PERSISTENCE: FILE API

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Project 4: Grades and regrades

Midterm 2: Done!

Project 6: Map-Reduce (not xv6)

- Available tomorrow
- Can still request project partner!

PERSISTENCE

How to ensure data is available across reboots

- even after power outages, hardware failure, system crashes?

Topics:

- Persistent storage devices (HDDs, RAID, SSDs)
- File API for processes
- FS implementation (meta-data structures, allocation policies)
- Crash recovery (journaling)
- Advanced Topics: Distributed systems?

RAID LEVEL COMPARISONS

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

RAID-0 is always fastest and has best capacity (but at cost of reliability)

RAID-1 better than RAID-5 for random workloads

RAID-5 better than RAID-1 for sequential workloads

LEARNING OUTCOMES: FILE API

How to **name** files?

- What are **inode numbers**?
- How to **lookup** a file based on pathname?
- What is a **file descriptor**?

What are directories?

What is the difference between **hard and soft links**?

How can **special requirements** be communicated to file system (fsync)?

FILES

WHAT IS A FILE?

Array of persistent bytes that can be read/written

File system consists of many files

Refers to collection of files (file system image)

Also refers to part of OS that manages those files

Many local file systems:ext2, ext3, ext4, xfs, zfs, btrfs, reiserfs, f2fs

Files are common abstraction across all…

Files need names so can access correct one

FILE NAMES

Three types of names

1. Unique id: inode numbers
2. Path
3. File descriptor

1) NAME: INODE NUMBER

Each file has exactly one **inode number**

Inodes are unique (at a given time) within file system

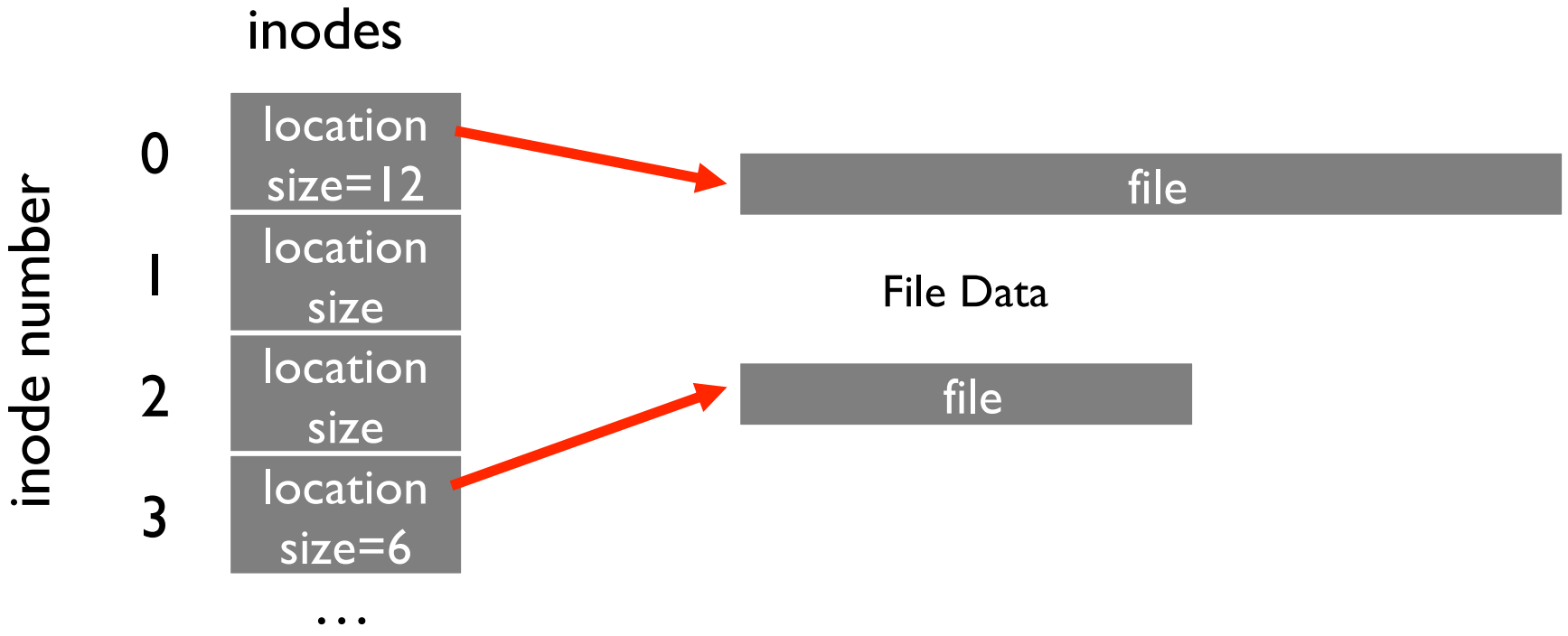
Different file systems may use the same number,
numbers may be recycled after deletes

See inodes via "ls –i"; see them increment…

WHAT DOES “I” STAND FOR?

“In truth, I don't know either. It was just a term that we started to use. ‘Index’ is my best guess, because of the slightly unusual file system structure that stored the access information of files as a flat array on the disk…”

~ Dennis Ritchie



Meta-data: Describes data

Investigate meta-data more in next lecture

Inodes stored in known, fixed block location on disk

Simple math to determine location of particular inode

FILE API (ATTEMPT 1)

```
read(int inode, void *buf, size_t nbyte)
write(int inode, void *buf, size_t nbyte)
seek(int inode, off_t offset)
```

read() and write() track current offset of file to access next
seek() sets offset; does not cause disk seek until read/write performed

Disadvantages?

- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

2) PATHS

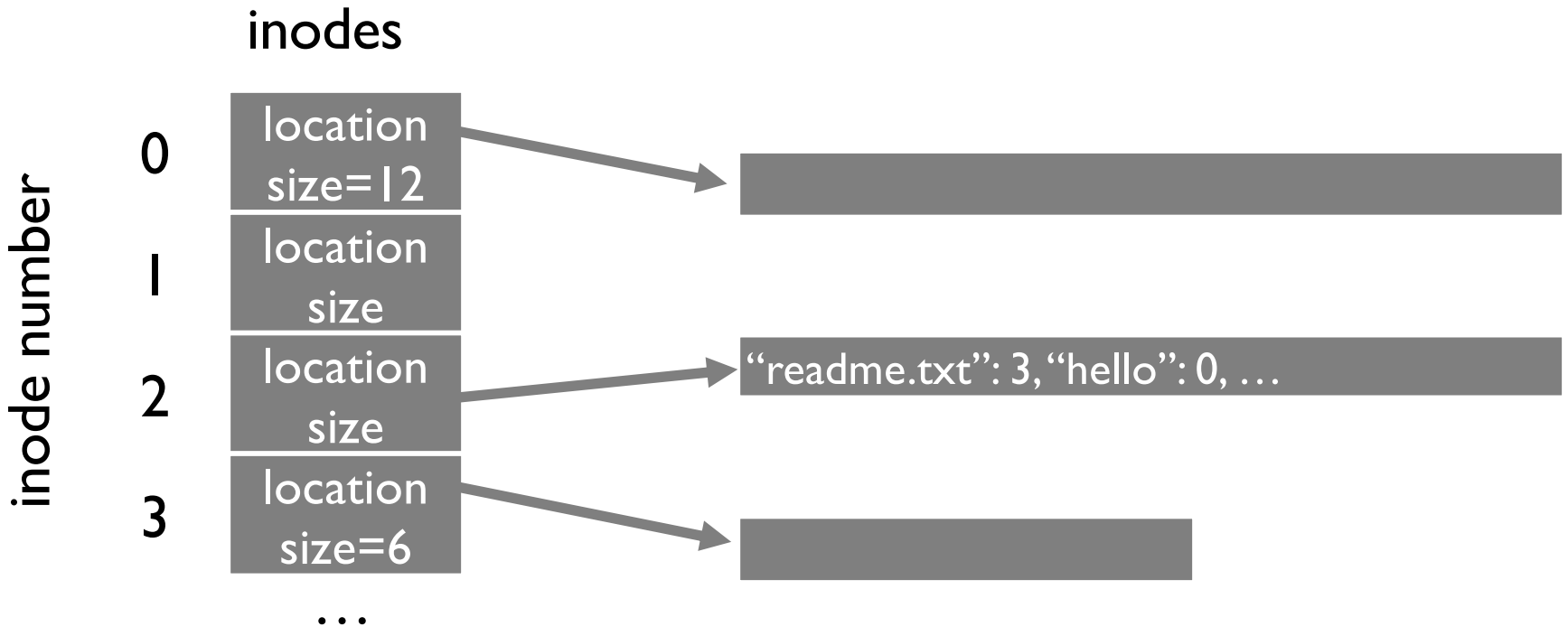
String names are friendlier than number names

File system still interacts with inode numbers

Store *path-to-inode* mappings in a special file; what is that special file?

Directory!

Start with a single directory, stored in known location (typically inode 2)



What should inode number 2 point to?

What is the name of the file stored with inode 0? File with inode 3?

2) PATHS

Generalize to multiple directories…

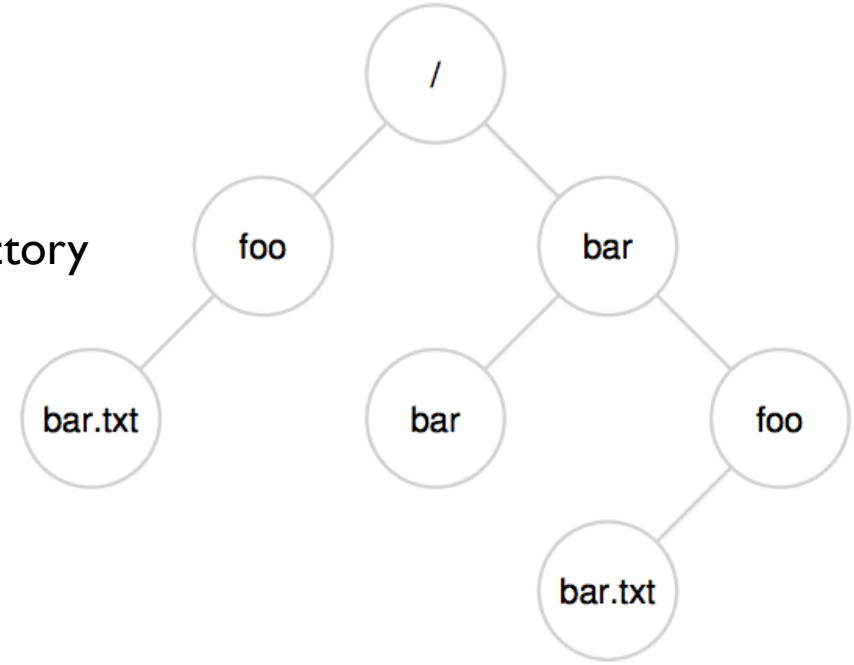
Directory Tree instead of single root directory
File name needs to be unique only within a directory

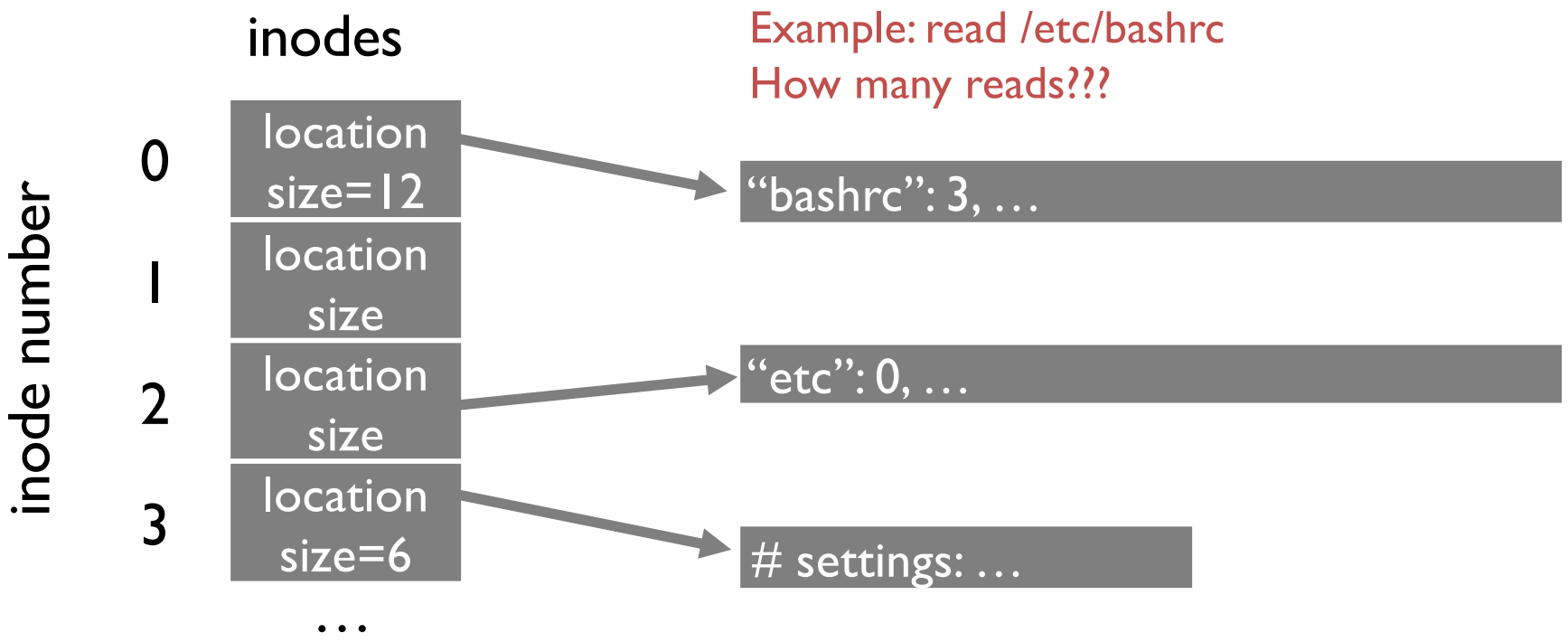
What are the path names of all the files?

/foo/bar.txt
/bar/bar
/bar/foo/bar

Store file-to-inode mapping in each directory

Reads for getting final inode called "traversal"





1. Inode #2 → Get location of root directory
2. Read root directory data; see "etc" maps to inode 0
3. Inode #0 → Get location of etc directory
4. Read /etc directory; see "bashrc" is at inode 3
5. Inode #3 → Get location of /etc/bashrc file data
6. Read /etc/bashrc file data

DIRECTORY CALLS

Directories can be be stored very similarly to files

Add a bit to inode to designate if data is for “file” or “directory”

mkdir: create new directory

readdir: read/parse directory entries

Why no writedir?

SPECIAL DIRECTORY ENTRIES

```
$ ls -la
total 728
drwxr-xr-x  34 trh  staff    1156 Oct 19 11:41 .
drwxr-xr-x+ 59 trh  staff    2006 Oct  8 15:49 ..
-rw-r--r--@  1 trh  staff    6148 Oct 19 11:42 .DS_Store
-rw-r--r--   1 trh  staff     553 Oct  2 14:29 asdf.txt
-rw-r--r--   1 trh  staff     553 Oct  2 14:05 asdf.txt~
drwxr-xr-x   4 trh  staff     136 Jun 18 15:37 backup
…
```

What will you see here?
cd /; ls -lia

FILE API (ATTEMPT 2)

```
read(char *path, void *buf, off_t offset, size_t nbyte)
write(char *path, void *buf, off_t offset, size_t nbyte)
```

Disadvantages?

Expensive traversal!

Goal: traverse once

FILE NAMES

Three types of names:

1. ~~inode~~
2. ~~path~~
3. file descriptor

3) FILE DESCRIPTOR (FD)

Idea:

Do expensive traversal once (open file)
Store inode in descriptor object (kept in memory)

Do reads/writes via descriptor, which tracks offset

Each process:

File-descriptor table contains pointers to open file descriptors

Integers used for file I/O are indexes into this table

stdin: 0, stdout: 1, stderr: 2

FILE API (ATTEMPT 3)

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

advantages:

- string names
- hierarchical
- traverse once
- offsets precisely defined

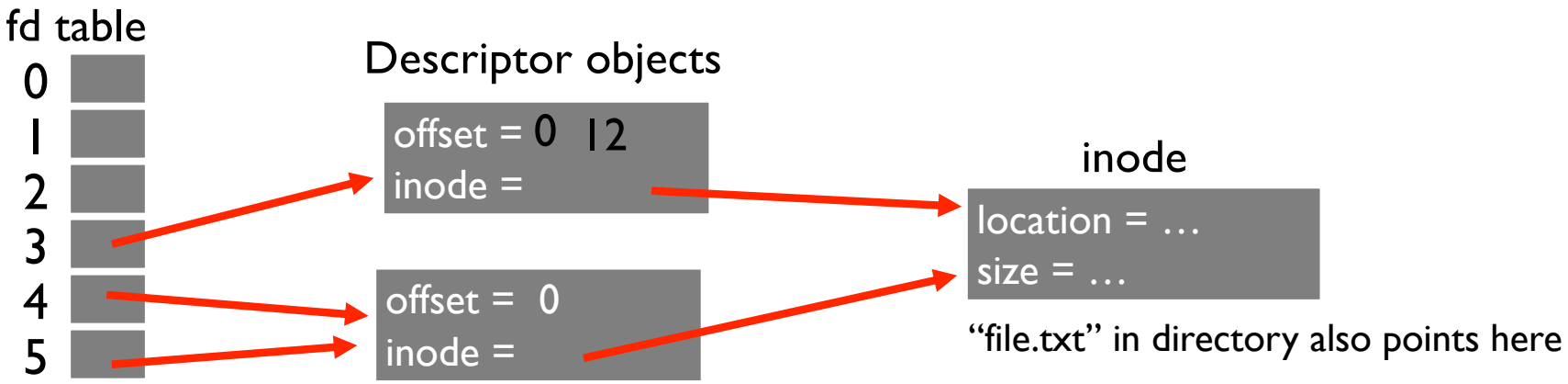
FD TABLE (XV6)

```
struct file {
  ...
  struct inode *ip;
  uint off;
};

// Per-process state
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
  ...
}
```

```
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
```

CODE SNIPPET: OPEN VS. DUP




```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
int fd2 = open(“file.txt”); // returns 4
int fd3 = dup(fd2);         // returns 5
```

READ NOT SEQUENTIALLY

```
off_t lseek(int fildes, off_t offset, int whence)

    If whence is SEEK_SET, the offset is set to offset bytes

    If whence is SEEK_CUR, the offset is set to its current
         location plus offset bytes

    If whence is SEEK_END, the offset is set to the size of
         the file plus offset bytes
```

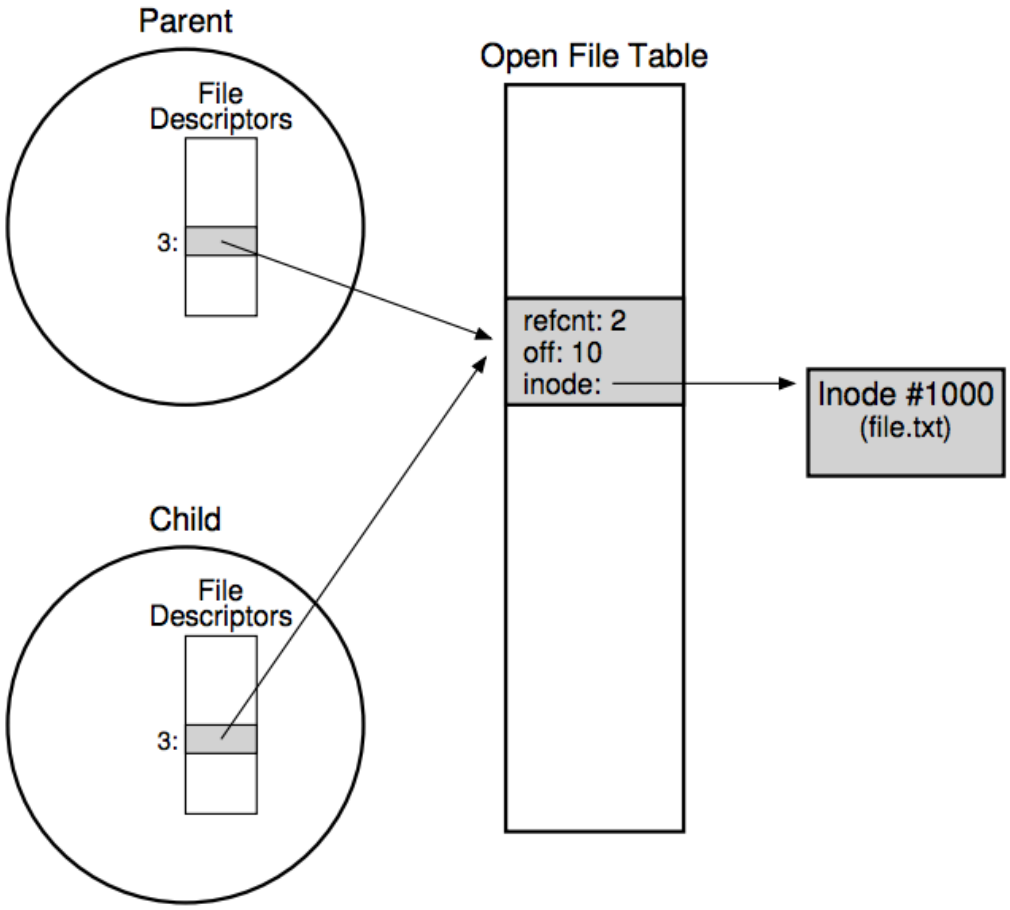
NEIGHBOR CHAT

```
int fd1 = open("file.txt"); // returns 12
int fd2 = open("file.txt"); // returns 13
read(fd1, buf, 16);
int fd3 = dup(fd2);         // returns 14
read(fd2, buf, 16);
lseek(fd1, 100, SEEK_SET);
```

What are the value of offsets in fd1, fd2, fd3 after the above code sequence?

WHAT HAPPENS ON FORK?

Man pages: The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent.



DELETING FILES

There is no system call for deleting files!

Inode (and associated file) is **garbage collected** when there are no references

Paths are deleted when: `unlink()` is called

FDs are deleted when: `close()` or process quits

REAL-WORLD ISSUES

A process can open a file, then remove the directory entry for the file so that it has no name anywhere in the file system, and still read and write the file. This is a disgusting bit of UNIX trivia and at first we were just not going to support it, but it turns out that all of the programs we didn't want to have to fix (csh, sendmail, etc.) use this for temporary files.

~ Sandberg *etal.*

LINKS

Hard links: Both path names use same inode number
File does not disappear until all removed; cannot link directories
Why not?

```
echo “Beginning…” > file1
ln file1 link
cat link
ls –li
echo “More info” >> file1
mv file1 file2
rm file2
```

No differences across two files that are hard linked
Note reference counts!
Links can be to files across directories as well

REFERENCE COUNTS

- Why is the reference count of "." always >= 2?

SOFT LINKS

Soft or symbolic links: Point to second path name
Can softlink to dirs

```
ln -s oldfile softlink
```

Softlink will have new inode number
Set bit in inode designating "soft link"; Interpret associated data as file name!
See identifying bits in "ls –li"; Note reference counts…

How can you get confusing behavior: "file does not exist"!
Confusing behavior: "cd linked_dir; cd ..; in different parent!

NEIGHBOR CHAT

Consider the following code snippet:

```
echo “hello” > oldfile
ln –s oldfile link1
ln oldfile link2
rm oldfile
```

What will be the output of `cat link1`

What will be the output of `cat link2`

COMMUNICATING REQUIREMENTS: FSYNC

File system keeps newly written data in memory for awhile
    Buffer cache
Useful for reads (don't have to access slow disk)

Also useful for writes
Write buffering improves performance (why?)

But what if system crashes before buffers are flushed?

fsync(int fd) forces buffers to flush from meory to disk, tells disk to flush its write cache
Makes data **durable**

What happens when you call close(fd)?

MAN PAGES FOR CLOSE(FD)

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes.

Typically, filesystems do not flush buffers when a file is closed.

If you need to be sure that the data is physically stored on the underlying disk, use fsync(2).

(It will depend on the disk hardware at this point.)

RENAME

rename(char *old, char *new):

- deletes an old link to a file
- creates a new link to a file

Just changes name of file, does not move (or copy) data

Even when renaming to new directory

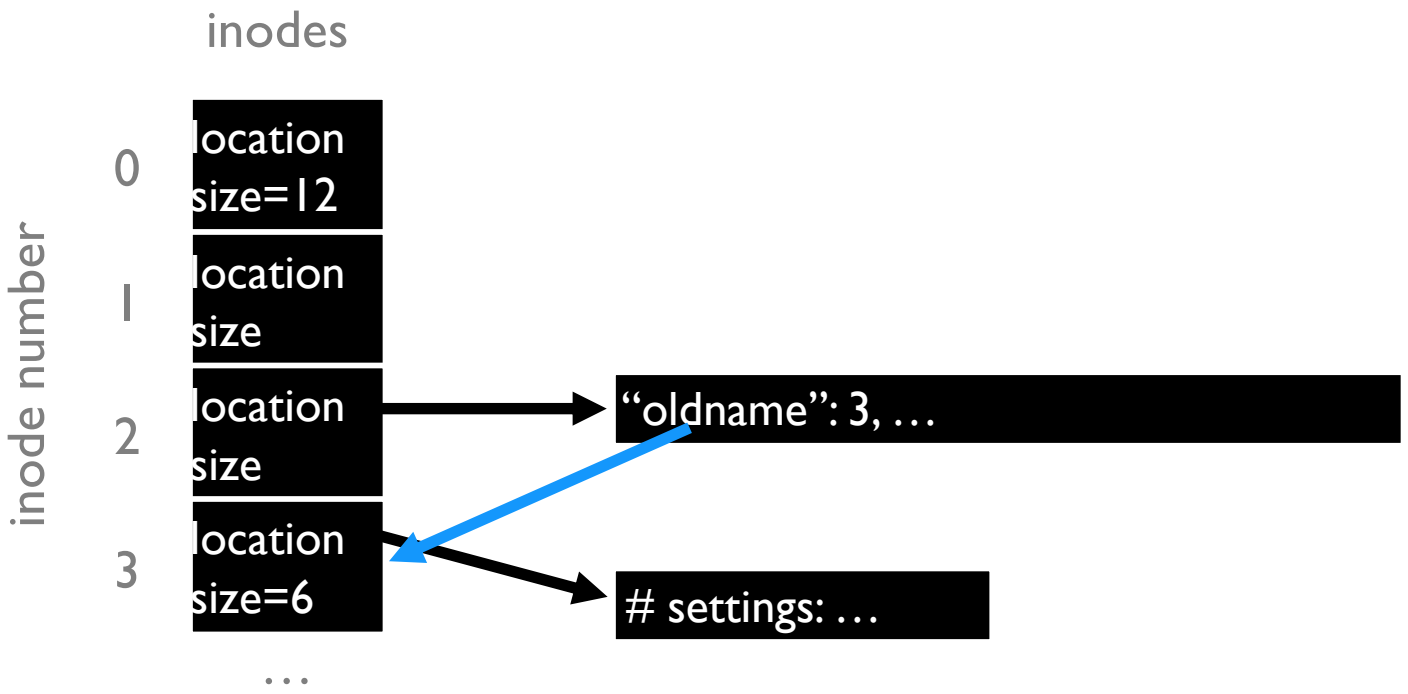
What can go wrong if system crashes at wrong time?

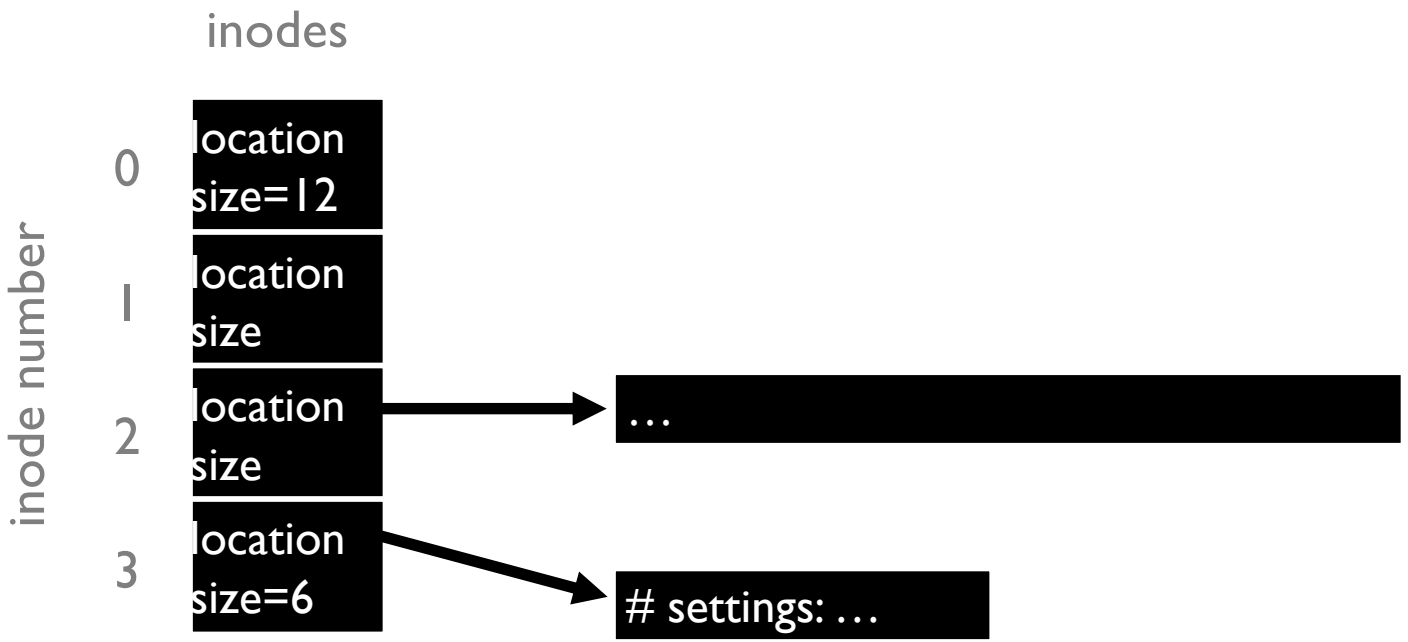
ATOMIC FILE UPDATE

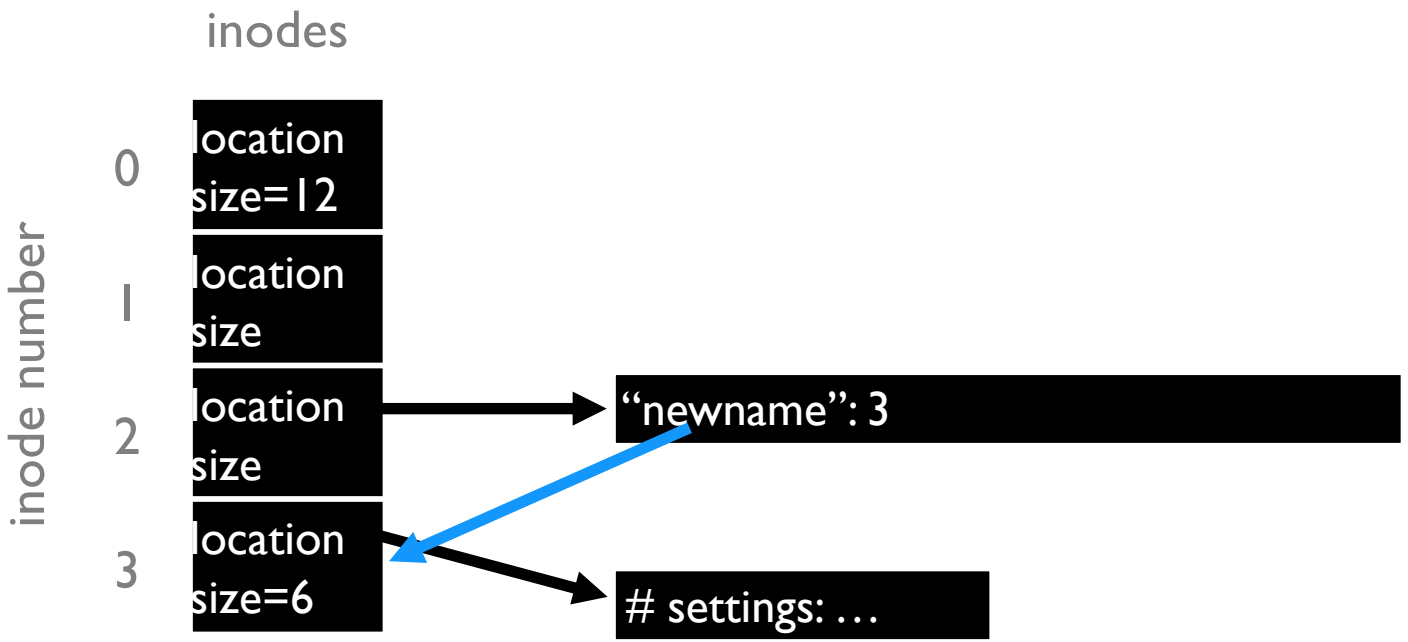
Rename operation must be atomic within file system

Say application wants to update file.txt atomically
If crash, should see only old contents or only new contents

1. write new data to file.txt.tmp file
2. fsync file.txt.tmp
3. rename file.txt.tmp over file.txt, replacing it


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
“oldname”: 3, …
settings: …


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
…
settings: …


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
“newname”: 3
settings: …

ATOMIC FILE UPDATE

With journaling file systems, will see how to make operations like rename atomic…

PERMISSIONS, ACCESS CONTROL

```
➜  xv6-sp19 ls -la .
total 5547
drwxrwxr-x  7 shivaram shivaram    2048 Mar 10 22:59 .
drwxr-xr-x 47 shivaram shivaram    6144 Apr  4 11:27 ..
-rwxrwxr-x  1 shivaram shivaram     106 Mar  6 15:23 bootother
-rw-r-----  1 shivaram shivaram     223 Feb 28 17:37 FILES
drwxrwxr-x  2 shivaram shivaram    2048 Mar  6 15:23 fs
-rw-rw-r--  1 shivaram shivaram  524288 Mar  6 15:23 fs.img
```

```
➜  xv6-sp19 fs la .
Access list for . is
Normal rights:
  system:administrators rlidwka
  system:anyuser l
  shivaram rlidwka
```

MANY FILE SYSTEMS

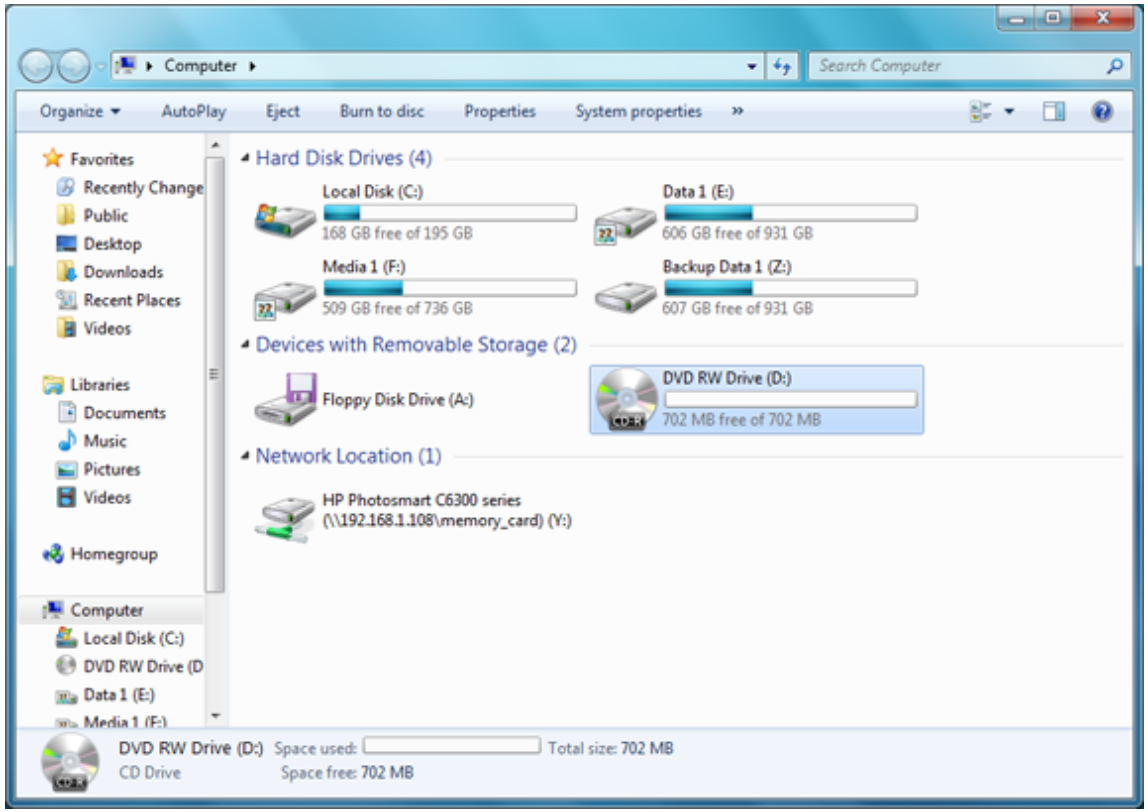
Users often want to use many file systems

For example:

- main disk
- backup disk
- AFS
- thumb drives

What is the most elegant way to support this?

MANY FILE SYSTEMS: APPROACH 1

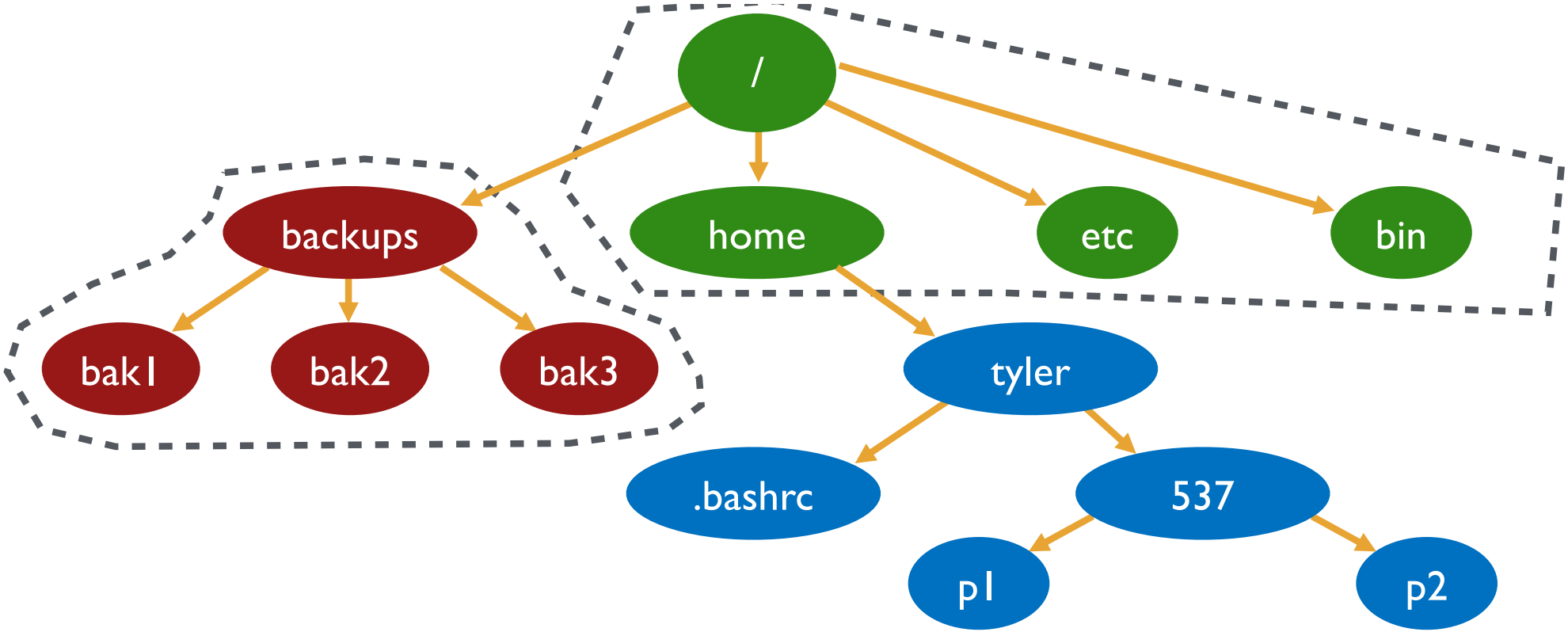


http://www.ofzenandcomputing.com/burn-files-cd-dvd-windows7/

MANY FILE SYSTEMS: APPROACH 2

Idea: stitch all the file systems together into a super file system!

```
sh> mount
/dev/sda1 on / type ext4 (rw)
/dev/sdb1 on /backups type ext4 (rw)
AFS on /home type afs (rw)
```



/dev/sda1 **on** /

/dev/sdb1 **on** /backups

AFS **on** /home

SUMMARY

Using multiple types of names provides convenience and efficiency

- inodes
- path names
- file descriptors

Special calls (fsync, rename) let developers communicate requirements to file system

Mount and link features provide flexibility