content/docs/pdf/whitepaper/D2c_More_than_Interface_ATA_vs_SCSI_042003.pdf
- Google for “200 ways to revive a hard drive”

Disks aren’t enough

Users want to:

- Store multiple files on a disk
  - Disks have a global list of blocks
- Protect files from unauthorized access
  - Disks allow access to any block
- Retrieve these files quickly
  - Disks only perform well with certain access patterns
- Reference the files with sensible names and group files
  - Have blocks referenced by number

Disks do provide blocks

Lots and lots of inexpensive blocks

- The going price is about 61,035,156 / $95.00
  - Assumes 4096 bytes/block. Taxes, shipping, and handling extra

Filesystems steal some of the blocks

- We call this “storing metadata”
- Consumes about 7% of the blocks

to give us what we really want from disks

Types of filesystem metadata

Mapping between files and blocks

Allows multiple files on a disk

Access control lists

Protects files from unauthorized access

Freespace lists

Allows files to grow and shrink, and be recycled

Directories

Provide naming and grouping of files

Additional metadata

Per file

- Identifier - “file number” aka inode
- Type (or not)
- Location – device, block list
- Size – real or otherwise
- Time, date, last modifier – monitoring, curiosity

Per filesystem

- Quotas: space available/consumed per user

“Extended” file attributes

BSD Unix

- archived
- nodump
- append-only (by user/by operating system)
- immutable (by user/by operating system)

MacOS

- Application that created the file (Creator)

Plan 9

- Identity of most recent mutator

Metadata

Use of metadata takes *time*

String lookups take a long time

Most operations in the interface read or write metadata

Solution: memoization

- Do the hard work once
- Save the answer in a table
- Refer to the answer with a convenient handle when needed
- Provide a way to free the answer when done

Memos applied to filesystems

Expensive string lookups happen in `open()`

- Saves a lot of stuff in the file descriptor table:
  - File-system / partition
  - File-system-relative file number
  - Read vs. write
  - Cursor position
  - In memory copies of frequently accessed metadata

Open returns the index for the file in the table

`close()` releases the data from the table

Problem: maintaining consistency

Consider three unrelated processes:

- Process A opens F read-only
- Processes B and C open F read-write

The filesystem should enforce:

- A can't write to F
- A can see updates made by B
- Simultaneous changes to the file should not corrupt state

Solution: split memos into shared and semi-shared parts

Shared state (Unix Model)

Mirror of on-disk structure

- File number, size, permissions, modification time, ...

Housekeeping info

- Back pointer to enclosing file system
- Pointer to disk device hosting the file
- Who holds locks on ranges of file

How to access file (vector of methods)

Semi-shared state (Unix Model)

Shared by *related* processes

- **“copied” by fork() and inherited across exec()**

Access mode (read vs. write, auto-append, ...)

Credentials of process (when it opened the file)

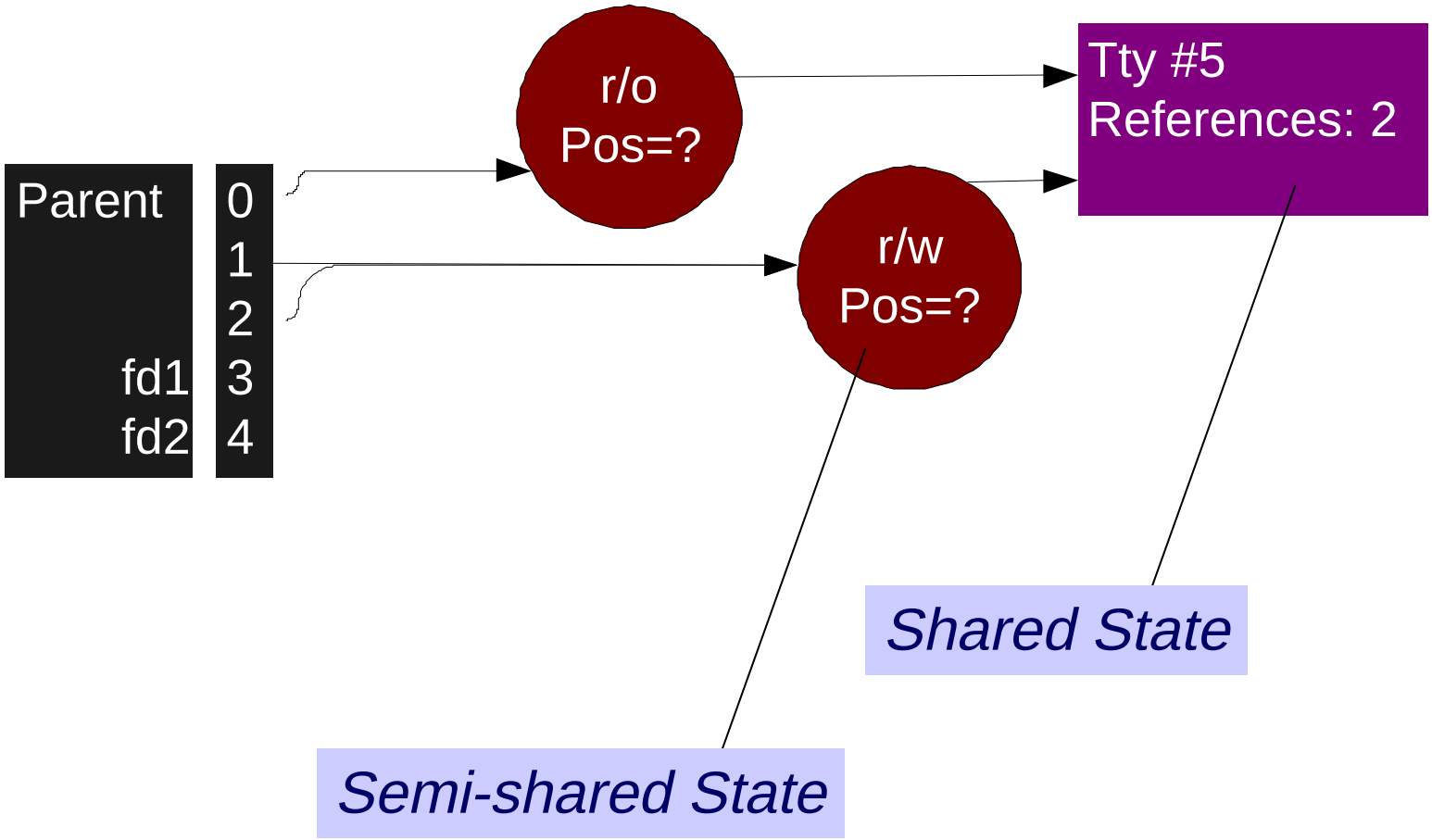
Cursor position

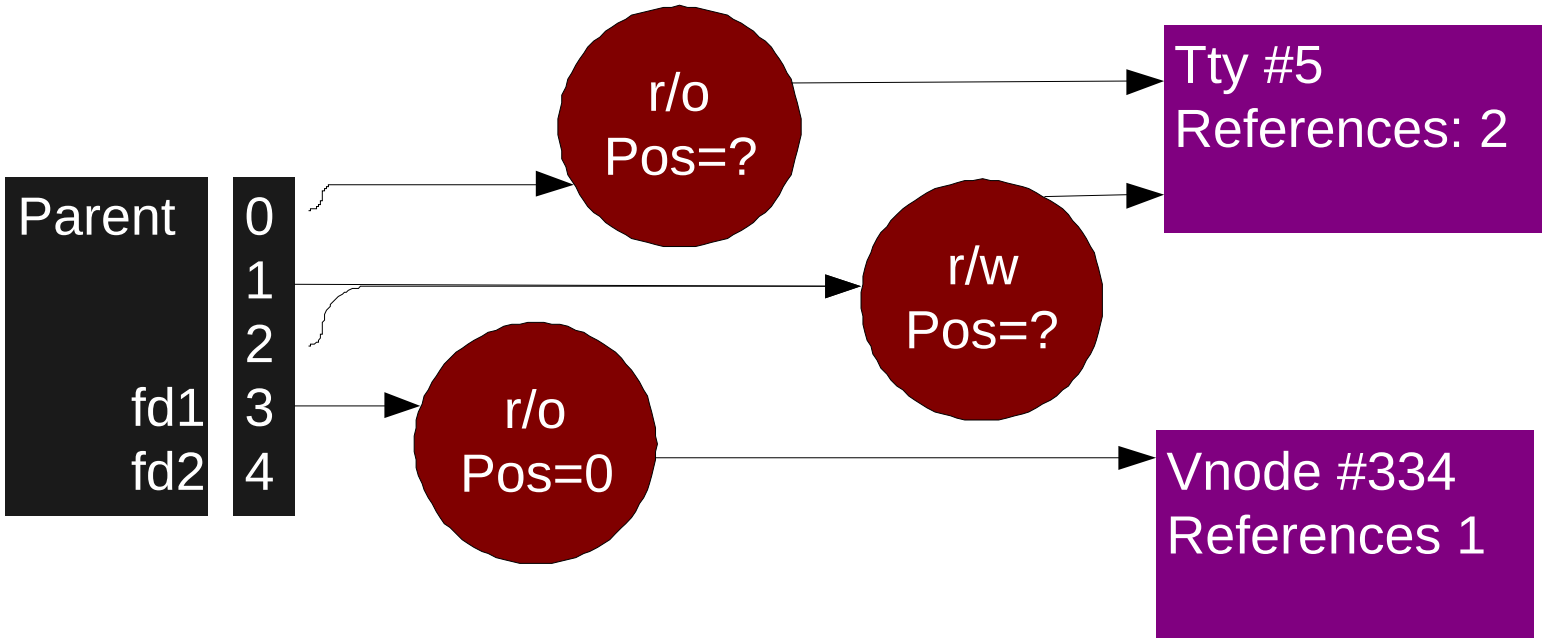
Pointer to the shared state

Example

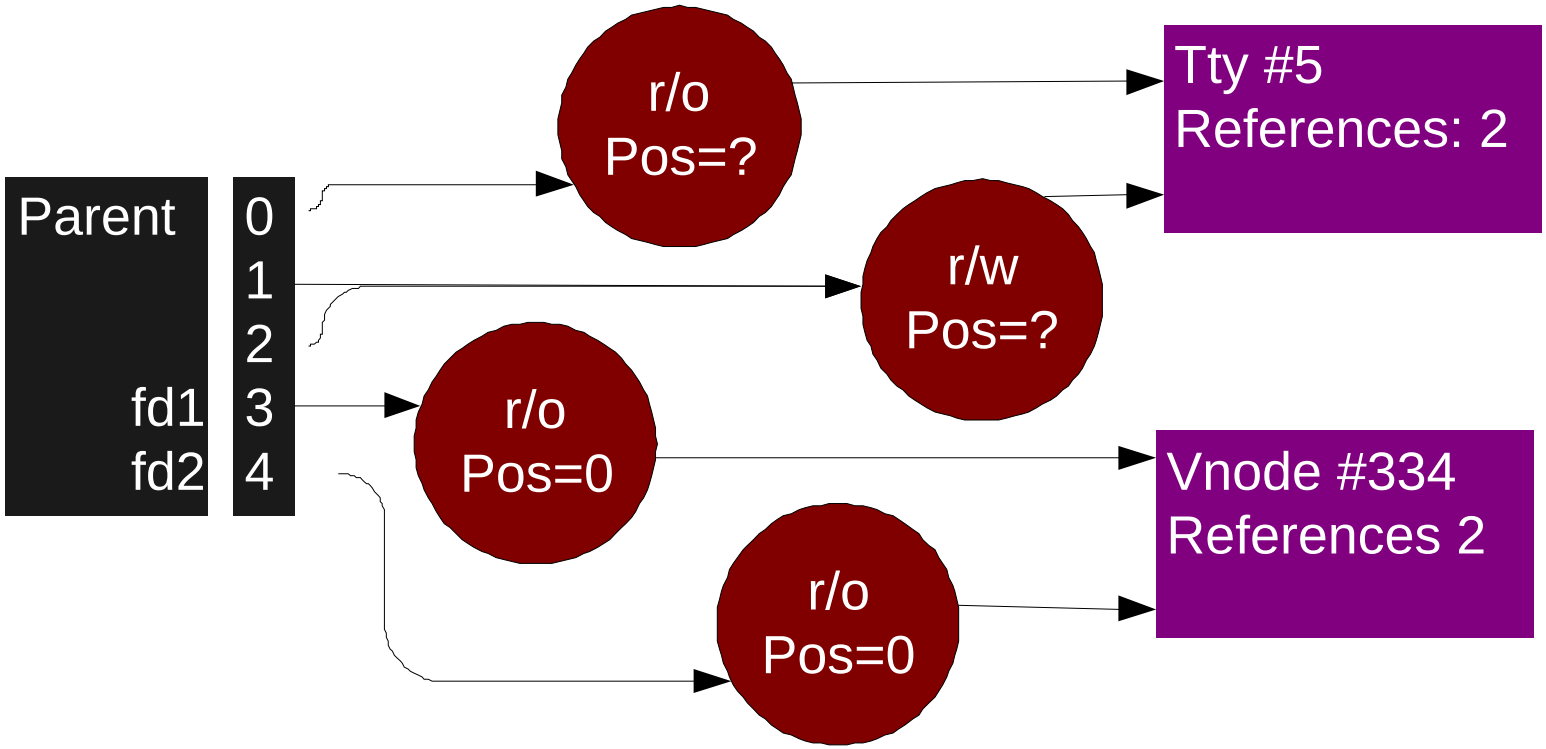
```
int fd1,fd2,child;
off_t pos1, pos2;
char buf[10];

fd1 = open(“foo.c”, O_RDONLY, 0);
fd2 = open(“foo.c”, O_RDONLY, 0);
if (!(child=fork())) {
  read(fd1, &buf, sizeof (buf));
  exit(0);
} else {
  waitpid(child, NULL, 0);
  pos1 = lseek(fd1, 0L, SEEK_CUR);/*10*/
  pos2 = lseek(fd2, 0L, SEEK_CUR);/*0*/
```

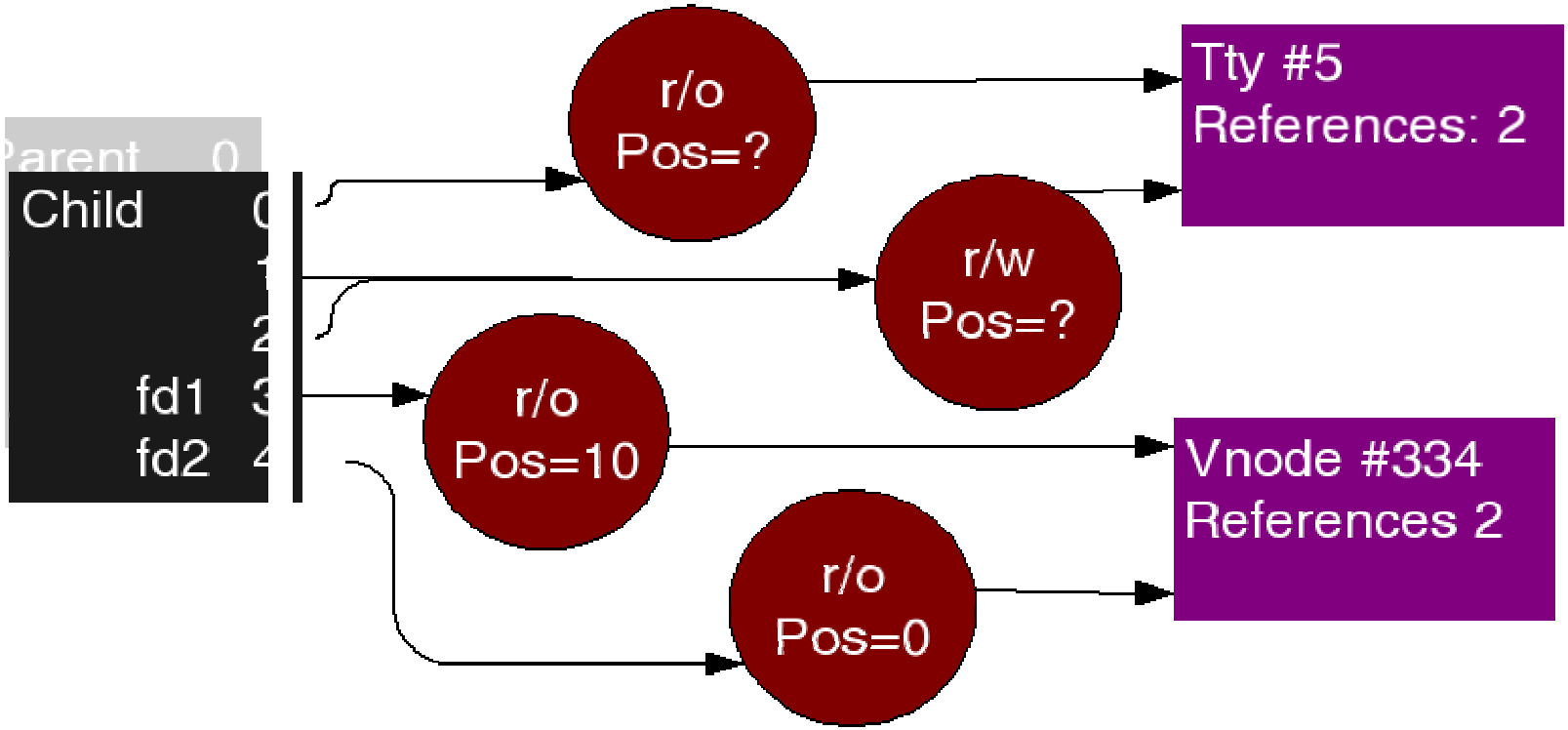


Parent
fd1
fd2
0
1
2
3
4
r/o
Pos=?
r/w
Pos=?
Tty #5
References: 2
Shared State
Semi-shared State



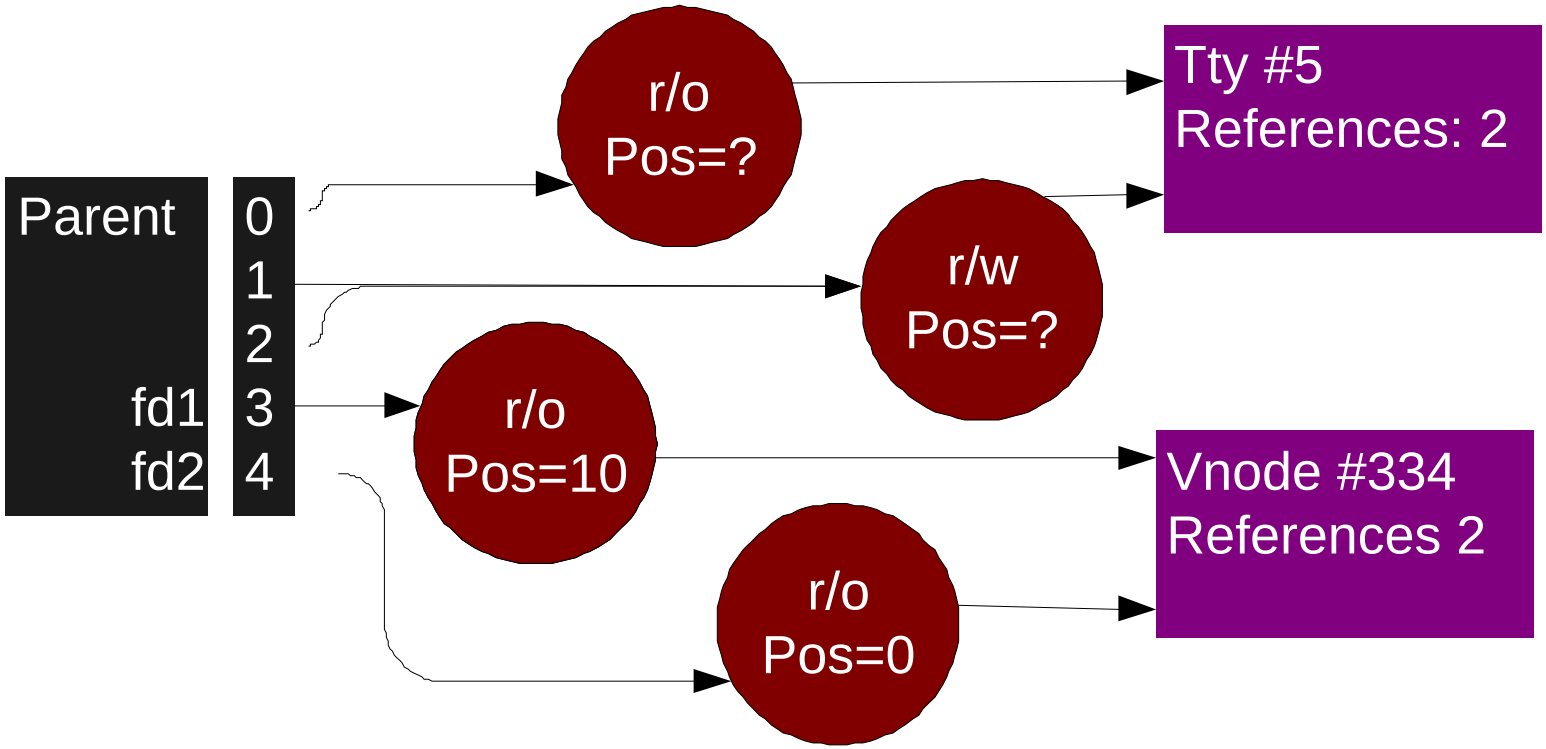

```
fd1 = open("foo.c", O_RDONLY, 0);
```




```
fd2 = open("foo.c", O_RDONLY, 0);
fork(); waitpid(child, NULL, 0);
```




```
read(fd1, &buf, sizeof (buf));
exit(0);
```




```
pos1 = lseek(fd1, 0L, SEEK_CUR);/*10*/
pos2 = lseek(fd2, 0L, SEEK_CUR);/*0*/
```

File types (or not)

Goal

- Avoid printing a binary executable file
- Find program which “understands” a file selected by user

Derive “type” from file names

- *.exe are executable, *.c are C

Store type in metadata

- MacOS: 4-byte type, 4-byte creator

Unix: file name/neither – Leave it (mostly) up to users

File Structure

What's in a file?

- Stream of bytes?
- Text?
  - What character set? US-ASCII? Roman-1? Unicode?
- Records?

Record structure?

- Fixed-length? Varying? Bounded?

File Structure - Unix

Program loader needs to know about executables

- "Magic numbers" in first two bytes
  - obsolete A.OUT types - OMAGIC, NMAGIC, ZMAGIC
  - ELF
  - #! - script

Otherwise, array of bytes

- User/application remembers meaning (hopefully!)

Advantage: easy to create new file formats

Disadvantage: identifying files becomes difficult

- Try the "file" command
- Read /usr/share/magic
  - Marvel at the dedication of the masses

File Structure – MacOS

Data fork

- Array of bytes
- Application-dependent structure

Resource fork

- Table of resources
  - Indexed by type and sequence number
  - For example, Icon #3, Menu #2, Window #3, Dialog box #4
- Many types are widely used & understood
  - Finder displays icons from resource fork

A good compromise between flexibility and structure

Access Methods

Provided by OS or optional program library

Sequential

- Like a tape
- read() next, write() next, rewind()
- Sometimes: skip forward/backward

Direct/relative

- Array of fixed-size records
- Read/write any record, by #

Access Methods – Indexed

File contains records

Records contain keys

Index maps keys ⇒ records

- Sort data portion by key
- Binary search in multi-level list

Fancy extensions

- Multiple keys, multiple indices
- Are we having a database yet?
  - Missing: relations, triggers, consistency, transactions, ...
- Unix equivalent: dbm/ndbm/gdbm/bdb/...

Filesystem Interface (Unix model)

Create – locate space, enter into directory

Write, Read – according to position pointer/cursor

Seek – adjust position pointer

Delete – remove from directory, release space

Truncate

- **Trim data from end**
- **Often all of it**

Append, Rename

Directory Operations

Lookup(inode, “index.html”)

Iterate over directory contents

Create(“index.html”)

Delete(“index.html”)

Rename(“index.html”, “index.html~”)

Scan file system

- **Unix find command**
- **Backup program**

Directory Types

Single-level

- Flat global namespace – only one test.c
- Ok for floppy disks (maybe)

Two-level

- Every user has a directory
- One test.c per user
  - [1003,221]PROFILE.CMD vs. [1207,438]PROFILE.CMD
- Typical of early timesharing

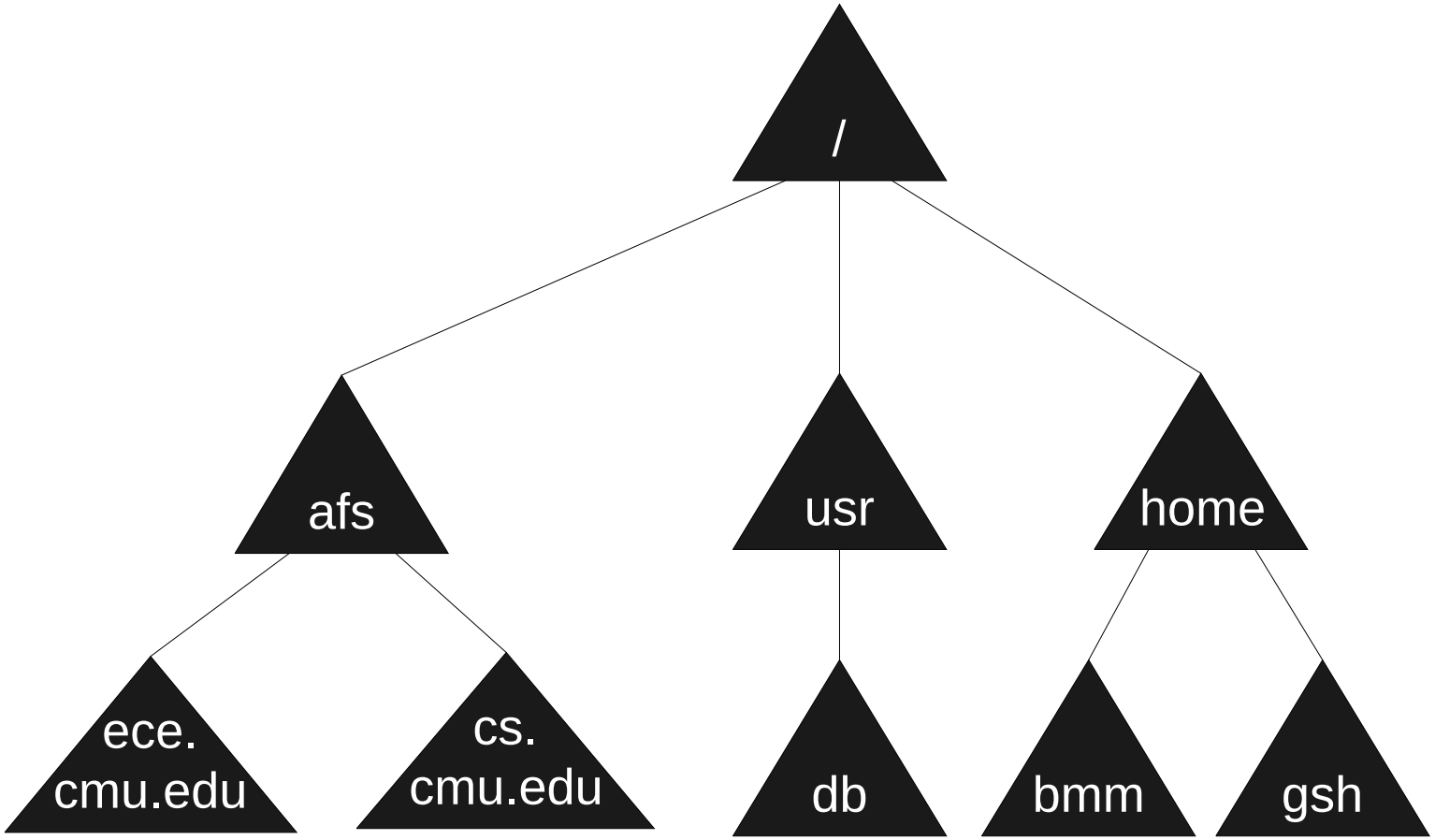
Are we having fun yet?

Tree Directories

Absolute Pathname

- Sequence of directory names
- Starting from “root”
- Ending with a file name

Tree Directories



Tree Directories

Directories are special files

- Created with special system calls – mkdir()
- Format understood, maintained by OS

Current directory (“.”)

- “Where I am now”
- Start of relative pathname
  - ./stuff/foo.c aka stuff/foo.c
  - ../joe/foo.c aka /usr/joe/foo.c
- Directory reference cached in user library or kernel
  - e.g., p->p_fd->fd_cdir

DAG Directories

Share files and directories between users

Not mine, not yours: *ours*

Destroy when *everybody* deletes

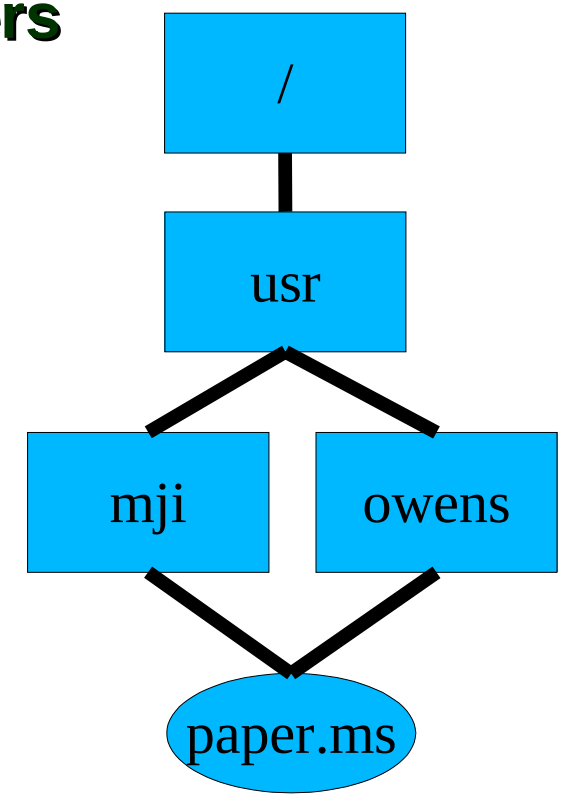
Files with no links exist until closed

Difficult for users to predict behavior

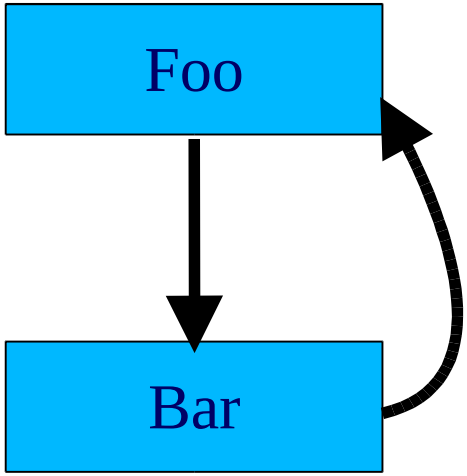
- Unlink and create breaks the link
- Open and truncate preserves it
- Both are reasonable choices

Unix "hard link"

- Files, not directories
  - (".. problem")



The ".. Problem"




```
$ ln .. bar
```

Soft links

Hard links “too hard”?

- Need a level of indirection in file system?
- No “one true name” for a file

Alternative: soft link / symbolic link / “short cut”

- Tiny file, special type
- Contains name of another file
- OS dereferences link when you open() it

Hard vs. Soft Links

Hard links

- Enable reference-counted sharing
- No name is “better” than another

Soft links

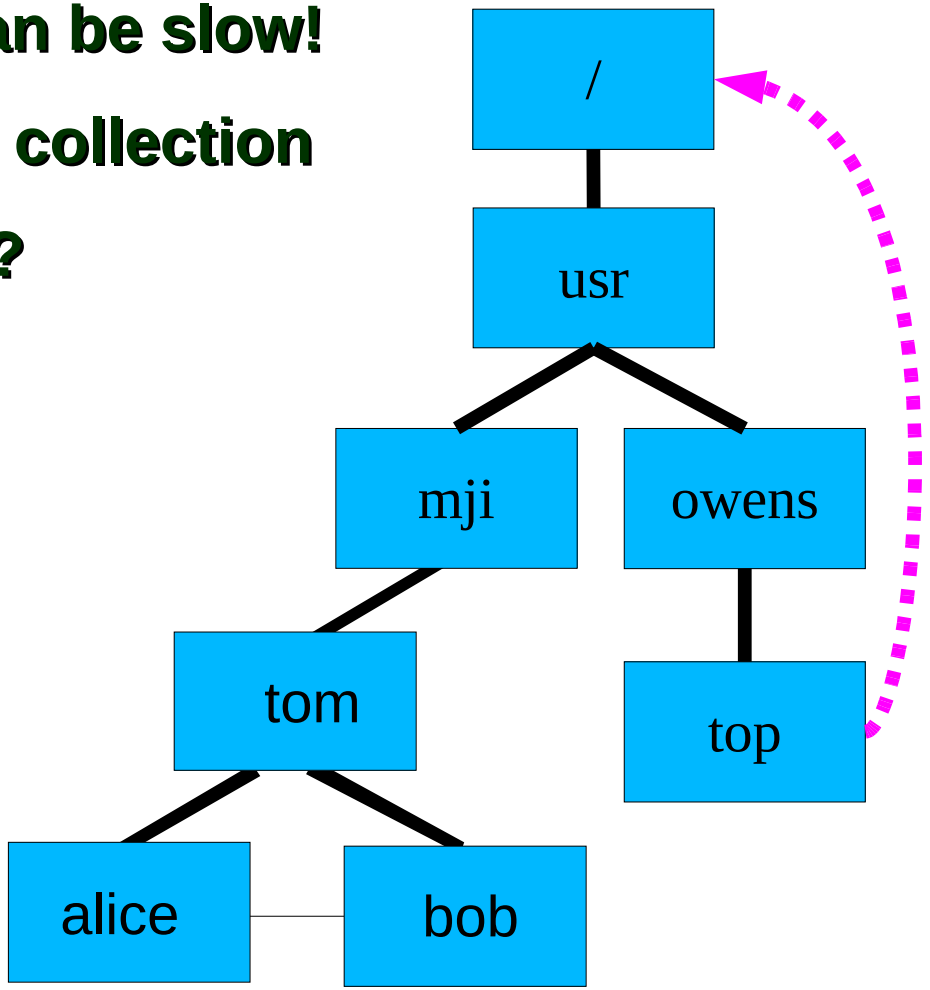
- Can soft-link a directory
  - one “true” parent, so no “.. problem”
- Work across file system & machine boundaries
- Easier to explain
- “Dangling link” problem
  - Owner of “one true file” can delete it
  - Soft links now point to nothing

Graph Directories

Depth-first traversal can be slow!

May need *real* garbage collection

Do we really need this?



Mounting

Multiple disks on machine

Multiple partitions on disk

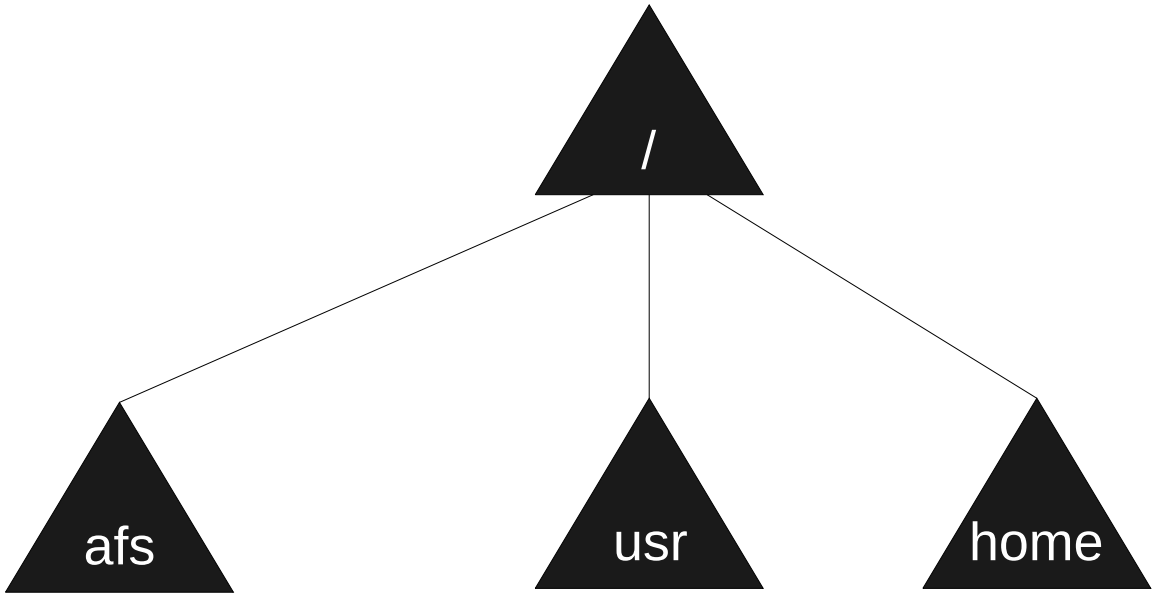
Single file system within a partition

- **Or, within a volume / logical volume / ...**

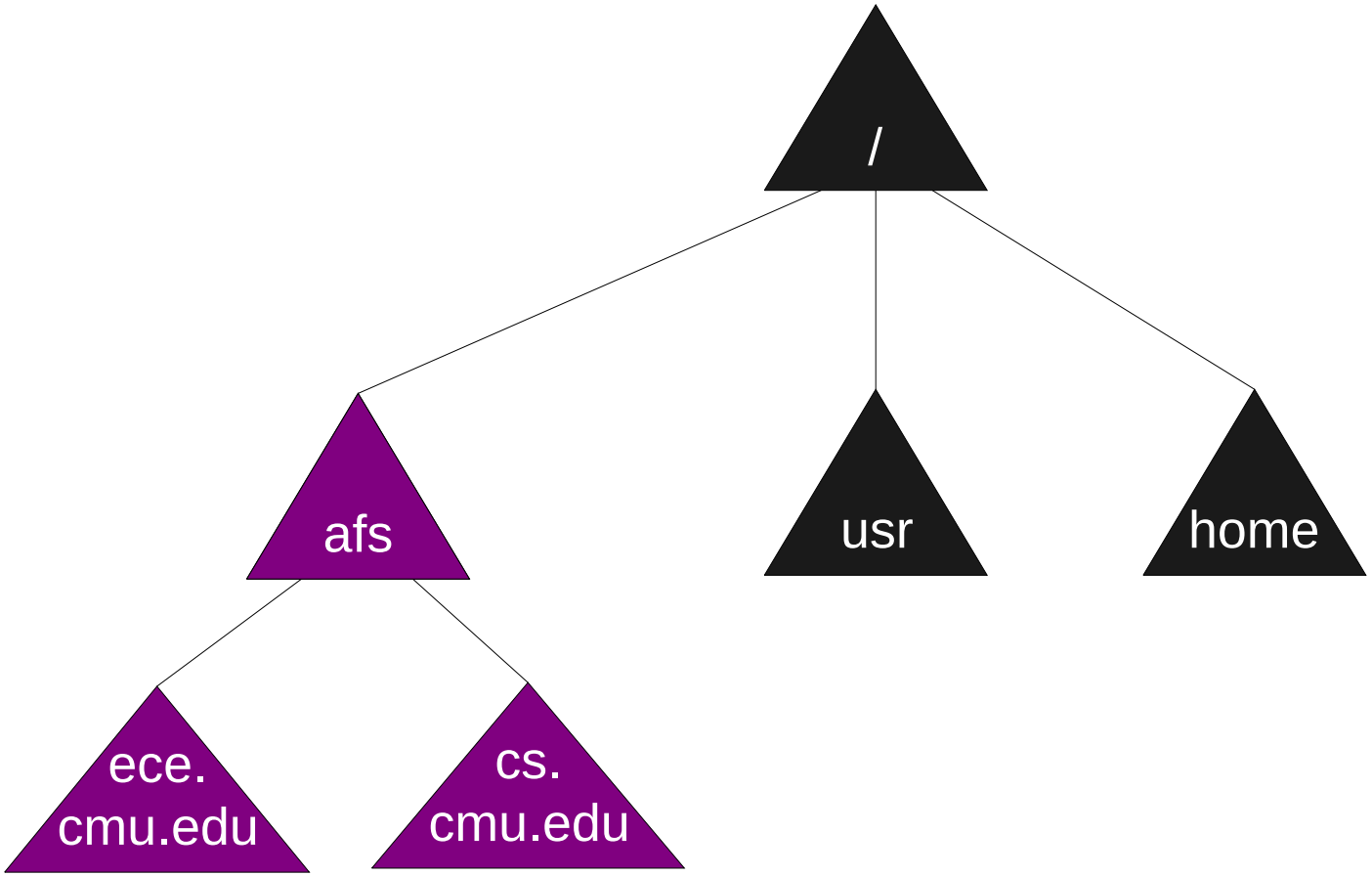
How to name files in “another” file system?

- **Wrong way**
  - **C:\temp vs. D:\temp**
  - **[1003,221]PROFILE.CMD vs. [1207,438]PROFILE.CMD**

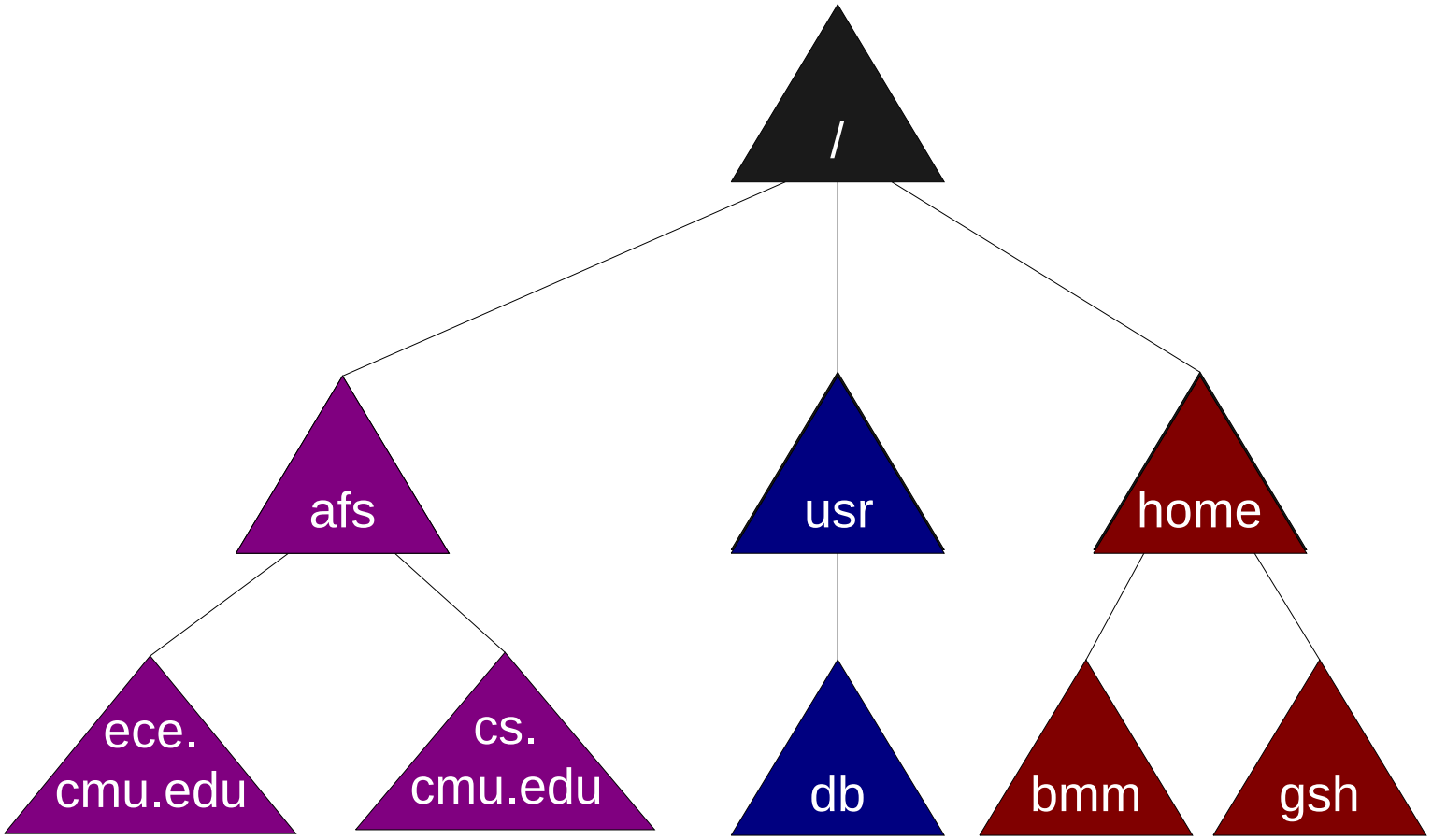
Mounting



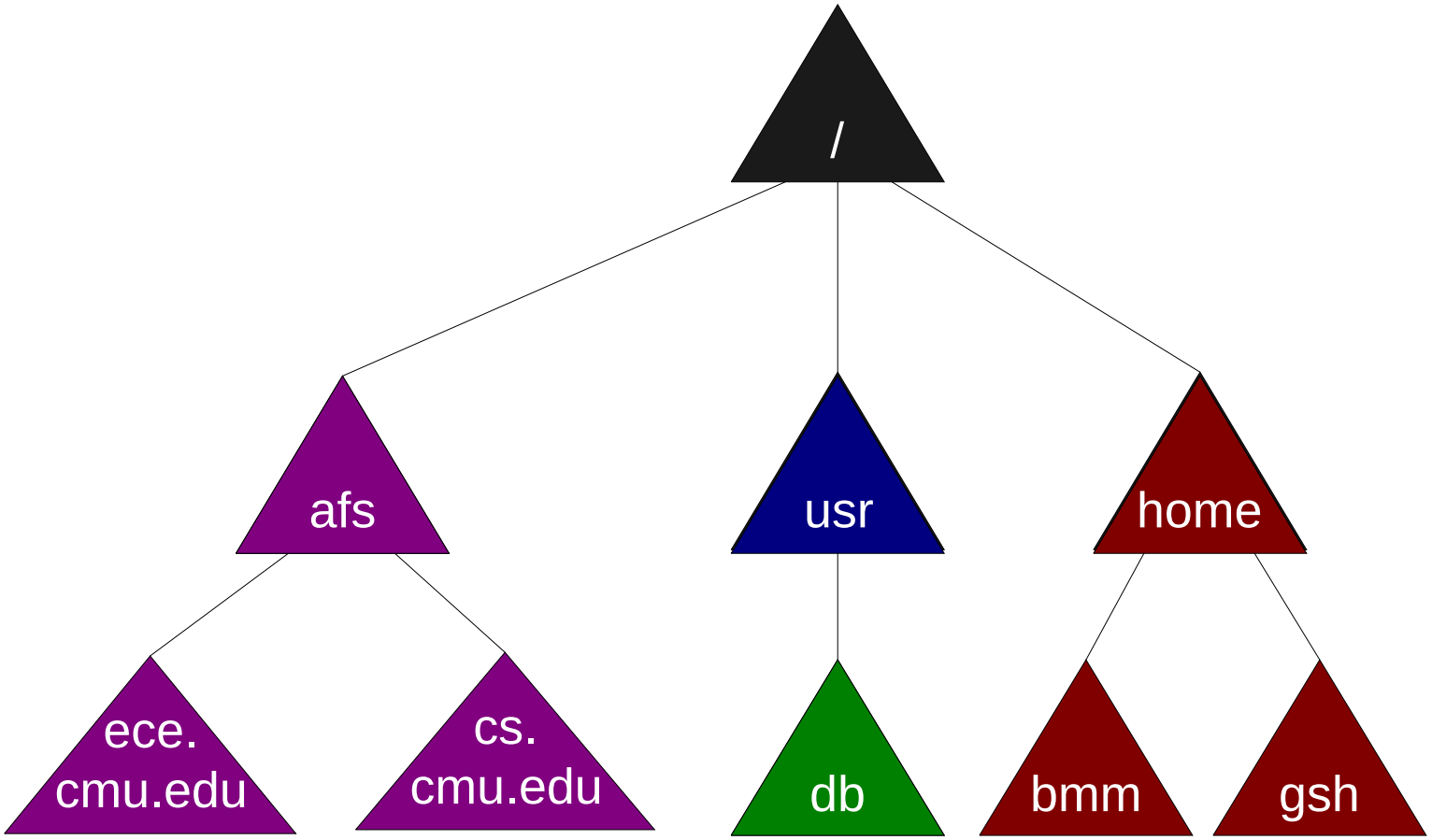
Mounting



Mounting



Mounting



Multiple Users

Users want to share files

What's a user?

- Strings can be cumbersome
- Integers are nicer for OS to compare
- Unix: User ID / “uid”
- Windows: Security ID / “SID”

What's a group?

- A set of users
- Typically has its own gid / SID

Protection

Override “bit” (e.g., MS-DOS)

- Bit says “don't delete this file”
  - Unless I clear the bit

Per-file passwords

- Annoying in a hurry

Per-directory passwords

- Still annoying

Protection

Access modes

- Read, Write, Execute, Append, Delete, List, Lock, ...

Access Control List (ACL)

- File stores list of (user, modes) tuples
- Cumbersome to store, view, manage

Capability system

- User is given a list of (file, access keys) tuples
- Revocation problem

Protection – typical

File specifies owner, group

- Permissions for owner, permissions for group members
  - Read, write, ...
- Permissions for “other” / “world”
  - Read, write, ...

Unix

- r, w, x = 4, 2, 1
- rwxr-x—x = 0751 (octal)
- V7 Unix: 3 16-bit words specified all permission info
  - permission bits, user #, group #
    - » Andrew's /etc/passwd has 32,670 users...

Summary

File

- Abstraction of disk/tape storage
  - Records, not sectors
  - Type information
- Naming
  - Complexity due to linking
- Ownership, permissions
- Semantics of multiple `open()`s

Extra details in 20.7, 20.8