

[537] File-System APIs

Chapter 39
Tyler Harter
11/03/14

Review RAID

RAID

Idea: build an awesome disk from small, cheap disks.

Metrics: ???

RAID

Idea: build an awesome disk from small, cheap disks.

Metrics: capacity, reliability, performance

Fundamental tradeoffs.

Why can't we have the best capacity and reliability?

RAID

RAID-0: no redundancy

RAID-1: mirroring

RAID-4: parity disk

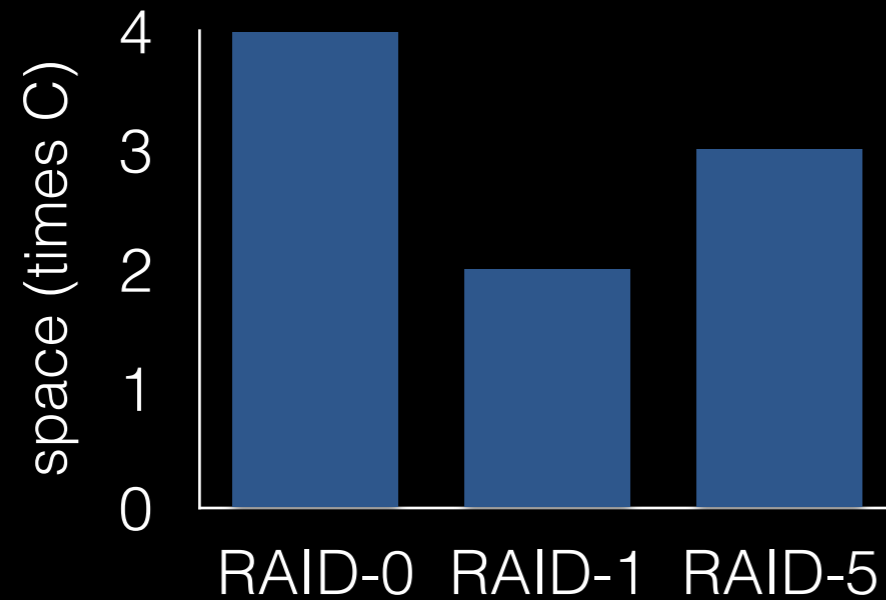
RAID-5: parity block (rotated between disks)

RAID Tradeoffs

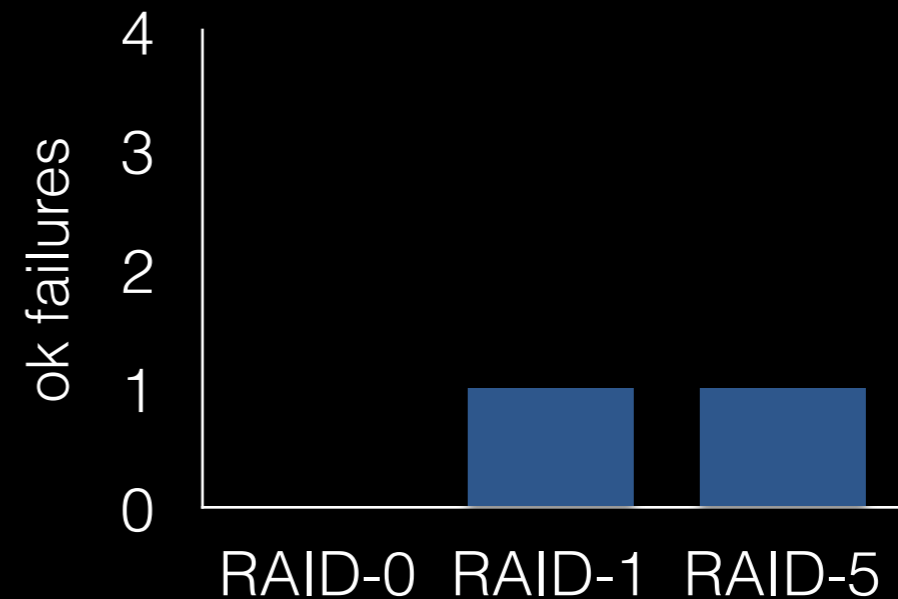
Assume **4** disks.

Eval RAID-0, RAID-1, and RAID-5 (why not RAID-4?)

capacity

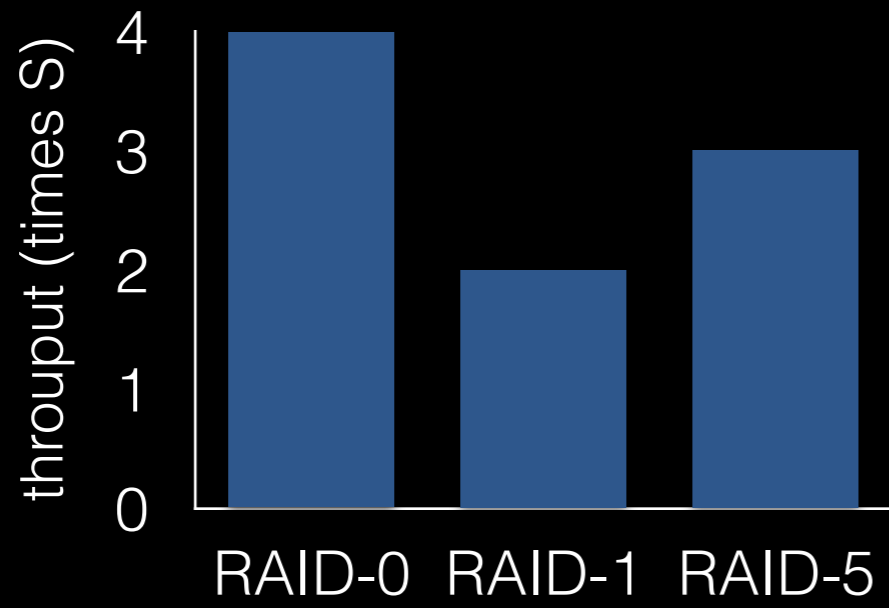


reliability

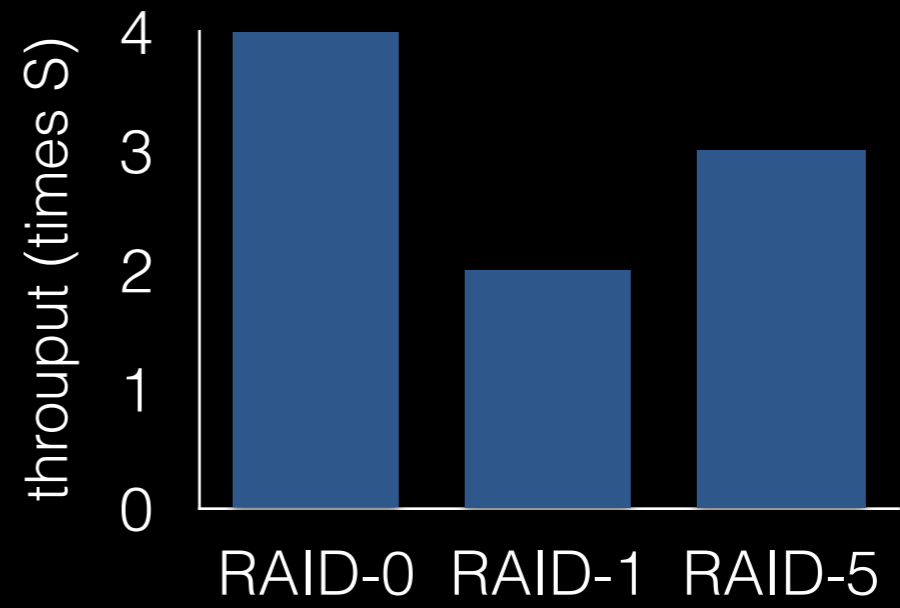


sequential

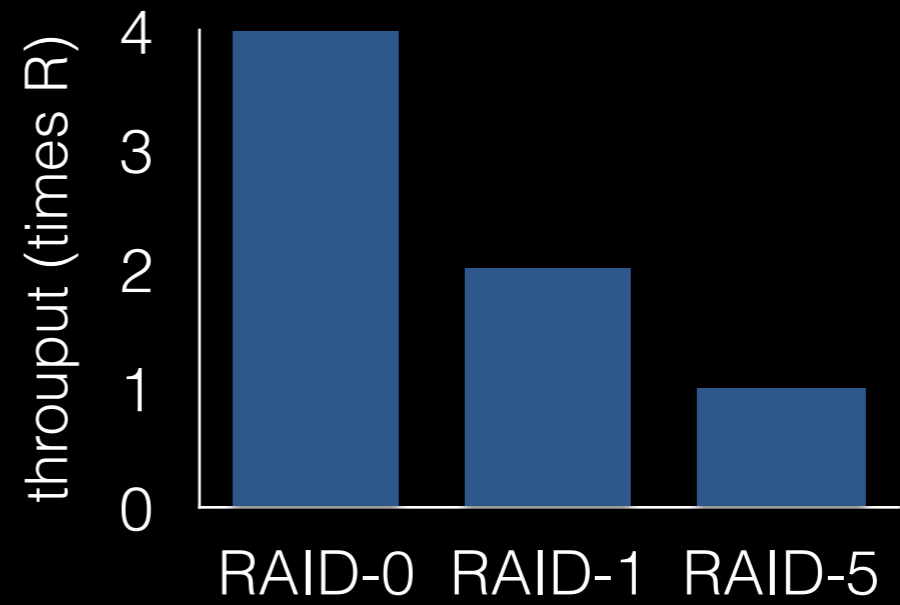
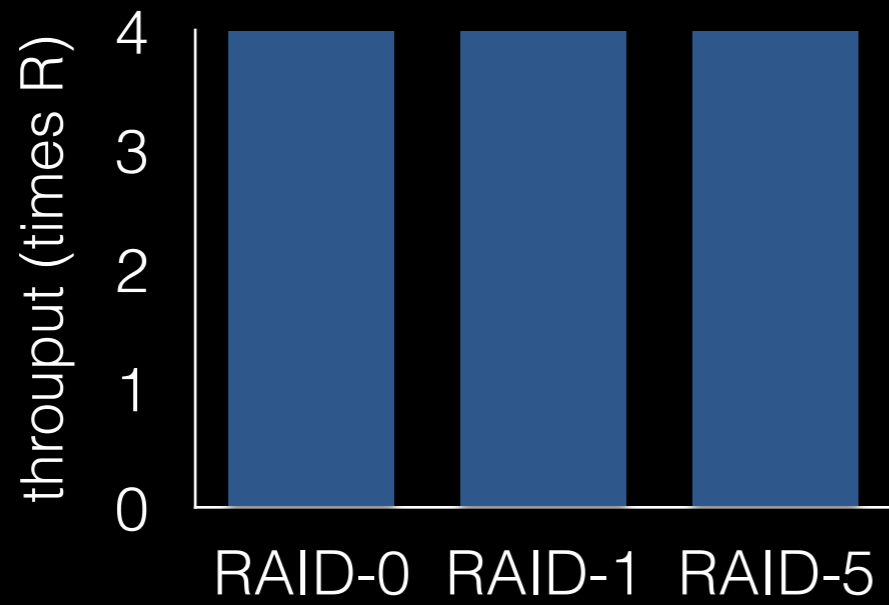
reads



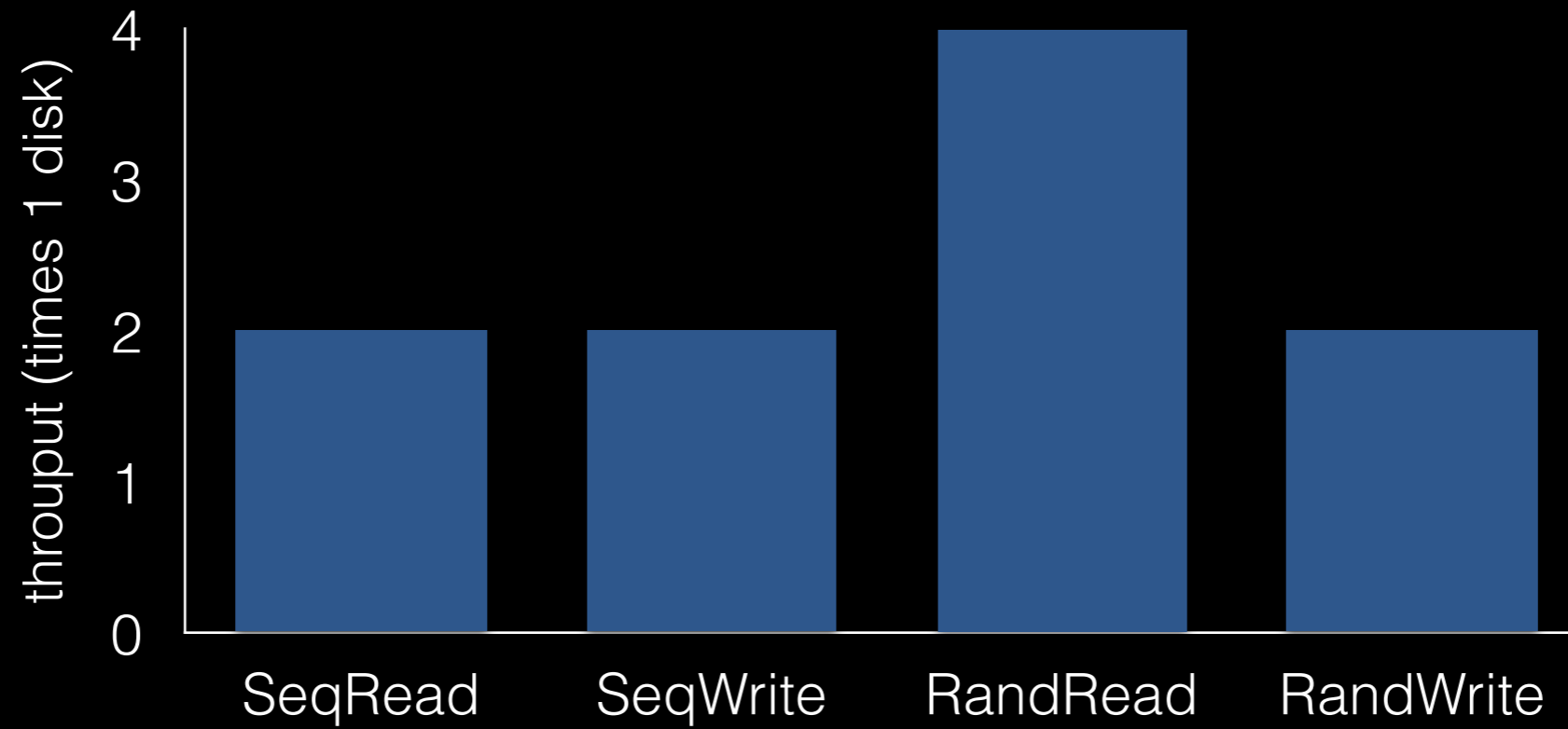
writes



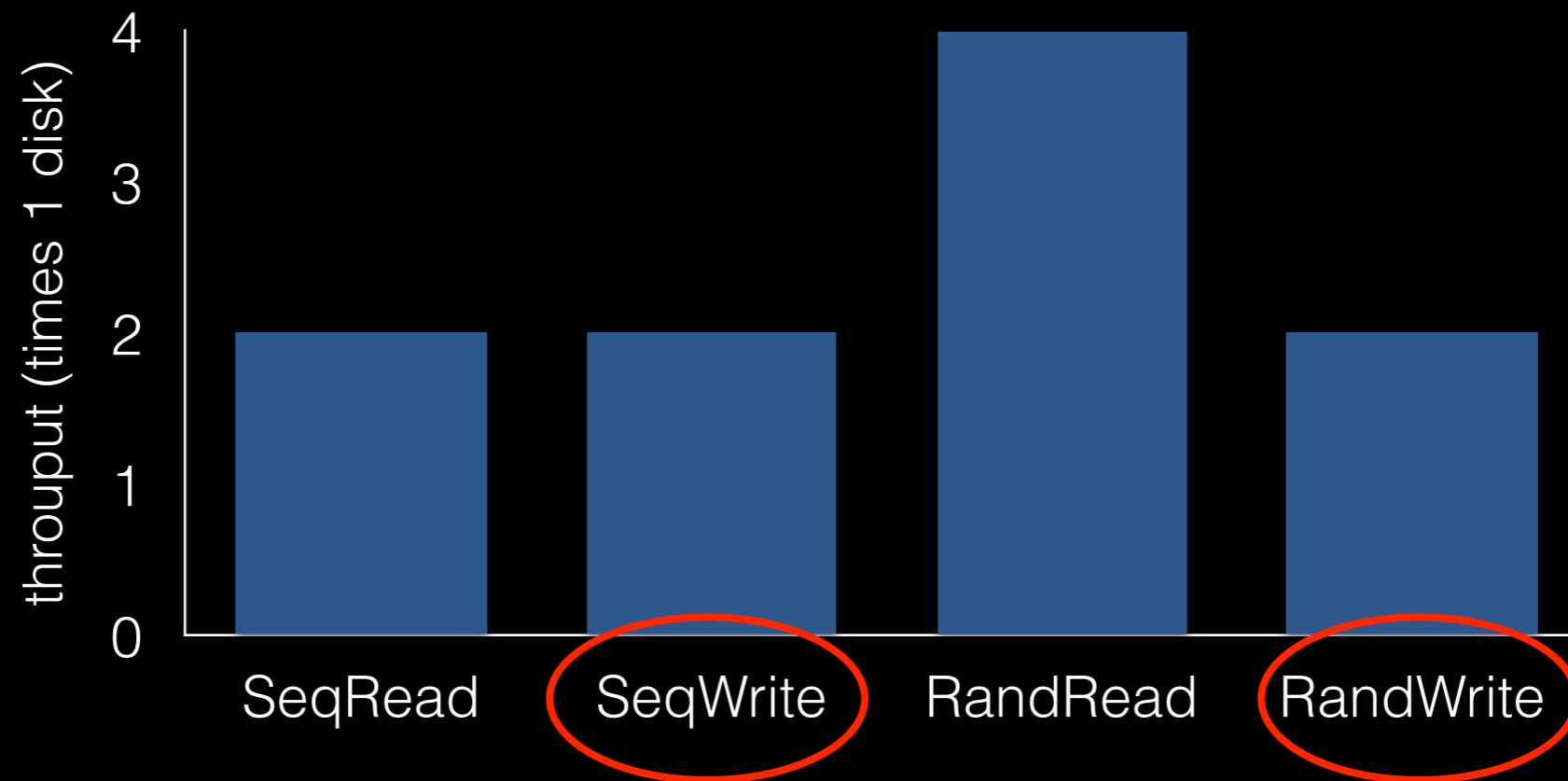
random



RAID-1 Analysis

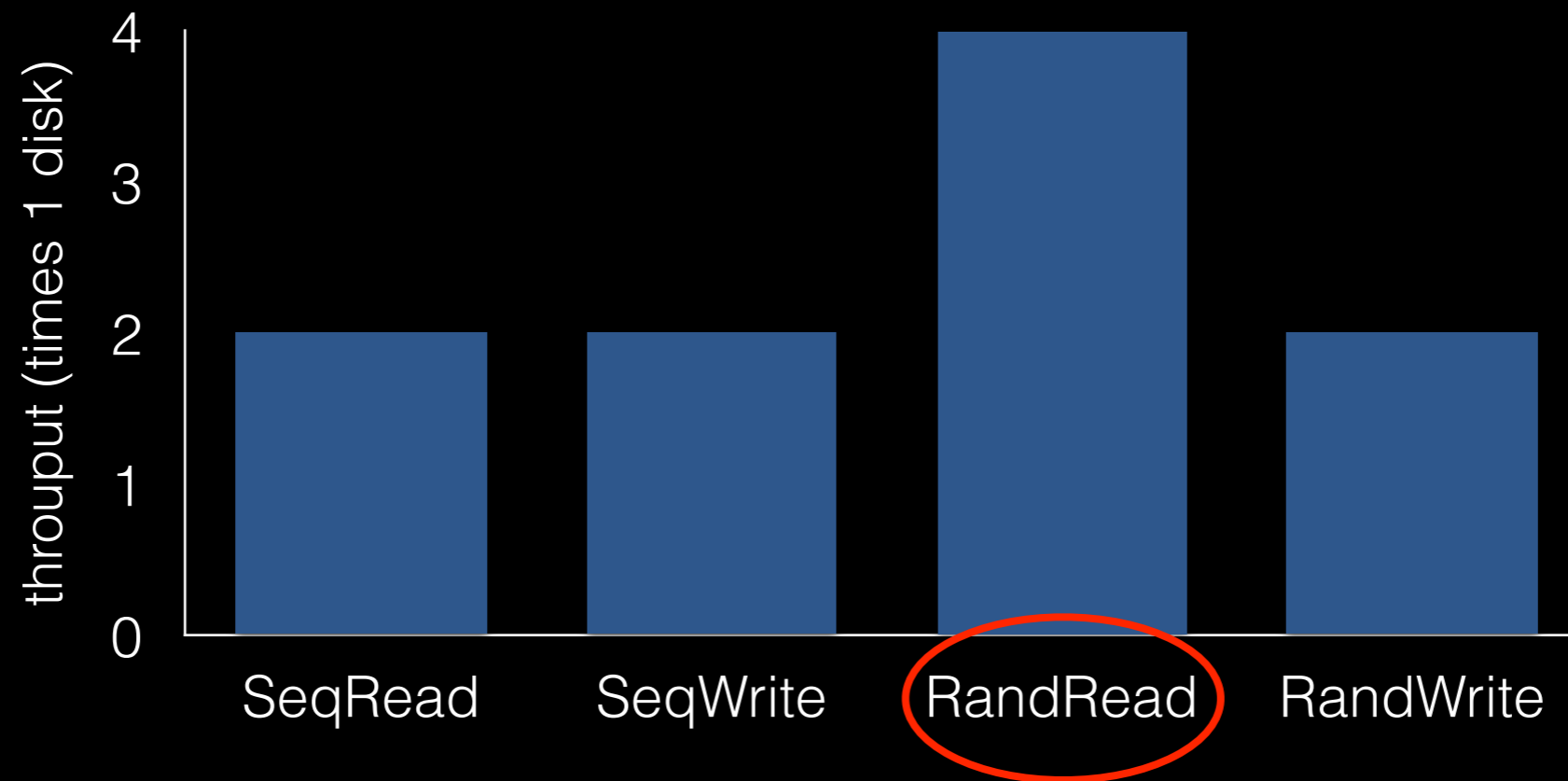


RAID-1 Analysis



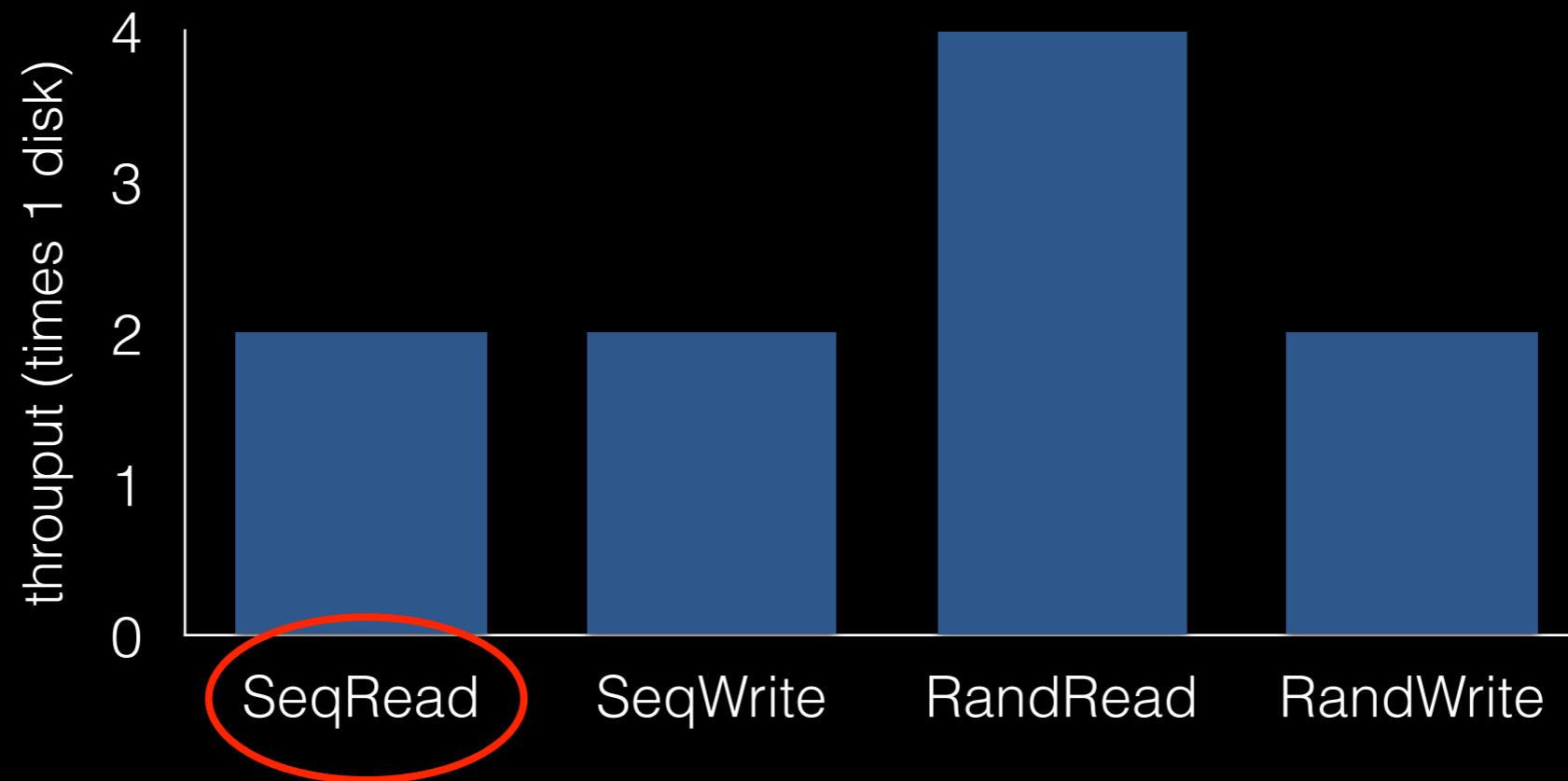
All data is written twice, so **write throughput** is halved.

RAID-1 Analysis



A mix of **random reads** can spread across all disks.

RAID-1 Analysis



Why do **sequential reads** only get half throughput?

RAID-1: Sequential Reads

Reads: 101, 102, 103, 104, 105, 106, 107, 108, ...

Assume 4 disks.

Each **logical** block is stored on two **physical** disks.

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:


98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:


98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:


98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$$\text{time} = 8 * x$$

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:




98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 2:



97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:



98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----



Disk 3:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----



$$\text{time} = 8 * x$$

Disk 0:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 1:

98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

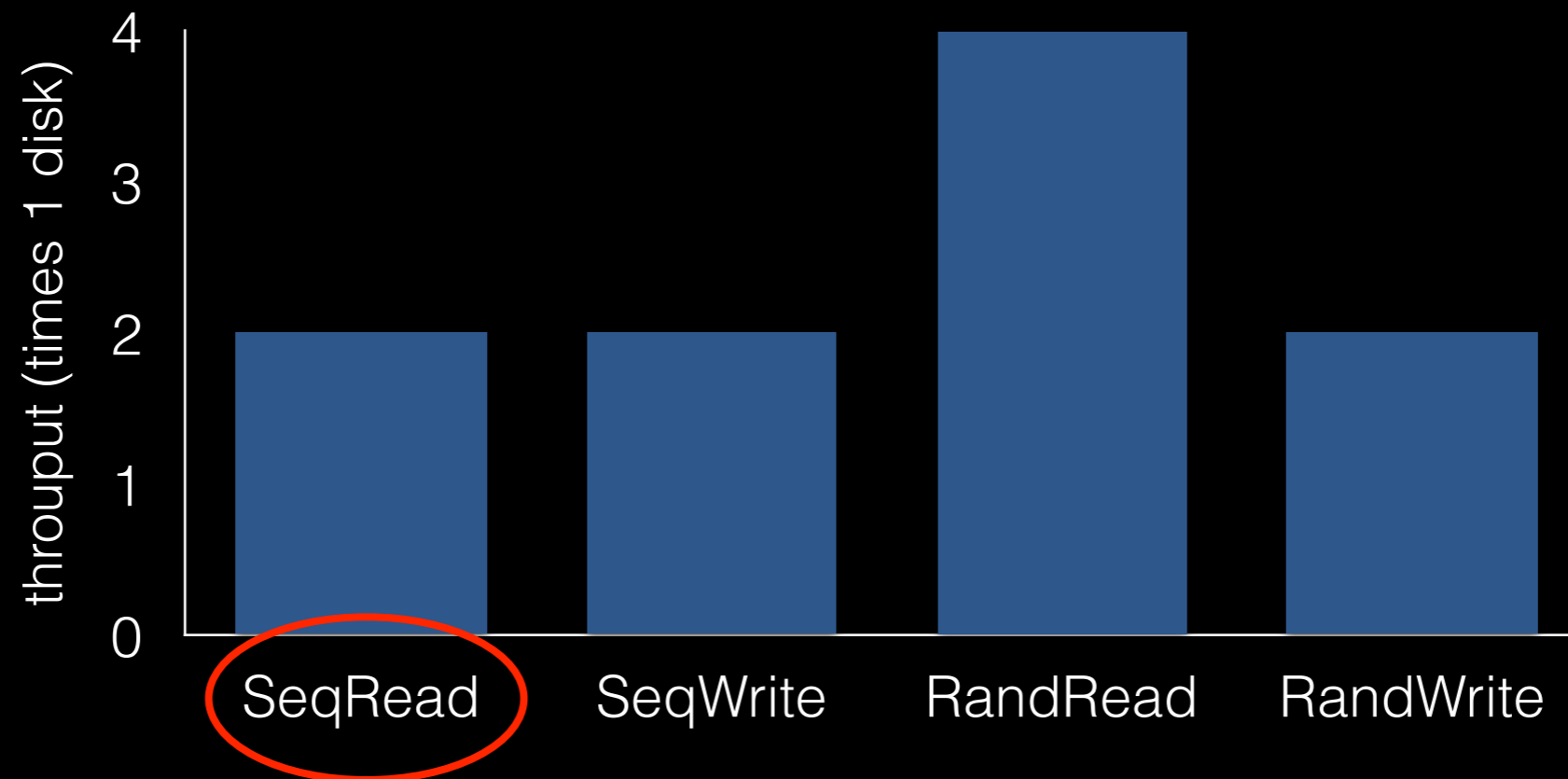
Disk 2:

97	99	101	103	105	107	109	111	113	115
----	----	-----	-----	-----	-----	-----	-----	-----	-----

Disk 3:

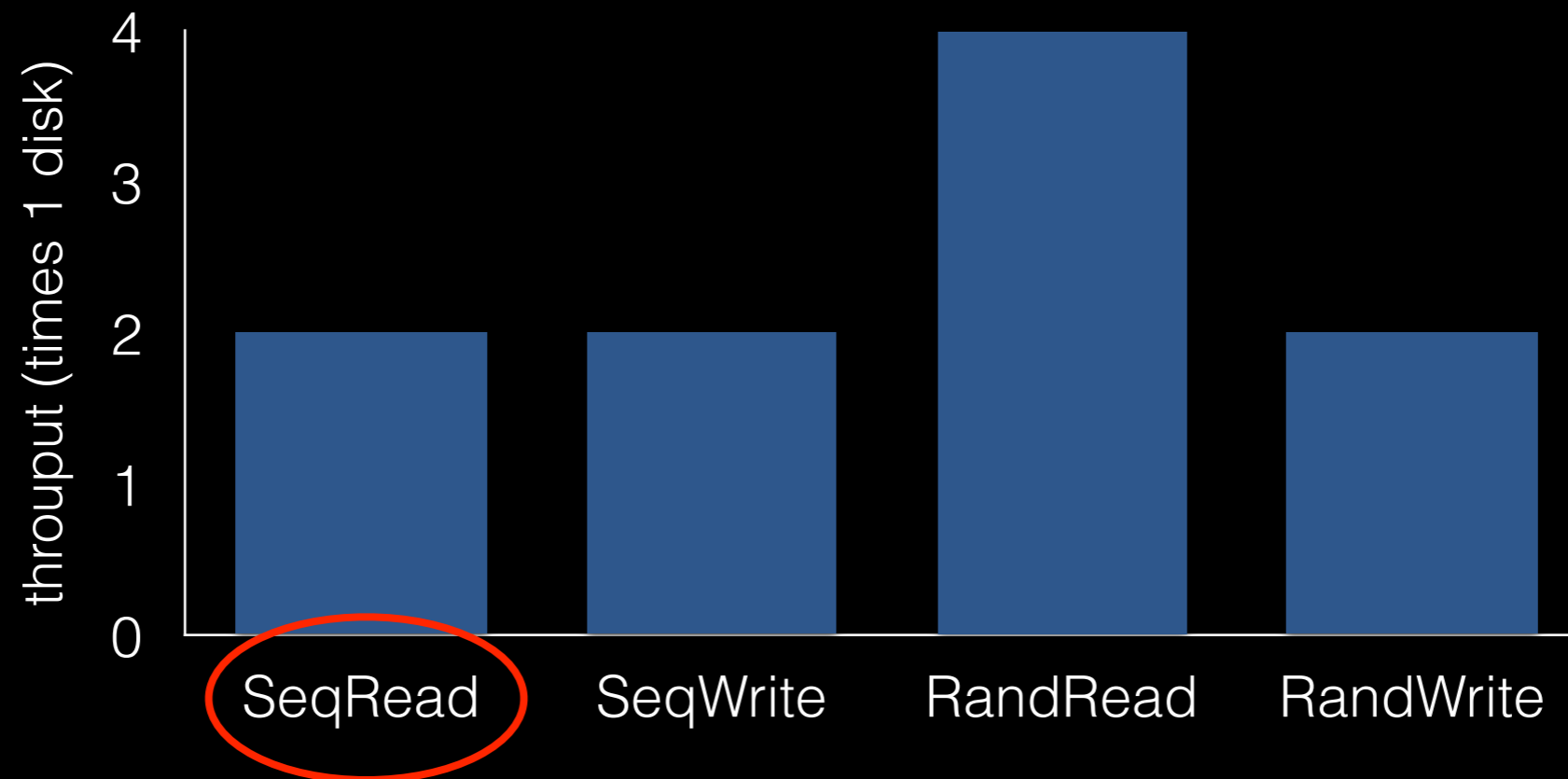
98	100	102	104	106	108	110	112	114	116
----	-----	-----	-----	-----	-----	-----	-----	-----	-----

RAID-1 Analysis



Why do **sequential reads** only get half throughput?

RAID-1 Analysis



Why do **sequential reads** only get half throughput?
Because skipping every other block doesn't save.

File-System Abstraction

What is a File?

Array of bytes.

Ranges of bytes can be read/written.

File system consists of *many* files.

What is a File?

Array of bytes.

Ranges of bytes can be read/written.

File system consists of **many** files.

Files need **names** so programs can choose the right one.

File Names

Three types of names:

- inode
- path
- file descriptor

File Names

Three types of names:

- **inode**
- path
- file descriptor

Inodes

Each file has **exactly one** inode number.

Inodes are unique (at a given time) within a FS.

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

Inodes

Each file has **exactly one** inode number.

Inodes are unique (at a given time) within a FS.

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

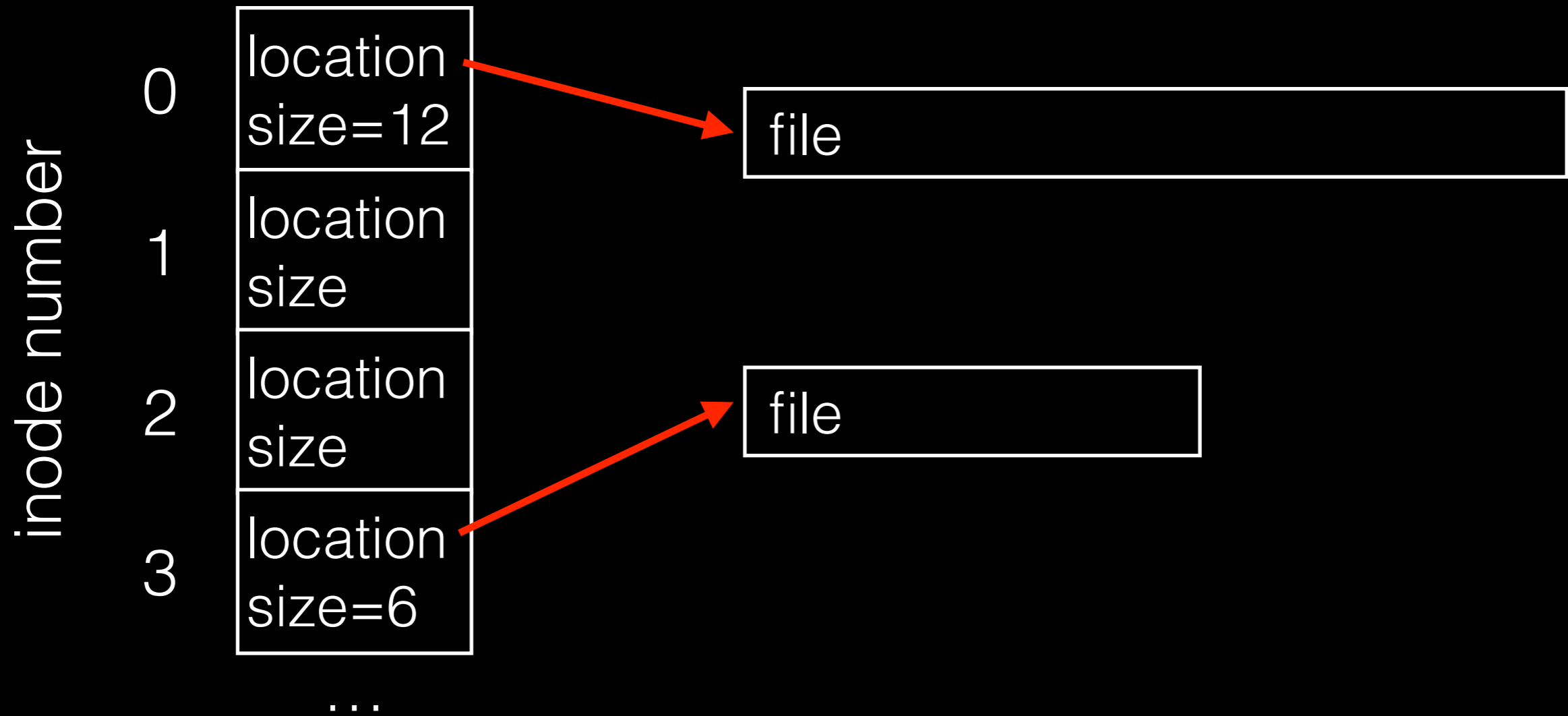
Show inodes via stat.

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

inodes



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)
```

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)
```

note: seek does not cause disk seek
unless followed by a read/write

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)
```

Disadvantages?

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)
```

Disadvantages?

- names hard to remember
- everybody has the same offset
- collisions (not hierarchical)

File API (attempt 1)

```
pread(int inode, void *buf,  
        off_t offset, size_t nbyte)  
pwrite(int inode, void *buf,  
         off_t offset size_t nbyte)  
seek(int inode, off_t offset)
```

Disadvantages?

- names hard to remember
- ~~everybody has the same offset~~
- collisions (not hierarchical)

File Names

Three types of names:

- inode
- path
- file descriptor

Paths

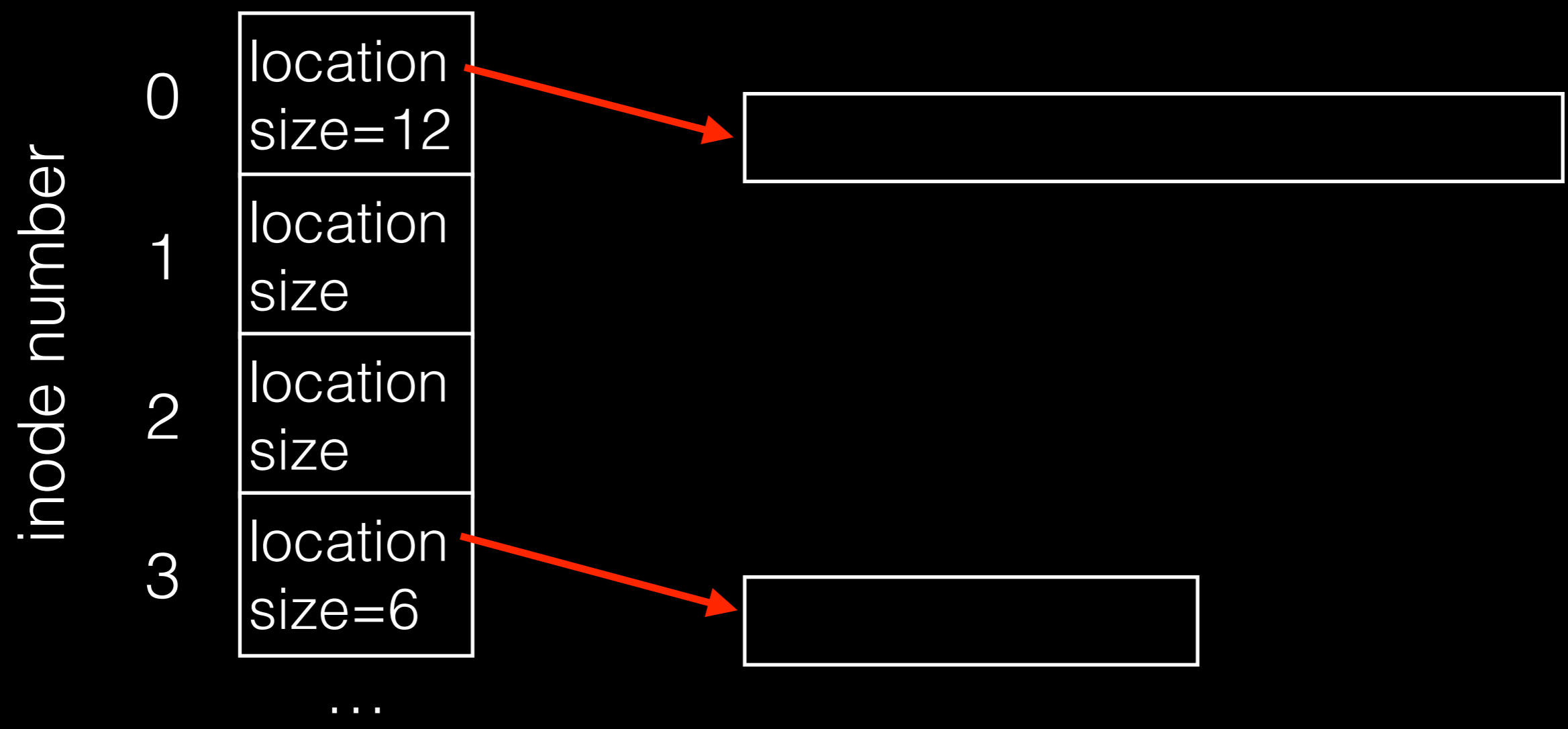
String names are friendlier than number names.

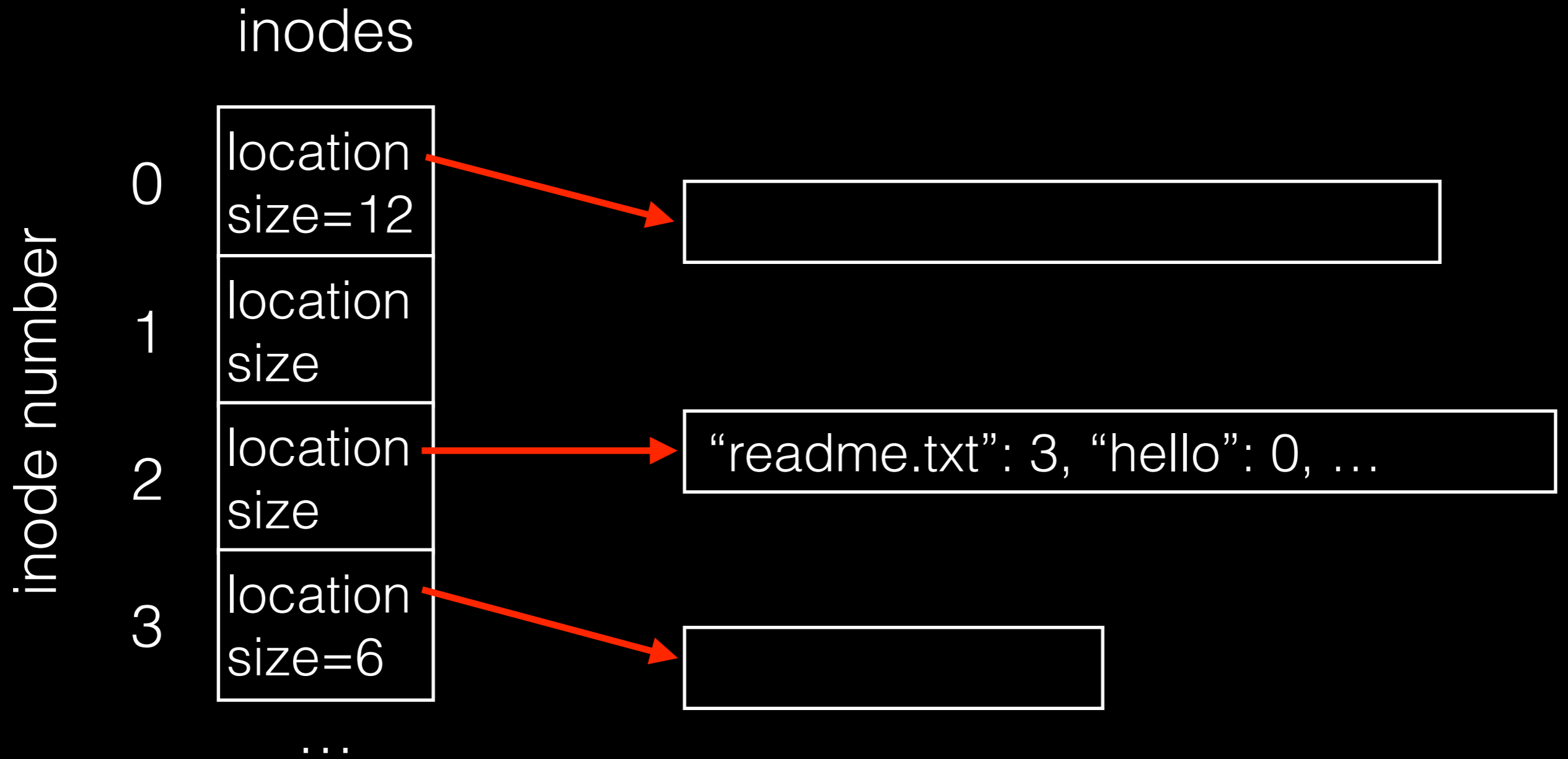
Paths

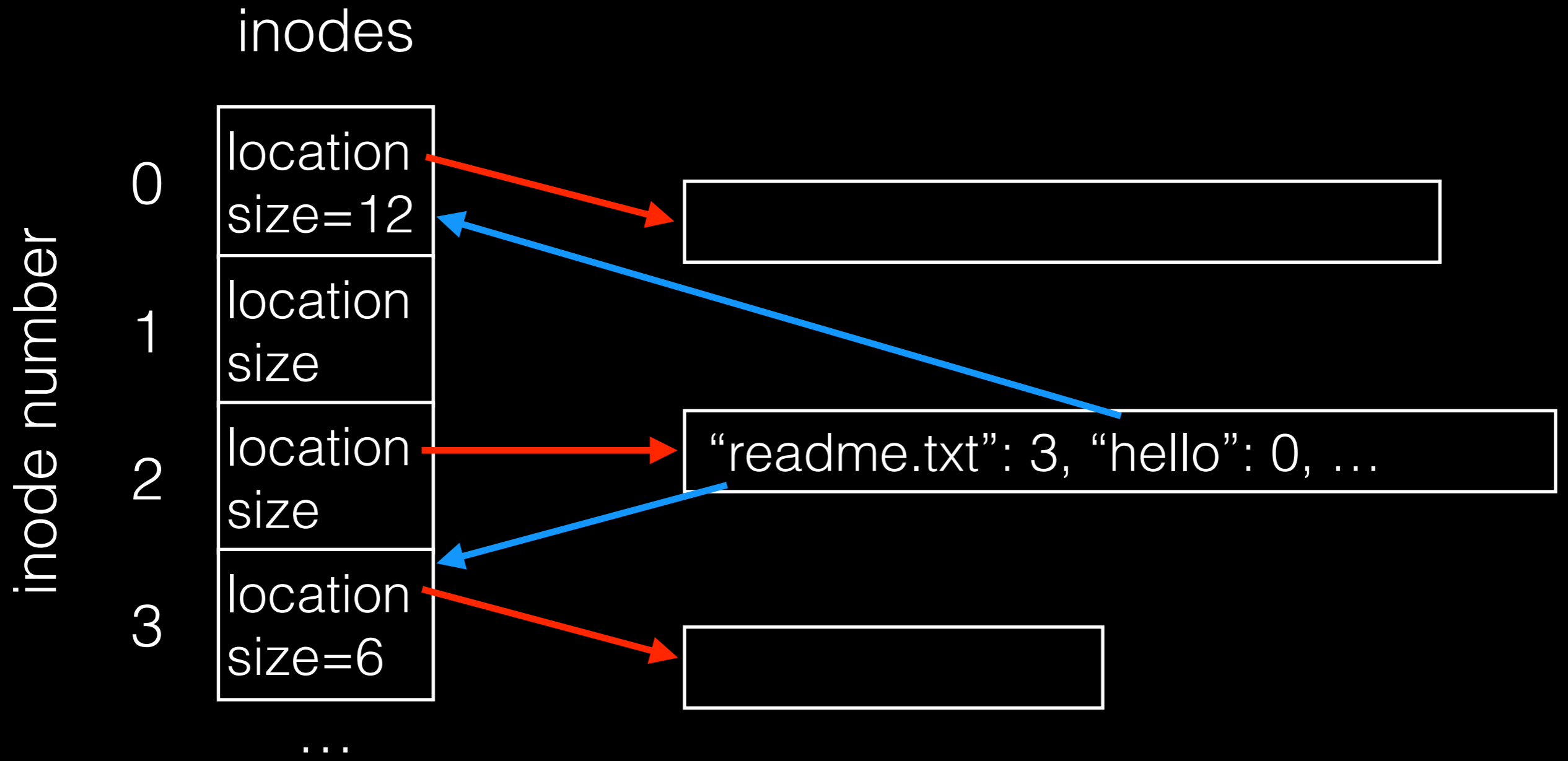
String names are friendlier than **number** names.

Store *path-to-inode* mappings in a predetermined “root” file (typically inode 2)

inodes







Paths

String names are friendlier than **number** names.

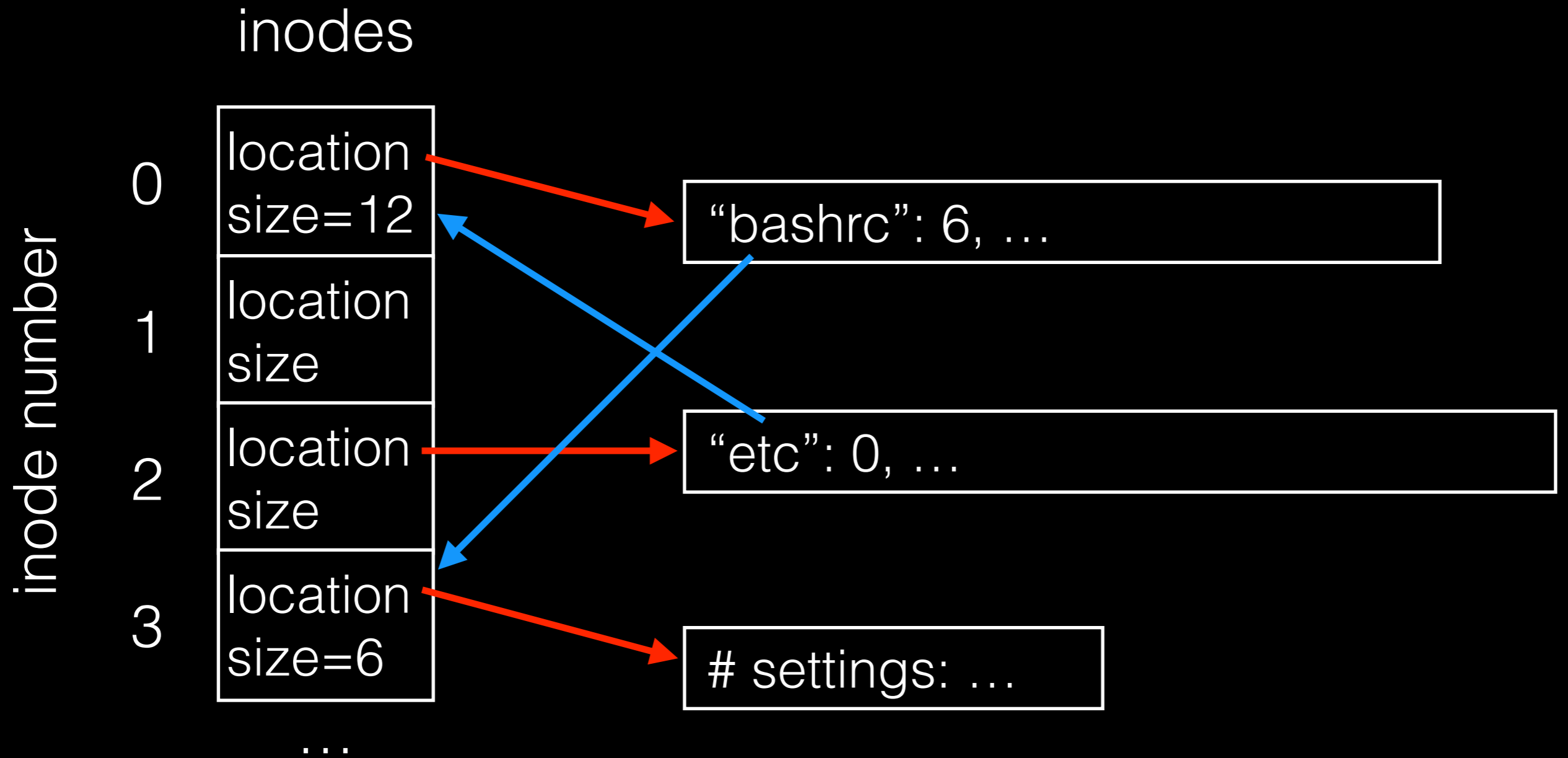
Store *path-to-inode* mappings in a predetermined “root” file (typically inode 2)

Paths

String names are friendlier than **number** names.

Store *path-to-inode* mappings in a predetermined “root” file (typically inode 2)

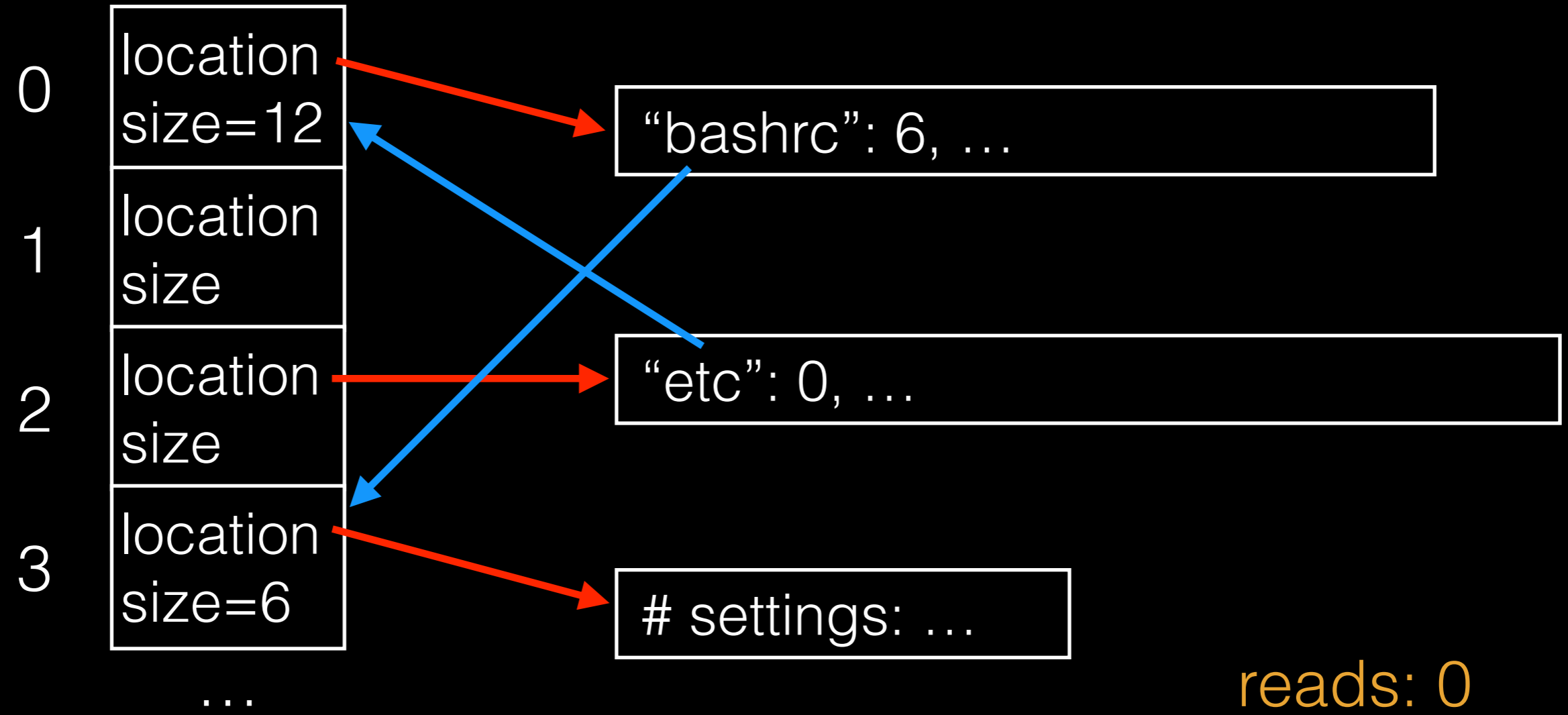
Generalize! Store path-to-inode mapping in many files. Call these special files **directories**.



inodes

read **/etc/bashrc**

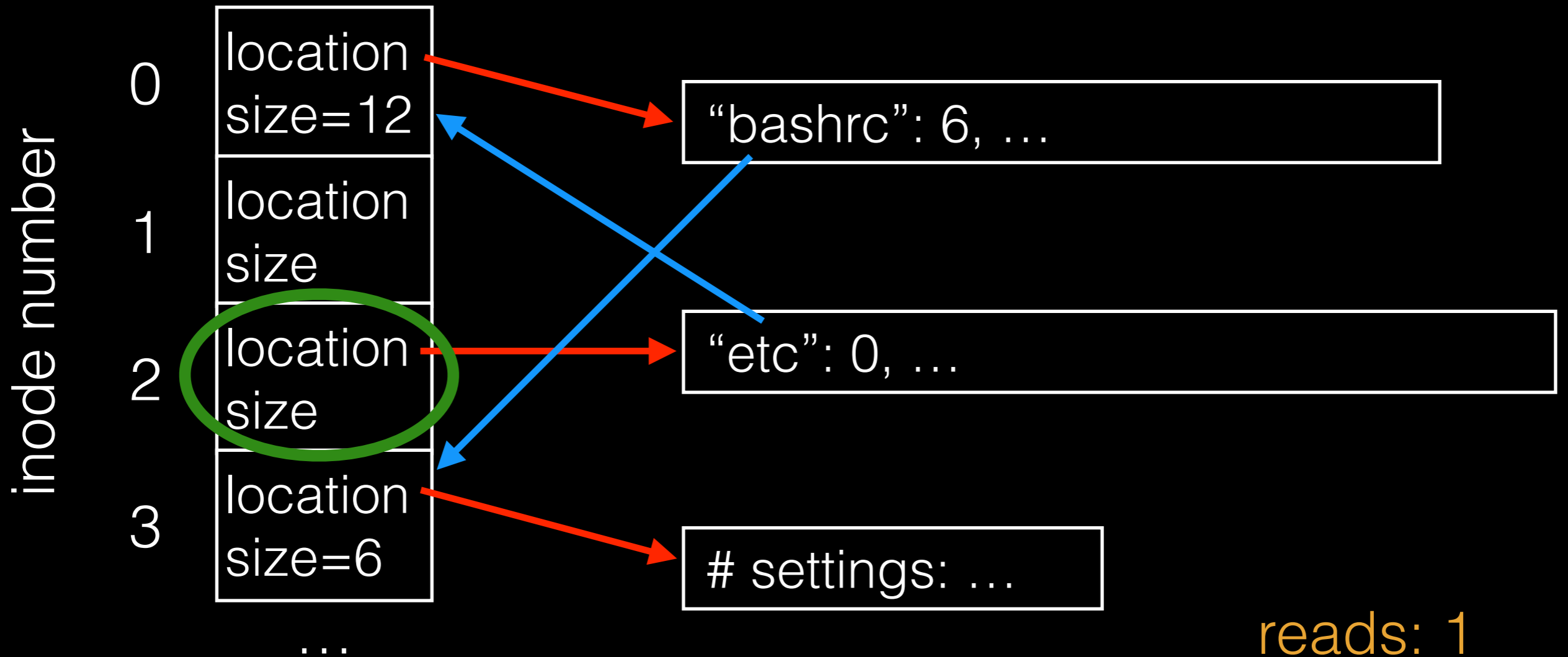
inode number



reads: 0

inodes

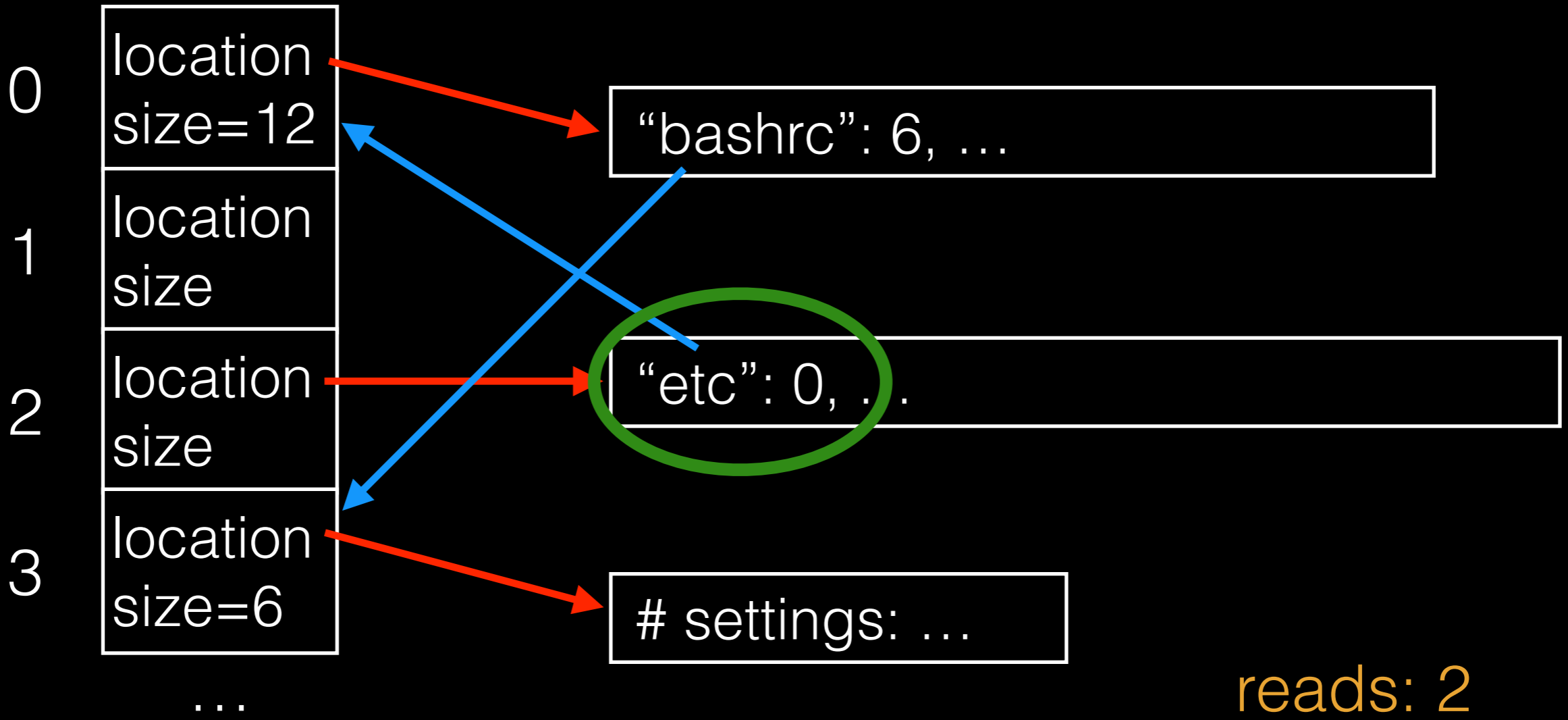
read **/etc/bashrc**



inodes

read **/etc/bashrc**

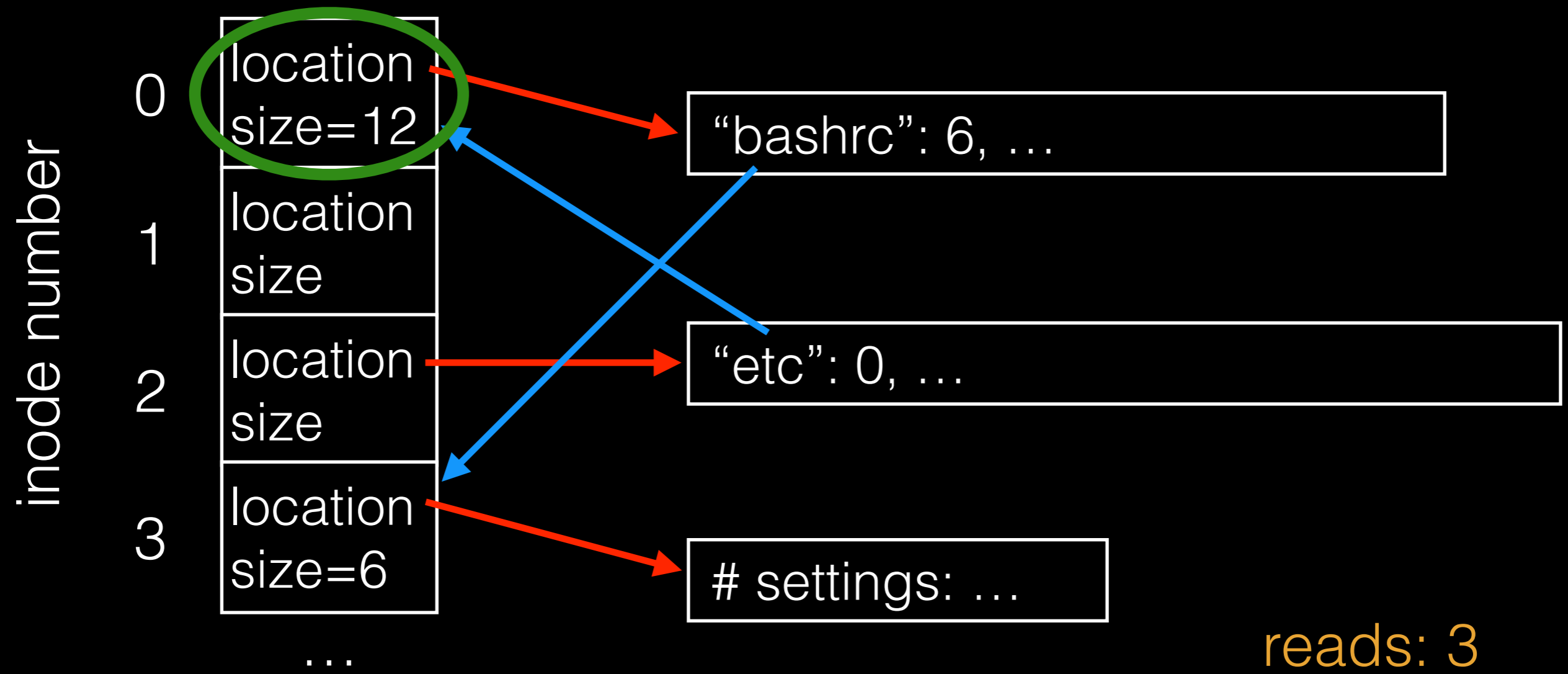
inode number



reads: 2

inodes

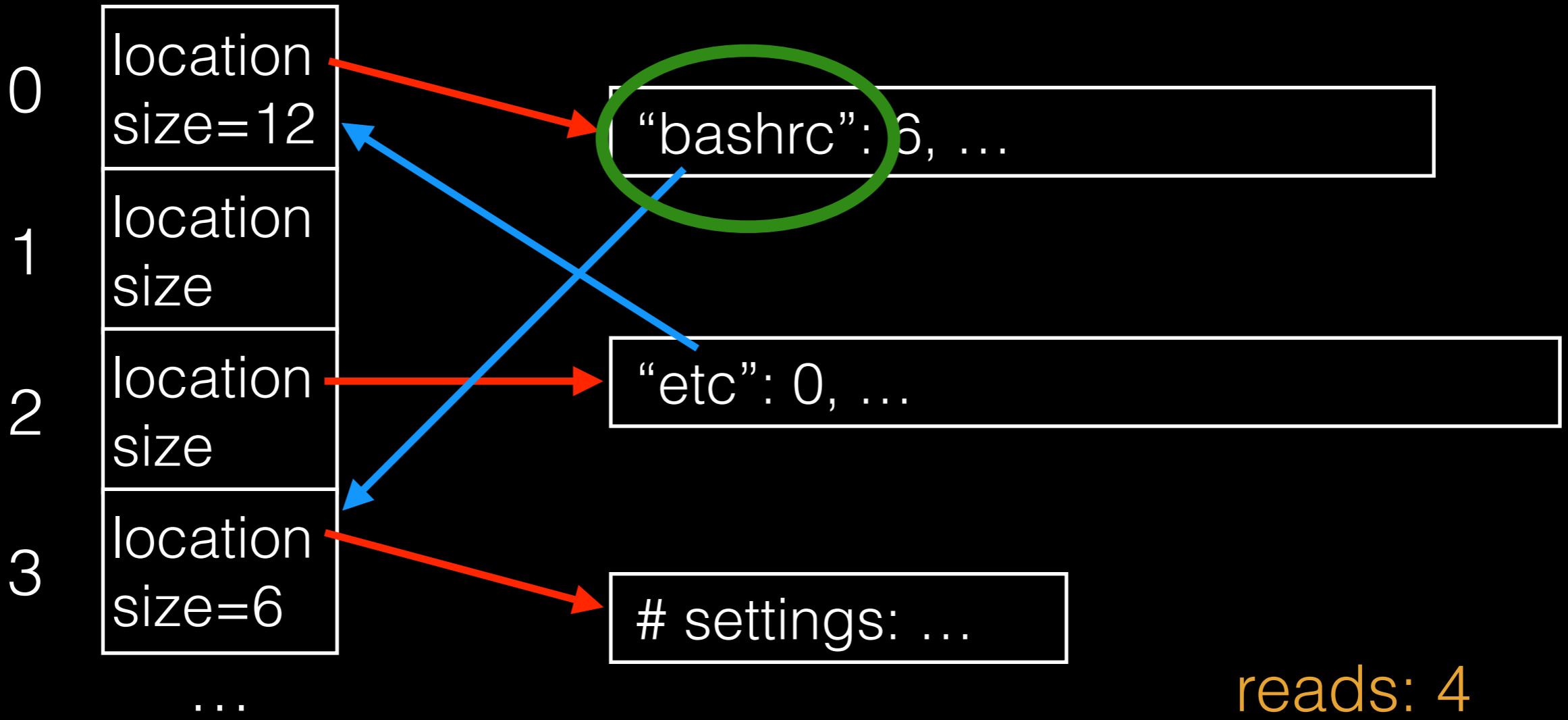
read **/etc/bashrc**



inodes

read **/etc/bashrc**

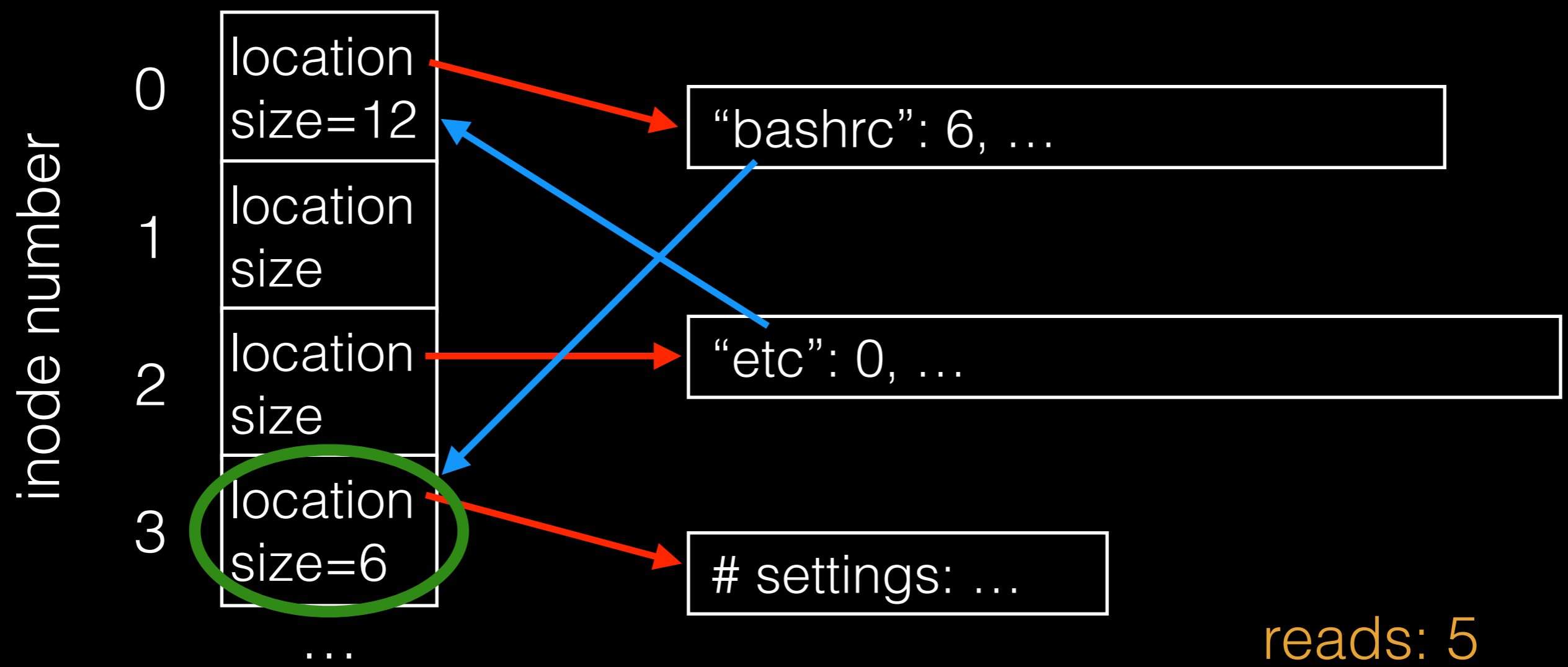
inode number



reads: 4

inodes

read **/etc/bashrc**

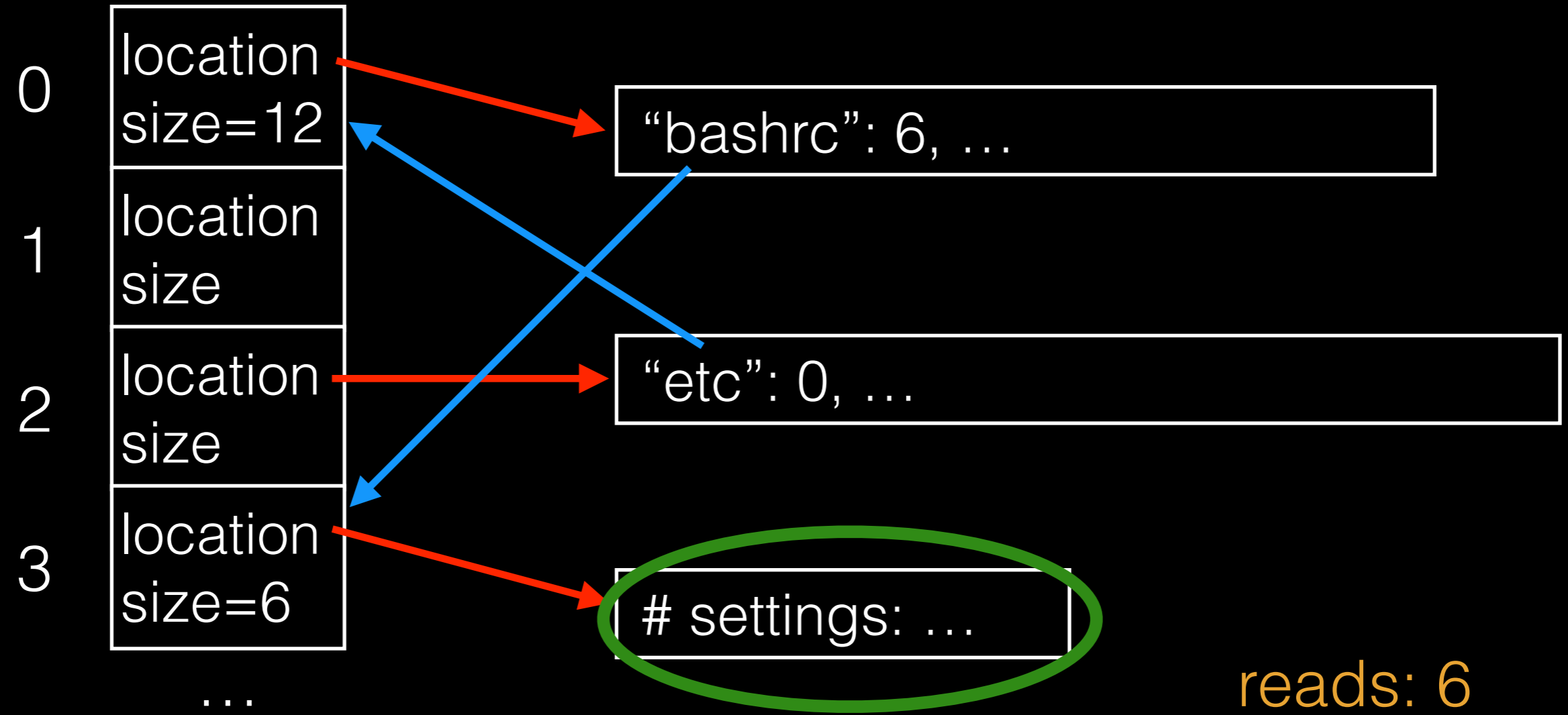


reads: 5

inodes

read **/etc/bashrc**

inode number



Paths

String names are friendlier than **number** names.

Store *path-to-inode* mappings in a predetermined “root” file (typically inode 2)

Generalize! Store path-to-inode mapping in many files. Call these special files **directories**.

Paths

String names are friendlier than **number** names.

Store *path-to-inode* mappings in a predetermined “root” file (typically inode 2)

Generalize! Store path-to-inode mapping in many files. Call these special files **directories**.

Reads for getting final inode called “**traversal**”.

Directory Calls

`mkdir`: create new directory

`readdir`: read/parse directory entries

Why no `writedir`?

Special Directory Entries

```
Tylers-MacBook-Pro:scratch trh$ 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
```

```
...
```

File API (attempt 2)

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

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

File API (attempt 2)

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

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

Disadvantages?

File API (attempt 2)

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

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

Disadvantages? Expensive traversal! Goal: traverse once.

File Names

Three types of names:

- inode
- path
- file descriptor

File Descriptor (fd)

Idea: do traversal once, and store inode in **descriptor** object. Do reads/writes via descriptor. Also remember offset.

A file-descriptor **table** contains pointers to file descriptors.

The **integers** you're used to using for file I/O are indexes into this table.

FD Table (xv6)

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

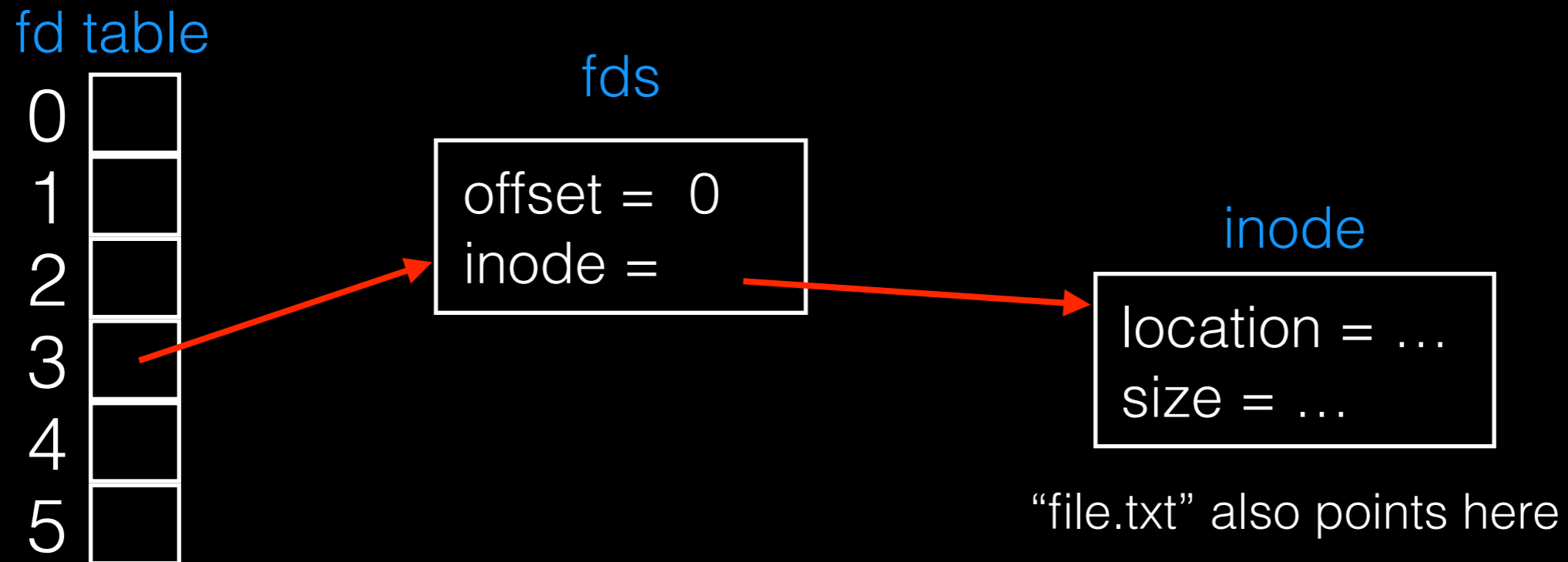
// Per-process state
struct proc {
    ...
    struct file *ofile[NOFILE]; // Open files
    ...
}
```

Code Snippet

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

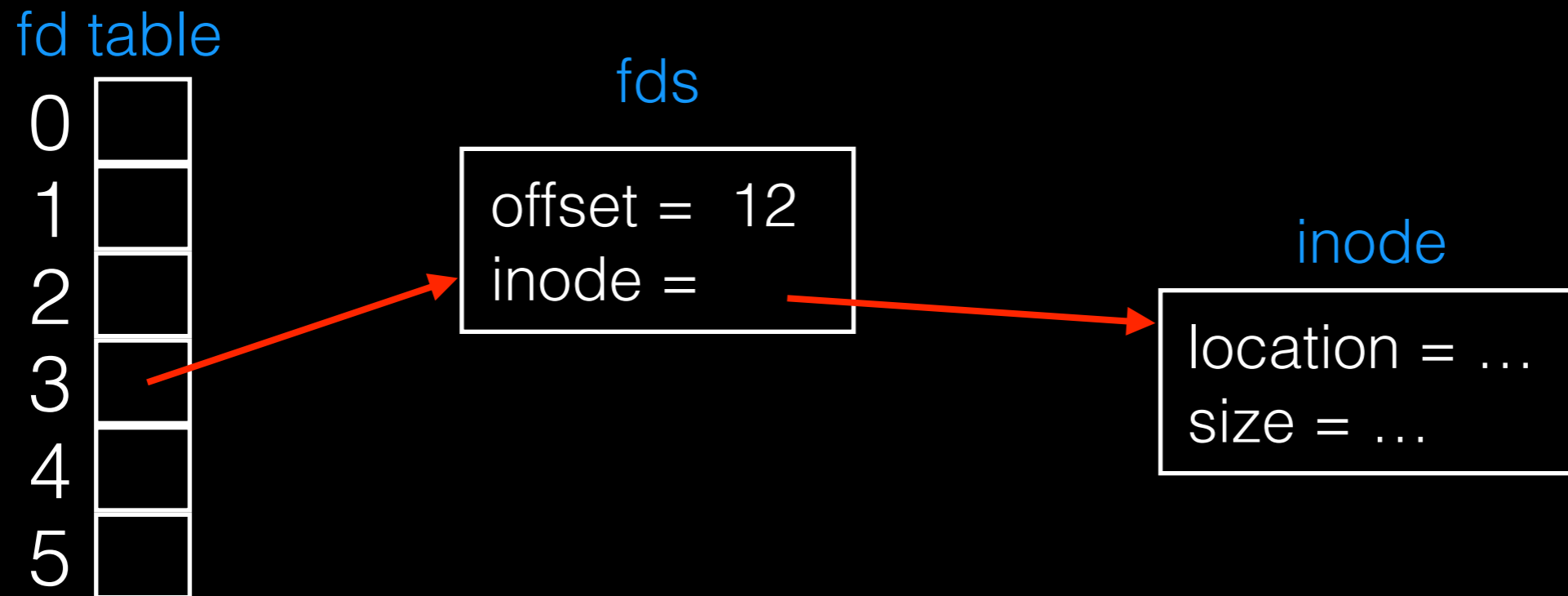
Code Snippet

```
int fd1 = open("file.txt"); // returns 3
```



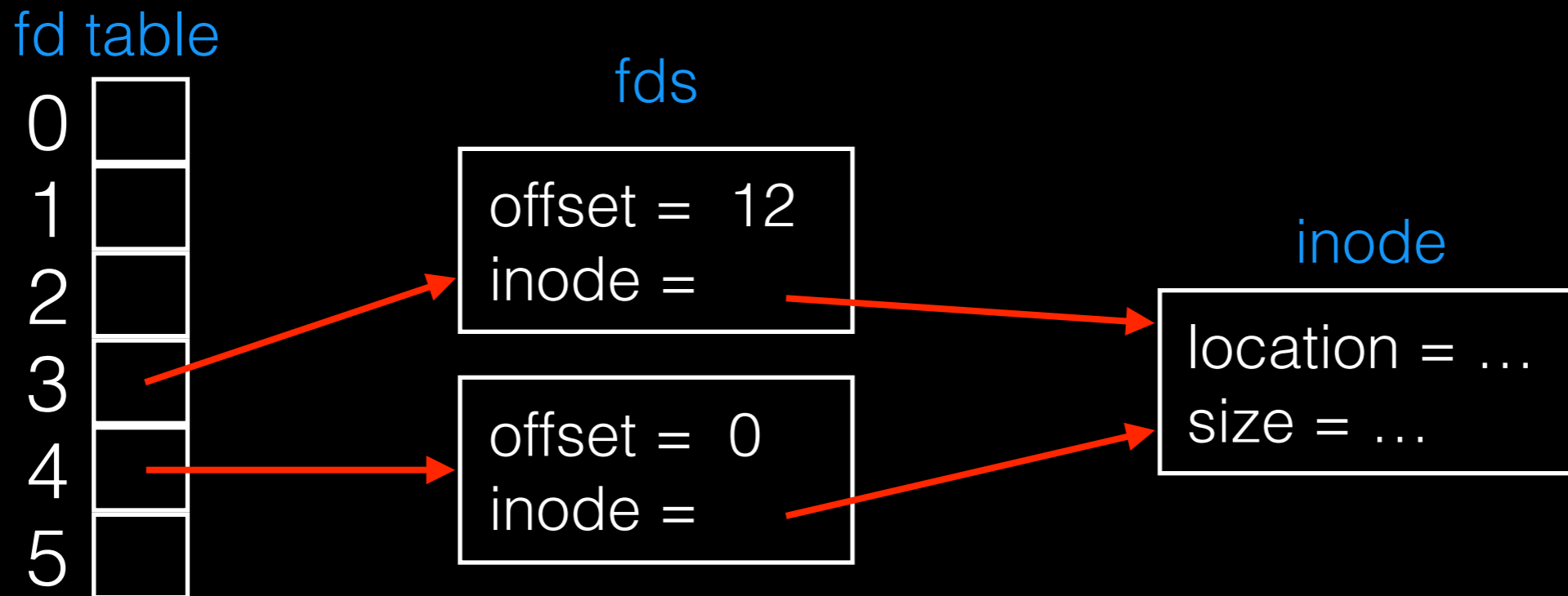
Code Snippet

```
int fd1 = open("file.txt"); // returns 3  
read(fd1, buf, 12);
```



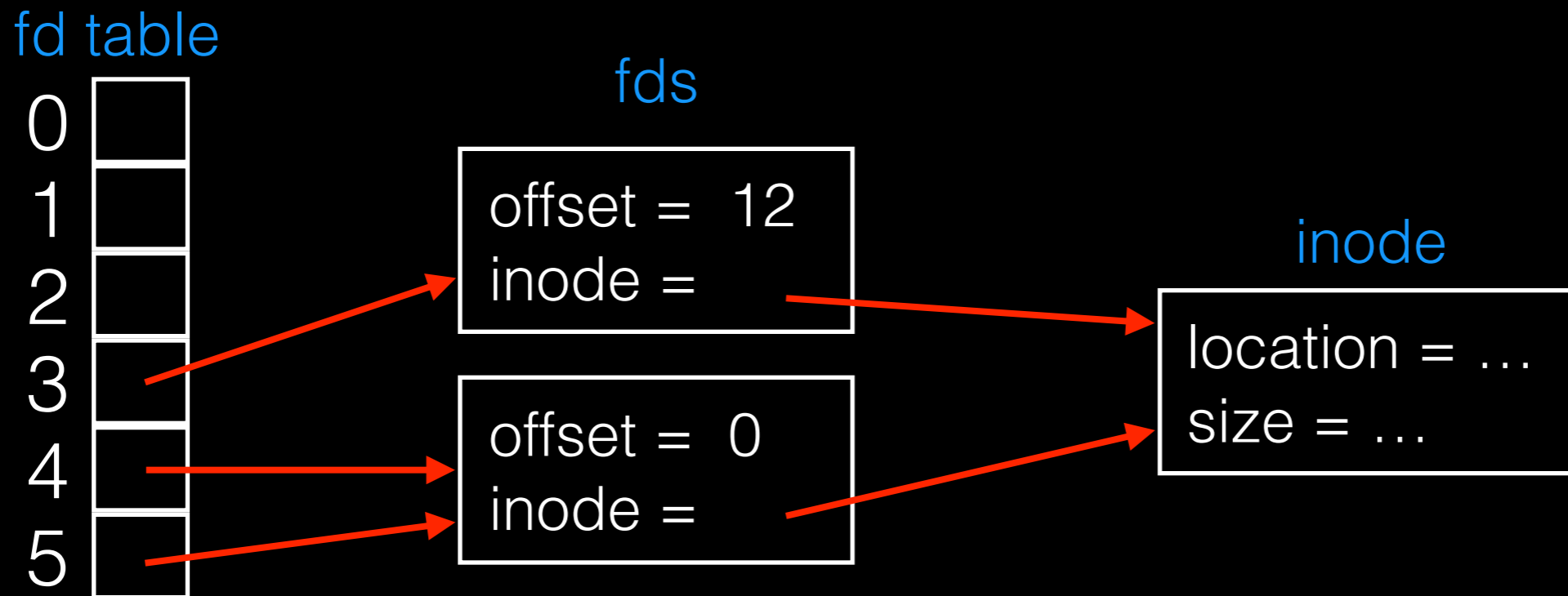
Code Snippet

```
int fd1 = open("file.txt"); // returns 3  
read(fd1, buf, 12);  
int fd2 = open("file.txt"); // returns 4
```



Code Snippet

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



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)
```


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
- different offsets

Deleting Files

There is no system call for deleting files!

Deleting Files

There is no system call for deleting files!

Inode (and associated file) is garbage collected when there are no references (from **paths** or **fds**).

Deleting Files

There is no system call for deleting files!

Inode (and associated file) is garbage collected when there are no references (from **paths** or **fds**).

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

FDs are deleted when: ???

Deleting Files

There is no system call for deleting files!

Inode (and associated file) is garbage collected when there are no references (from **paths** or **fds**).

Paths are deleted when: **unlink()** is called.

FDs are deleted when: **close()**, or process quits

Network File System Designers

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.*

Deleting Directories

Directories can also be unlinked with `unlink()`.
But only if empty!

How does “`rm -rf`” work?

Let's find out with **strace**!

```
void recursiveDelete(char* dirname) {
    char filename[FILENAME_MAX];
    DIR *dp = opendir (dirname);
    struct dirent *ep;
    while(ep = readdir (dp)) {
        snprintf(filename, FILENAME_MAX,
                 "%s/%s", dirname, ep->d_name);
        if(is_dir(ep))
            recursiveDelete(filename);
        else
            unlink(filename);
    }
    unlink(dirname);
}
```

my worst bug ever

Many File Systems

Many File Systems

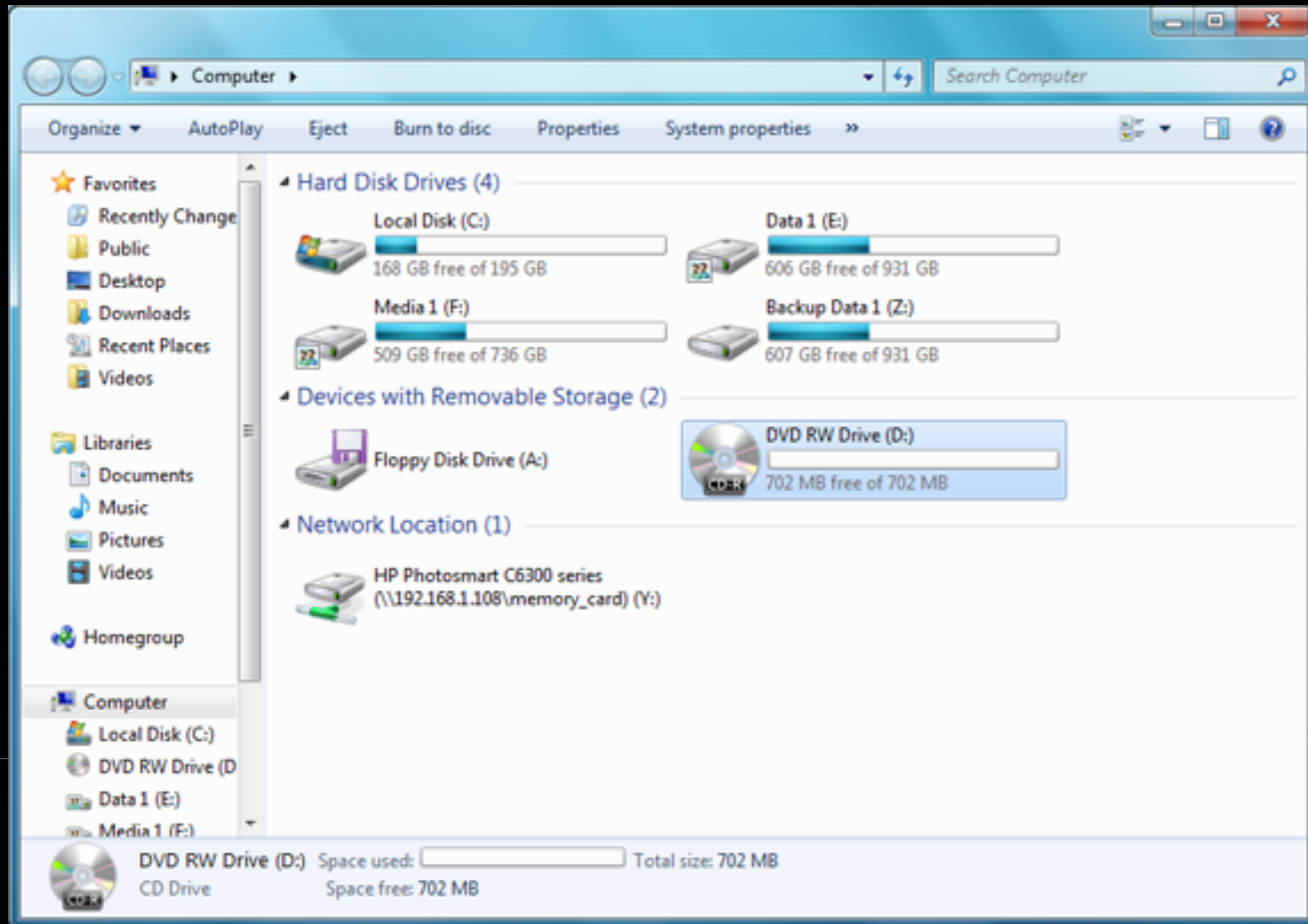
Users often want to use many file systems.

For example:

- main disk
- backup disk
- AFS
- thumb drives

What is the most **elegant** way to support this?

Many File Systems: Approach 1



Many File Systems: Approach 2

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

Many File Systems: Approach 2

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

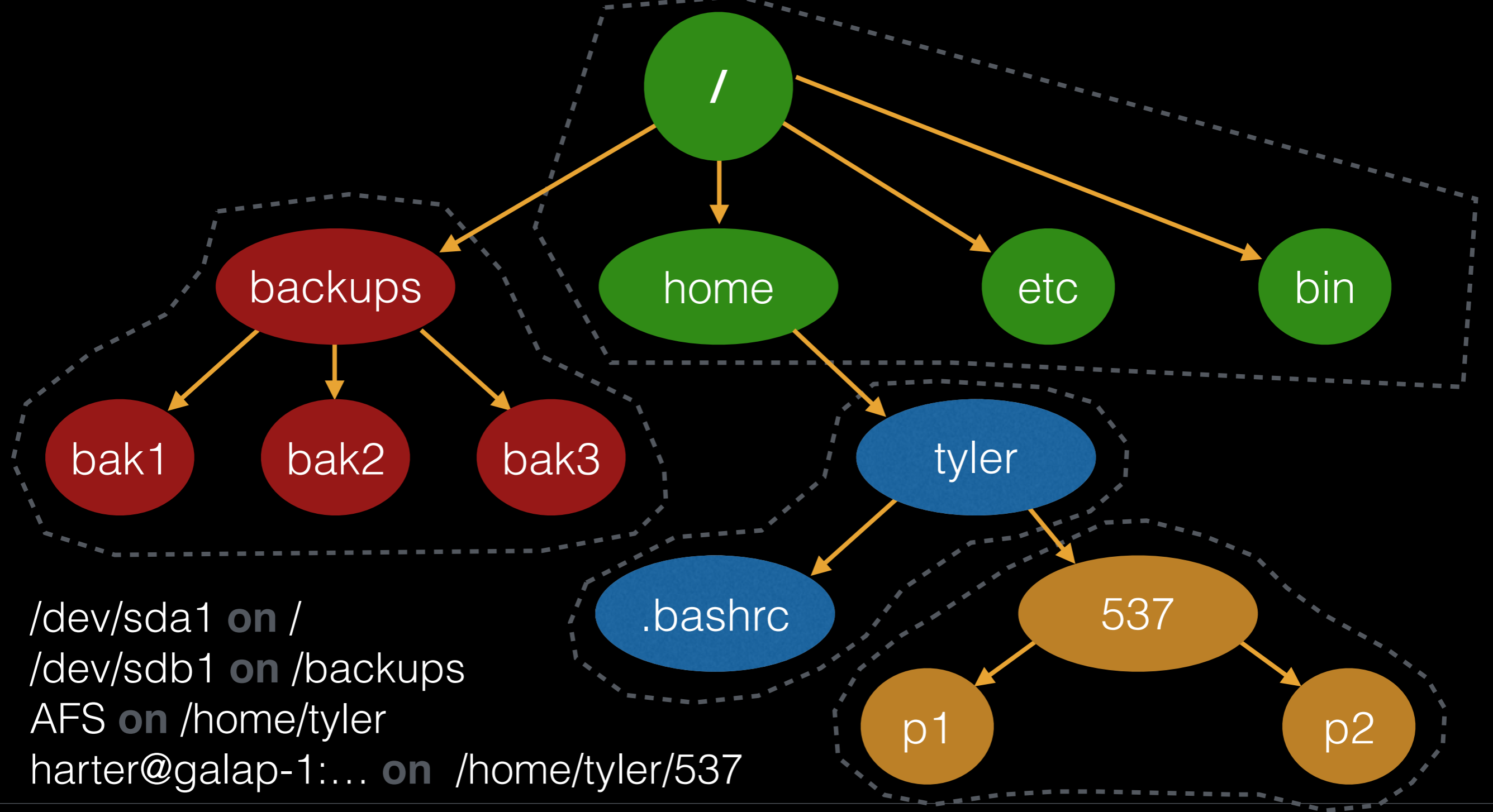
```
sh> mount
```

```
/dev/sda1 on / type ext4 (rw)
```

```
/dev/sdb1 on /backups type ext4 (rw)
```

```
AFS on /home/tyler type afs (rw)
```

```
harter@galap-1:~/537_projects /home/tyler/537 type sshfs (rw)
```



/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler
harter@galap-1:... **on** /home/tyler/537

Links: Demonstrate

Special Calls

fsync

Write buffering improves performance (why?).
But what if we **crash** before the buffers are flushed?

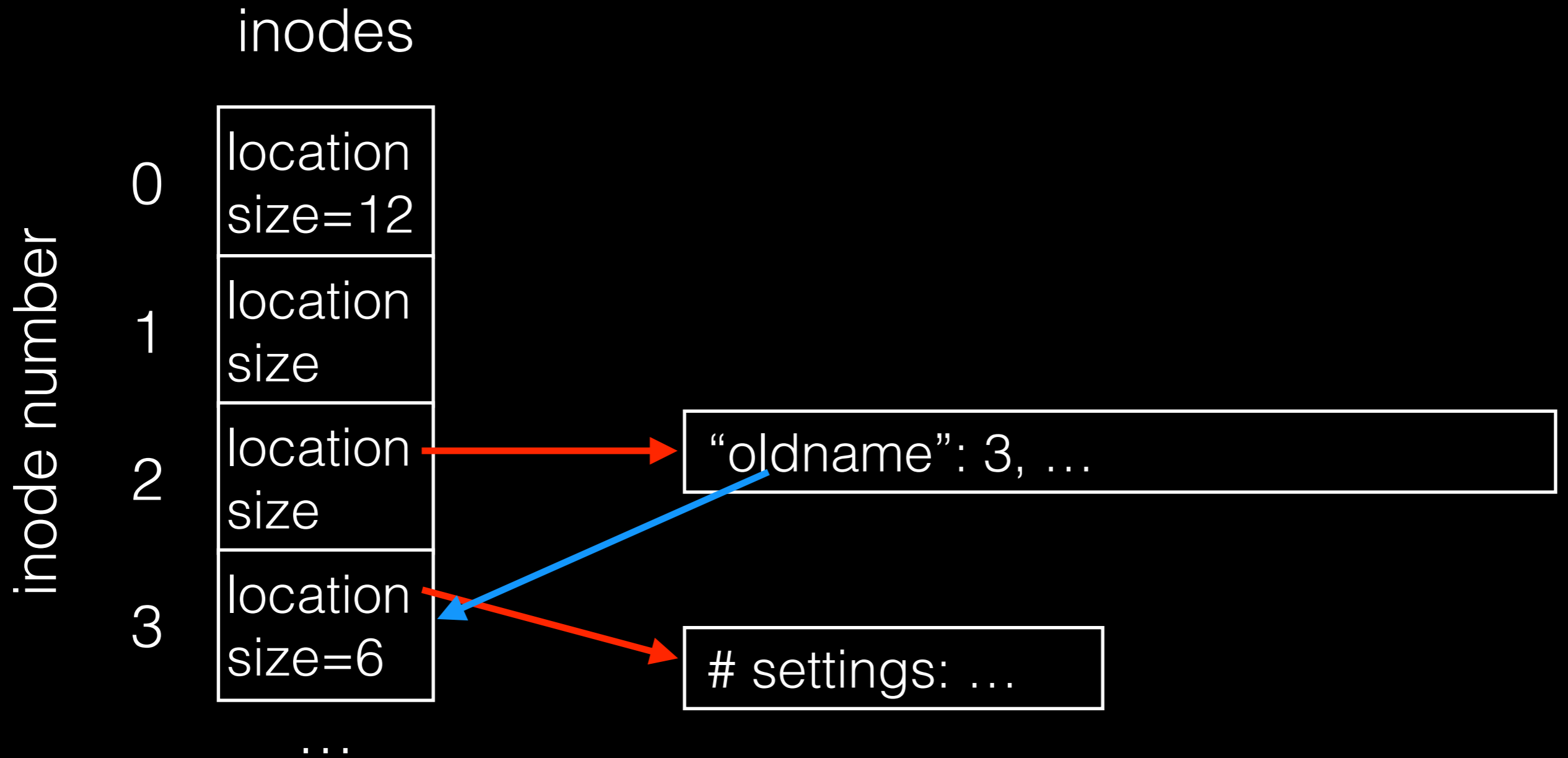
`fsync(int fd)` forces buffers to flush to disk, and
(usually) tells the disk to flush its write cache too.

This makes data **durable**.

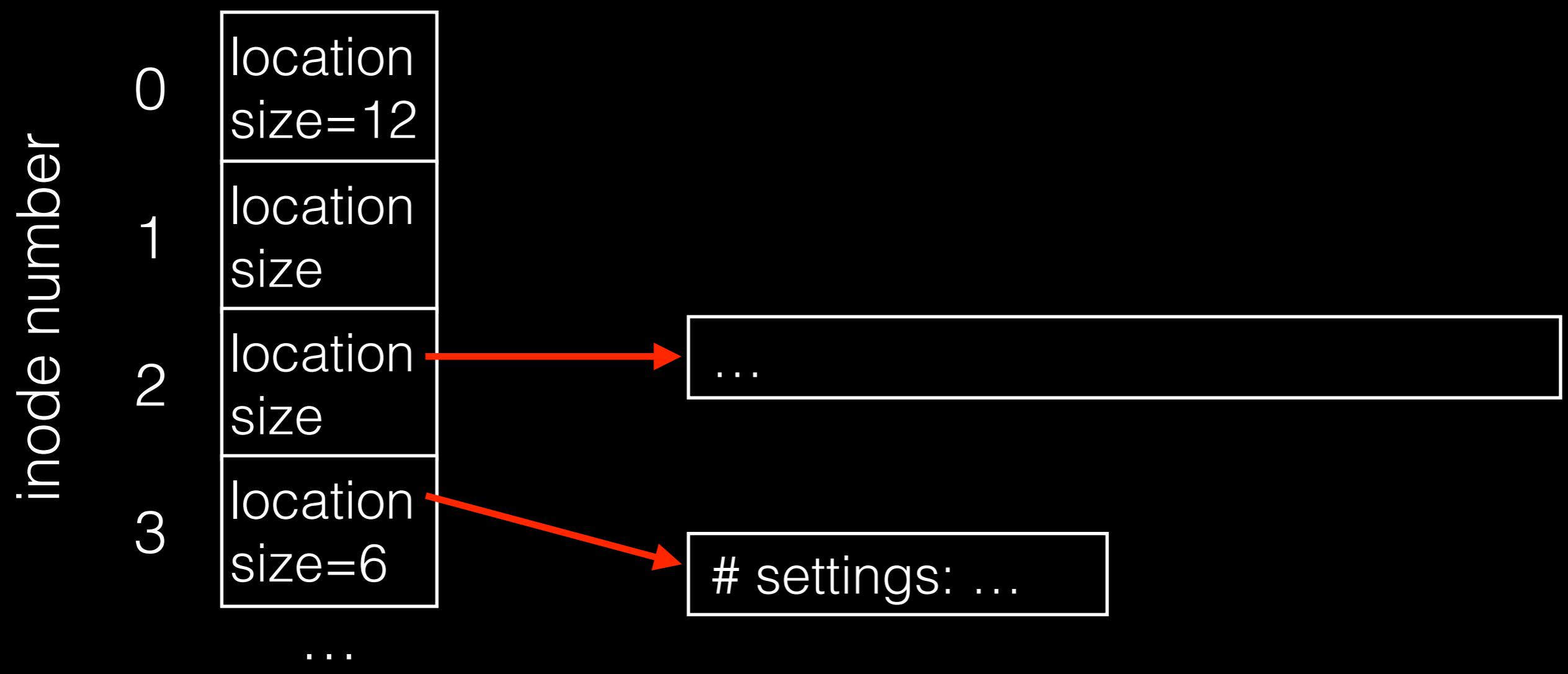
rename

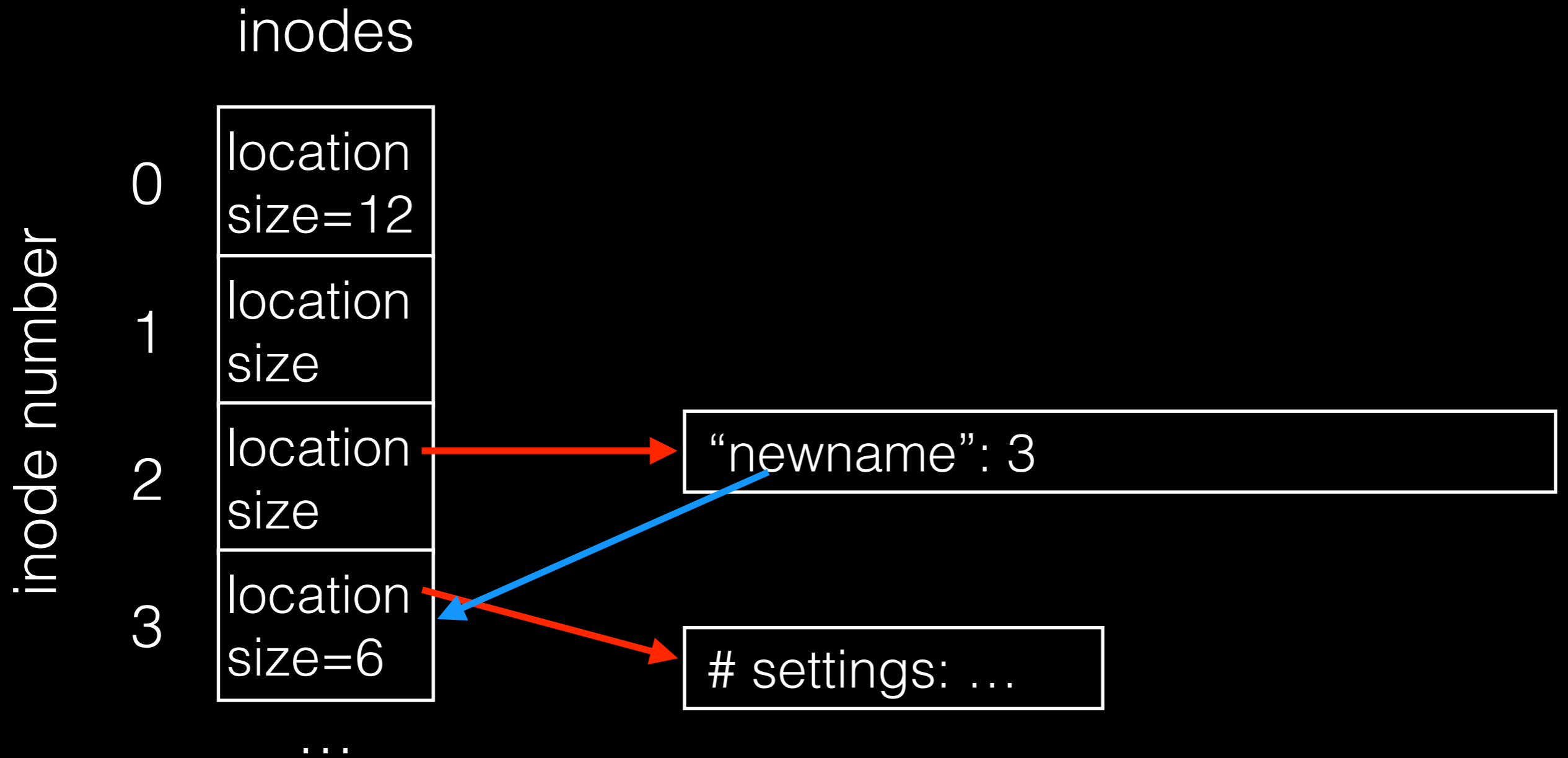
rename(char *old, char *new):

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



inodes





rename

rename(char *old, char *new):

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

rename

rename(char *old, char *new):

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

What if we crash?

rename

rename(char *old, char *new):

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

What if we crash?

FS does extra work to guarantee atomicity.

Atomic File Update

Say we want to update `file.txt`.

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

Concurrency

How can multiple processes avoid updating the same file at the **same time**?

Concurrency

How can multiple processes avoid updating the same file at the **same time**?

Normal locks don't work, as developers may have developed their programs independently.

Concurrency

How can multiple processes avoid updating the same file at the **same time**?

Normal locks don't work, as developers may have developed their programs independently.

Use **flock()**, for example:

- flock(fd, LOCK_EX)
- flock(fd, LOCK_UN)

Summary

Using multiple types of name provides

- convenience
- efficiency

Mount and link features provide flexibility.

Special calls (fsync, rename, flock) let developers communicate special requirements to FS.

Announcements

p4a and **p4b** are out!

- don't underestimate p4b.

Office hours now, in lab.