Operating Systems
Lecture 30: Filesystems

Nipun Batra
Nov 14, 2018

What is a File?

Array of bytes.

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

Inodes

- Each file has exactly one inode number.

Inodes

- Each file has exactly one inode number.

Inodes

- Each file has exactly one inode number.
- Inodes are unique (at a given time) within a FS.

Inodes

- Each file has exactly one inode number.
- Inodes are unique (at a given time) within a FS.

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`.

Inodes

Stat Unix utility

```
nipun@nipun-VirtualBox:~$ echo hello > abc.txt
nipun@nipun-VirtualBox:~$ cat abc.txt
hello
nipun@nipun-VirtualBox:~$ stat abc.txt
  File: abc.txt
  Size: 6           Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 440161      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   nipun)   Gid: ( 1000/   nipun)
Access: 2018-11-14 14:15:28.122932148 +0530
Modify: 2018-11-14 14:15:23.228486149 +0530
Change: 2018-11-14 14:15:23.228486149 +0530
 Birth: -
```

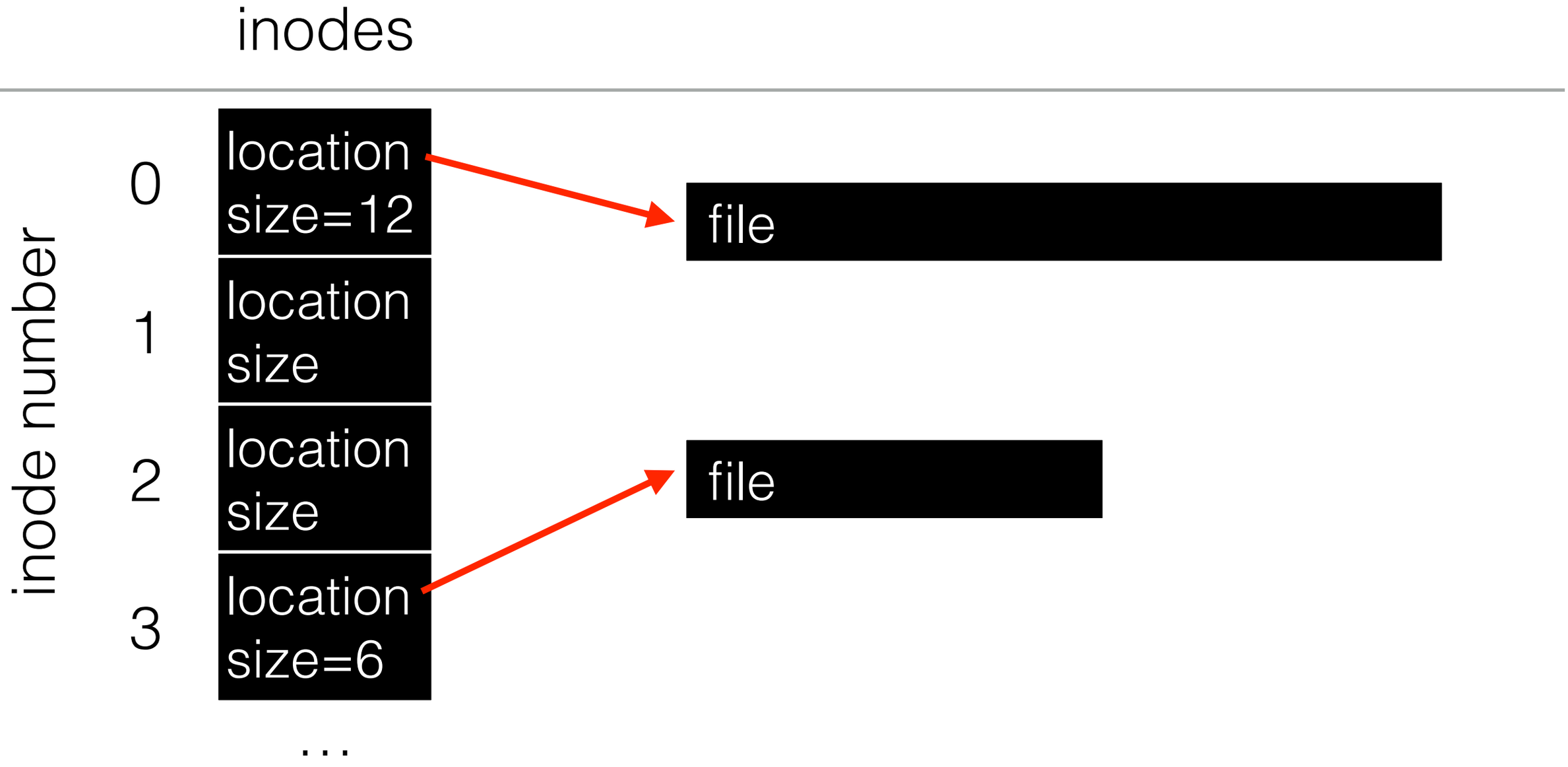
Command Line Demo

Demo1.sh

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
inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
...
file
file

File API (attempt 1)

```
read(int inode, void *buf, size_t nbyte)

write(int inode, void *buf, size_t nbyte)
```

File API (attempt 1)

```
read(int inode, void *buf, size_t nbyte)

write(int inode, void *buf, size_t nbyte)
```

Disadvantages?

- names hard to remember
- no way to change offset

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)
```

File Names

Three types of names:

- inode
- path
- file descriptor

Paths

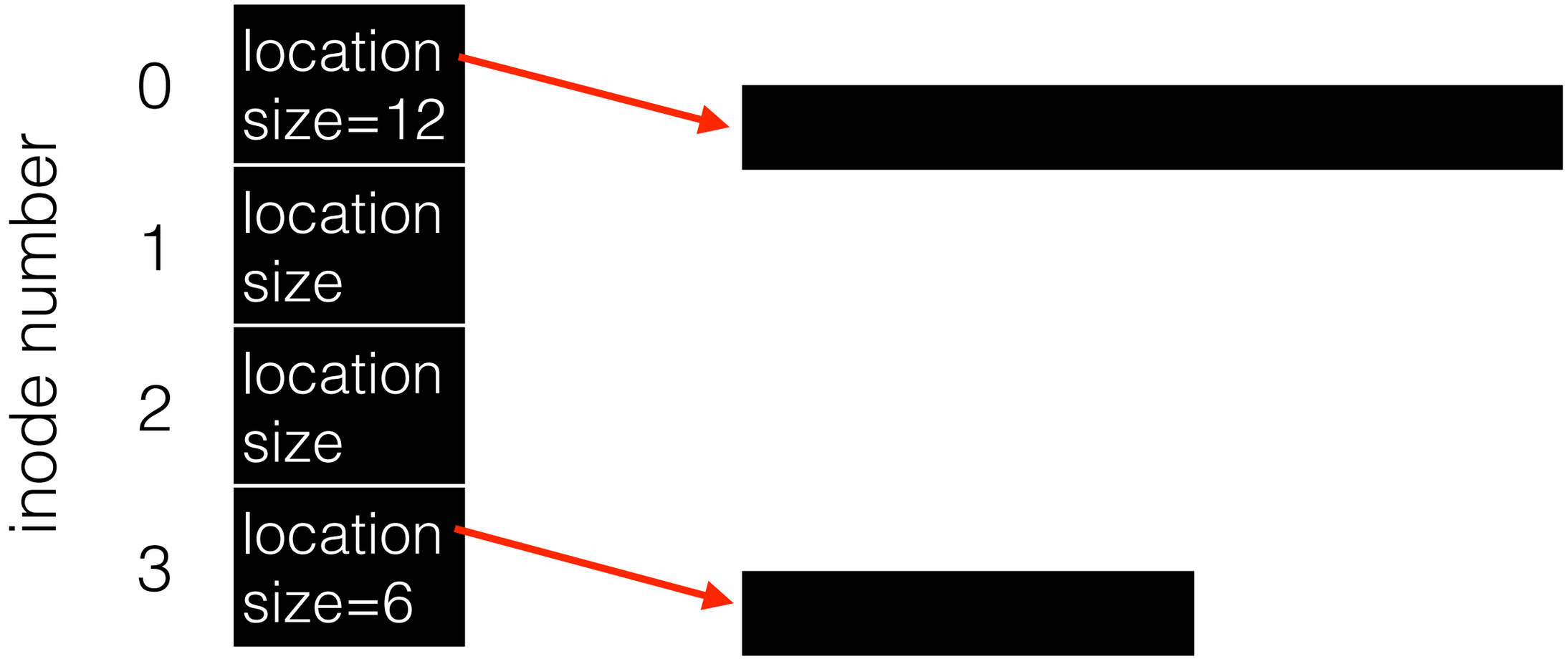
String names are friendlier than number names.

Paths

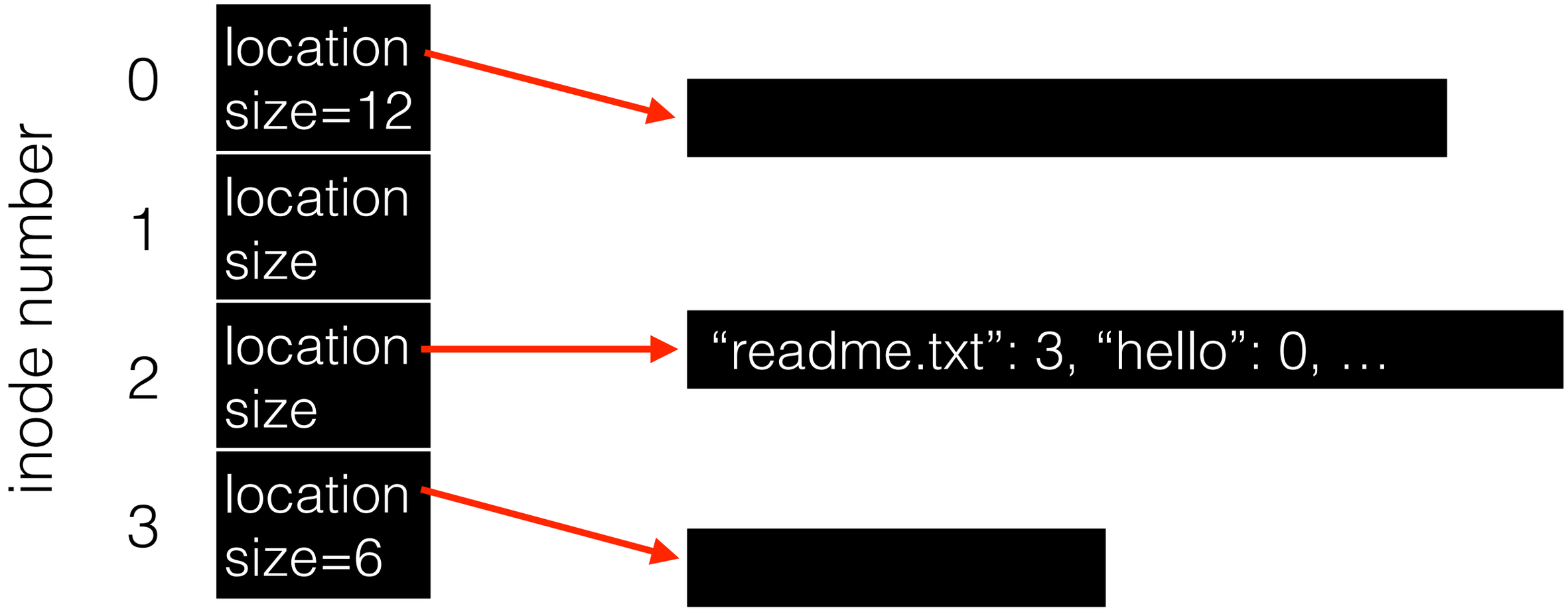
String names are friendlier than number names.

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

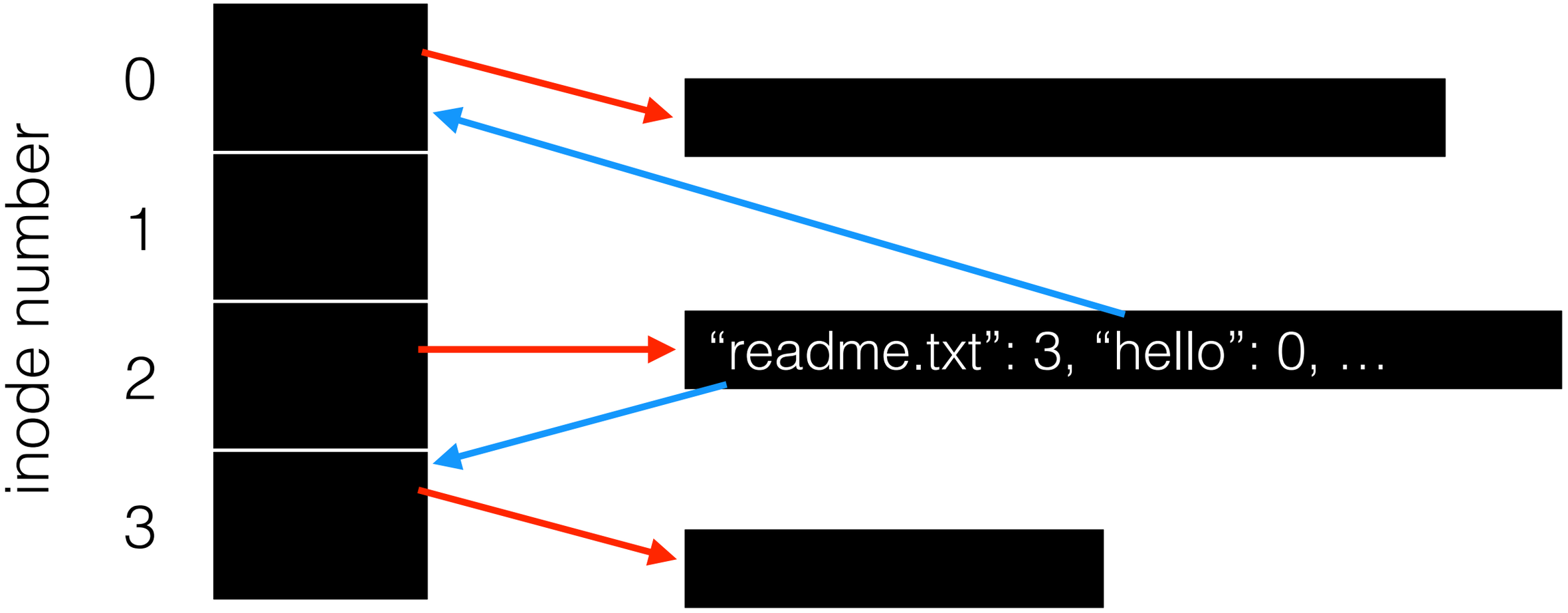
Paths


inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6

Paths


inode number
0
location
size=12
1
location
size
2
location
size
3
location
size=6
“readme.txt”: 3, “hello”: 0, …

Paths



Paths

String names are friendlier than number names.

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

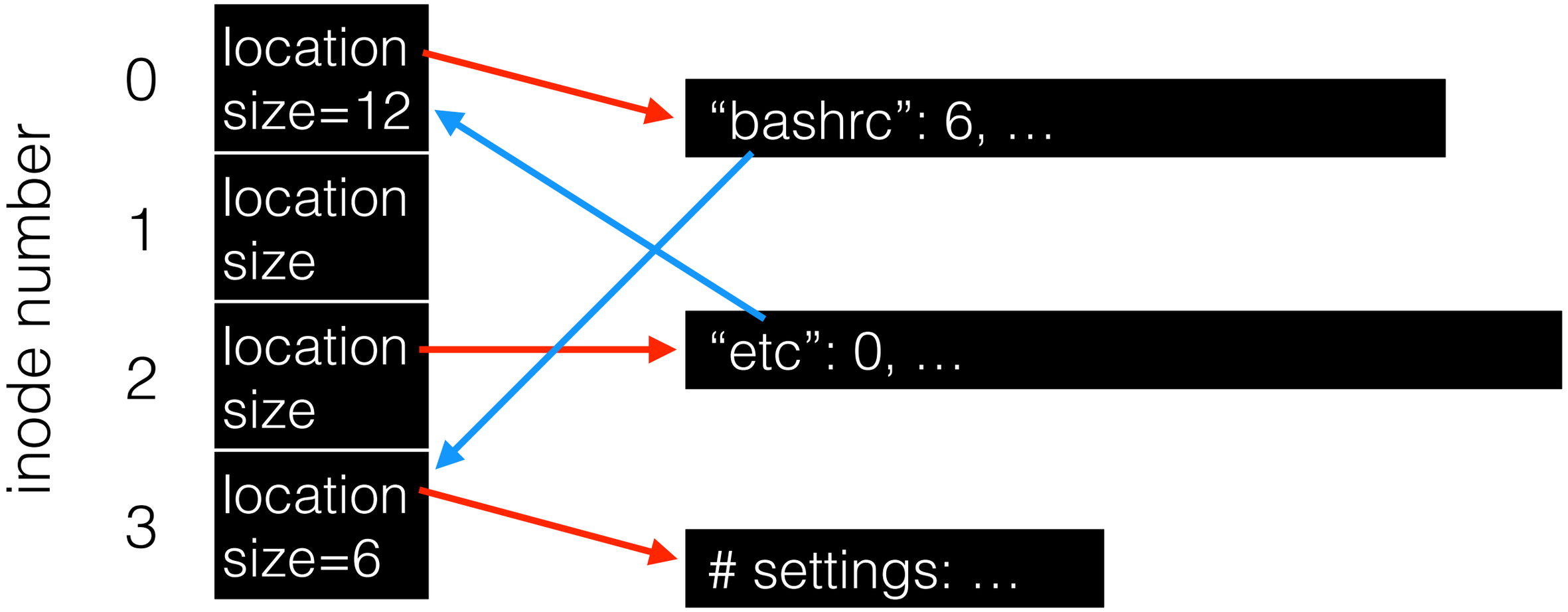
Paths

String names are friendlier than number names.

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

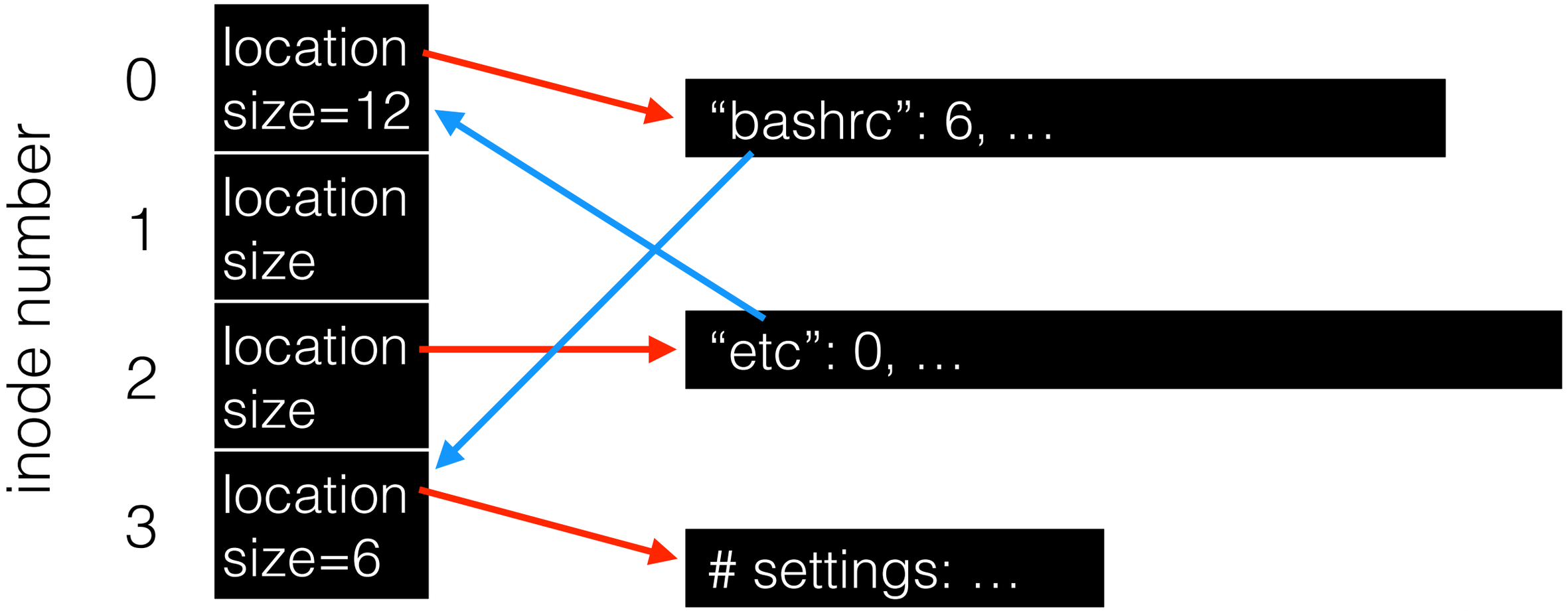
Generalize! Store path-to-inode mapping in many files. Call these special files directories.

Directories and Files



Directories and Files

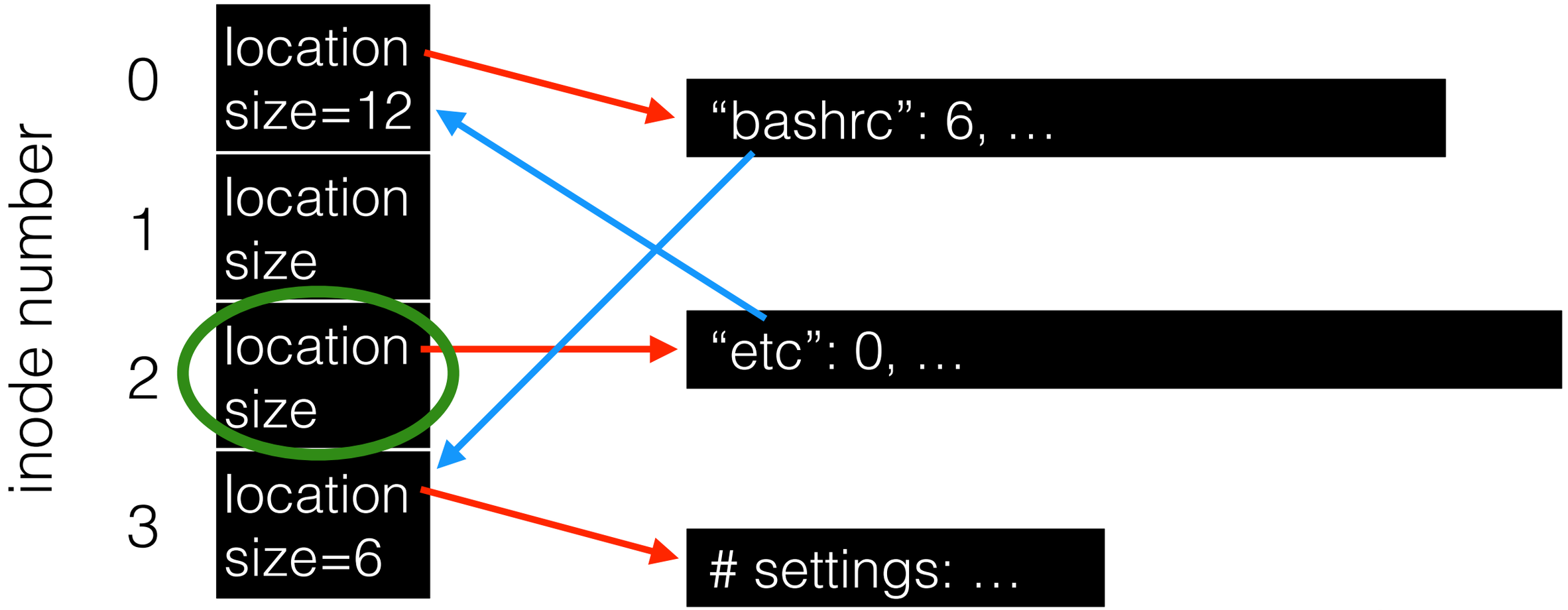
read **/etc/bashrc**



reads: 0

Directories and Files

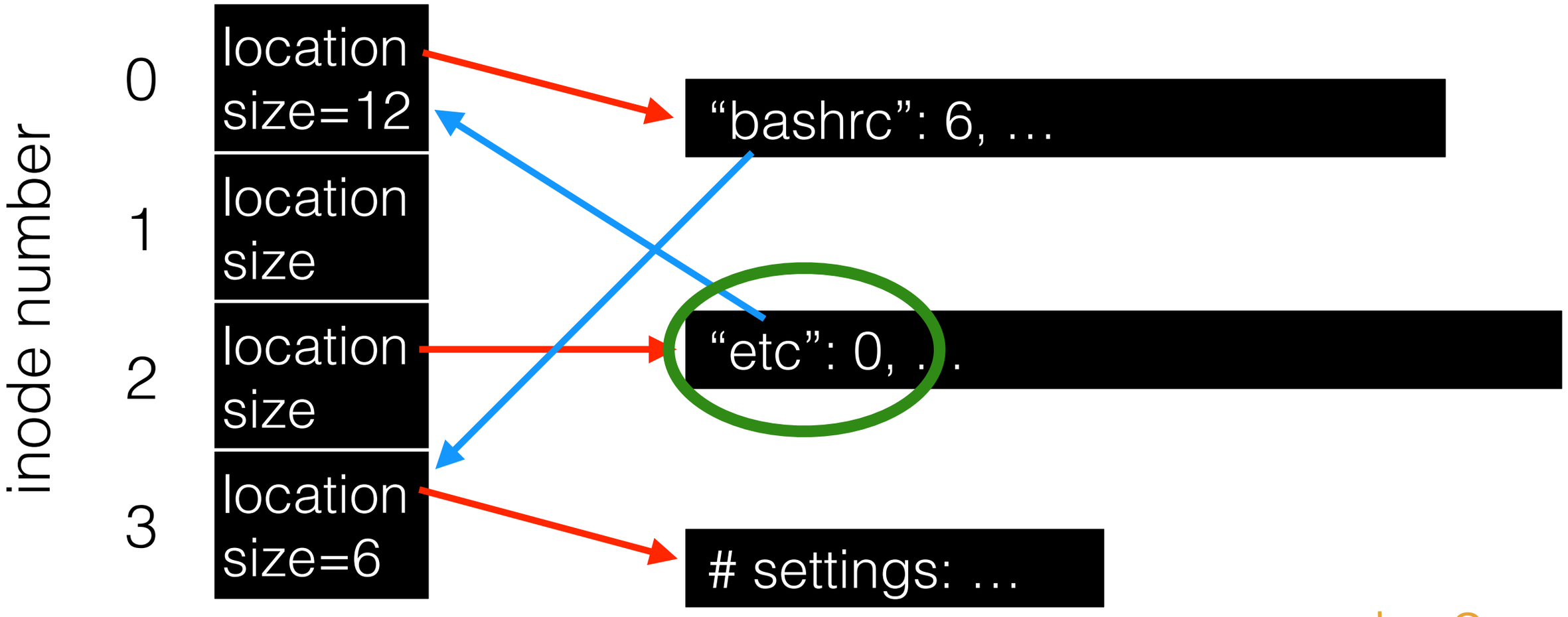
read **/etc/bashrc**



reads: 1

Directories and Files

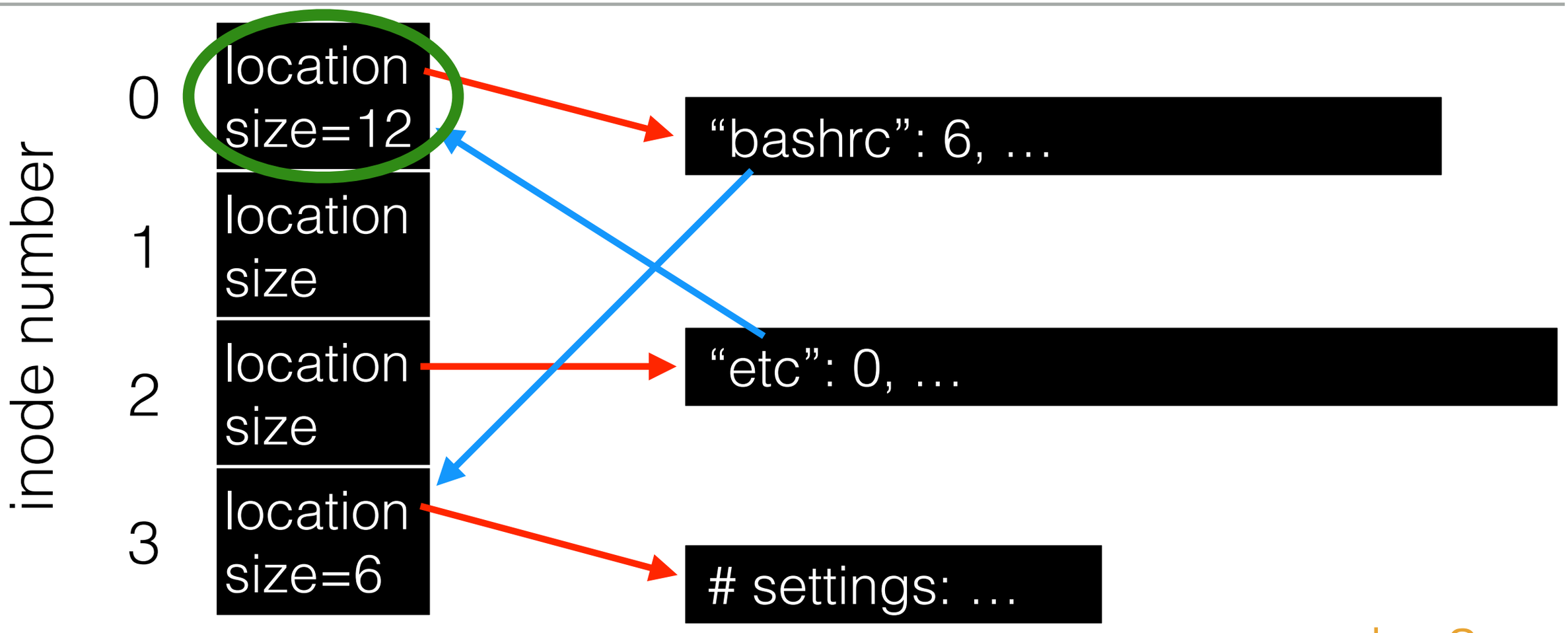
read **/etc/bashrc**



reads: 2

Directories and Files

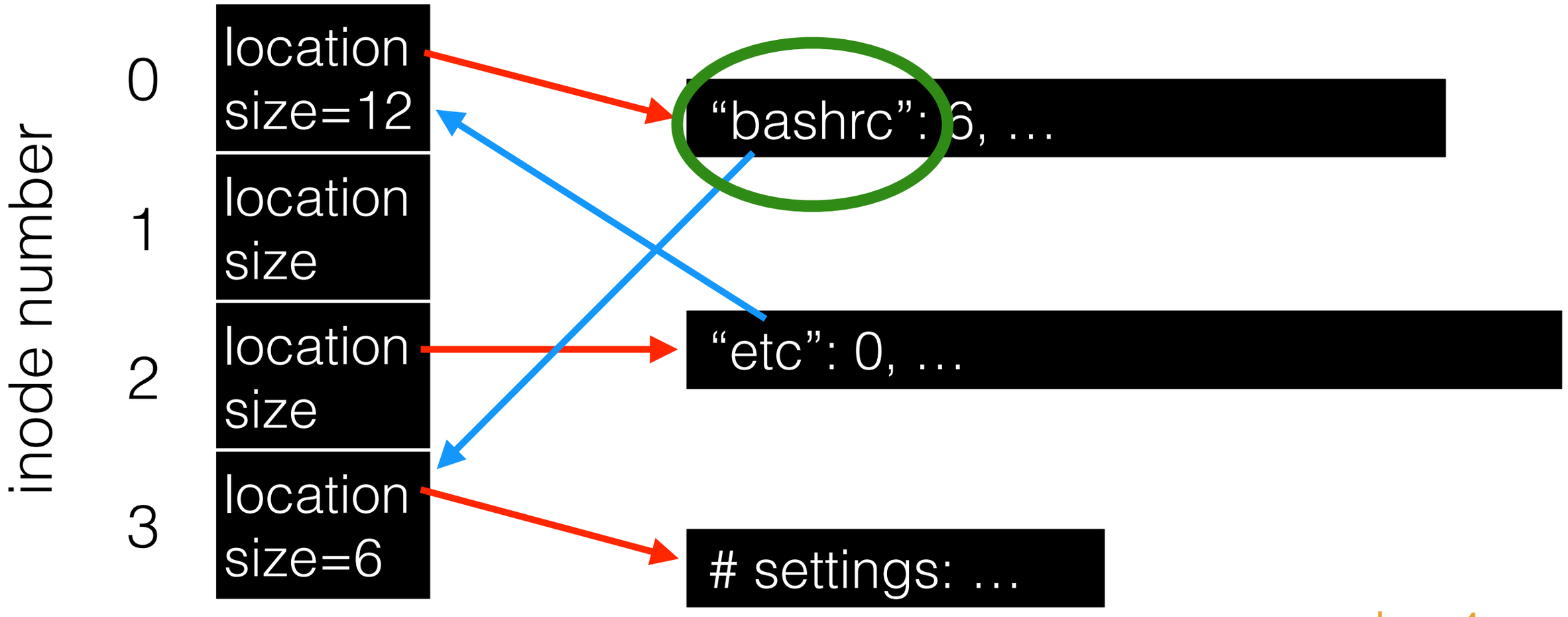
read **/etc/bashrc**



reads: 3

Directories and Files

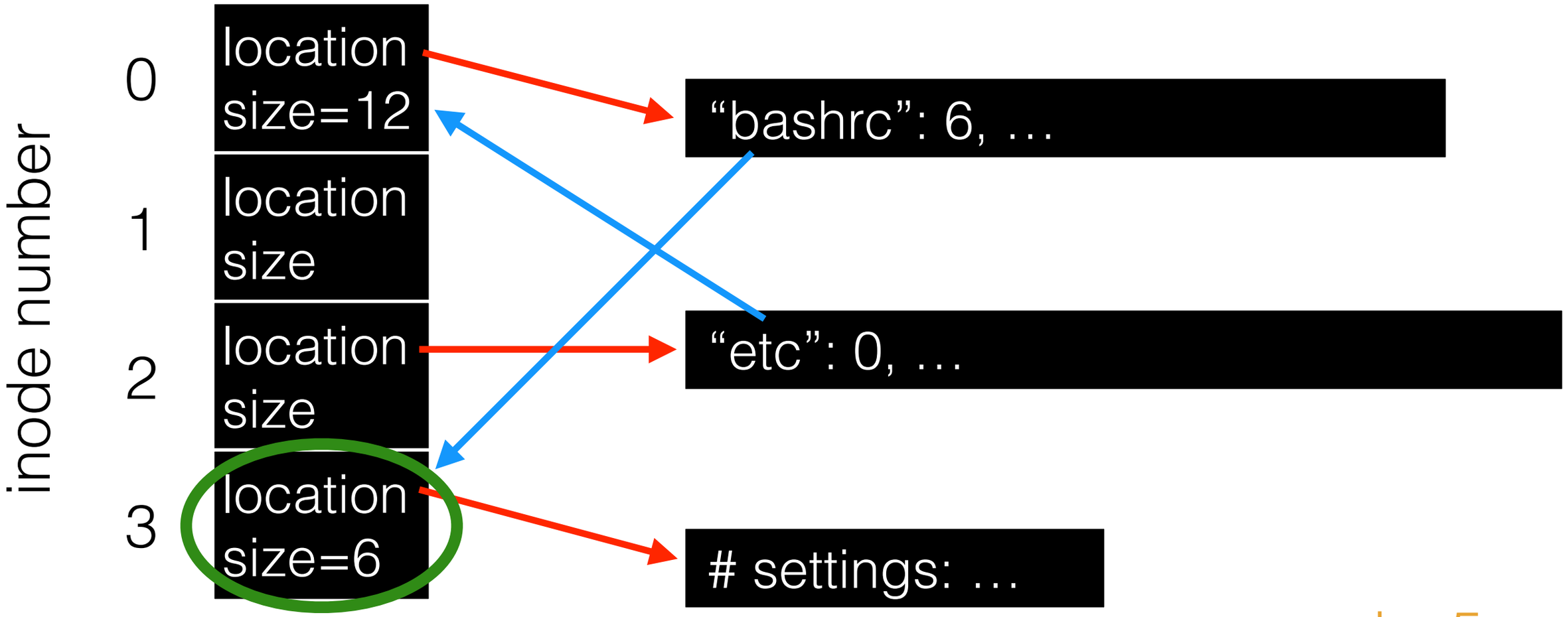
read **/etc/bashrc**



reads: 4

Directories and Files

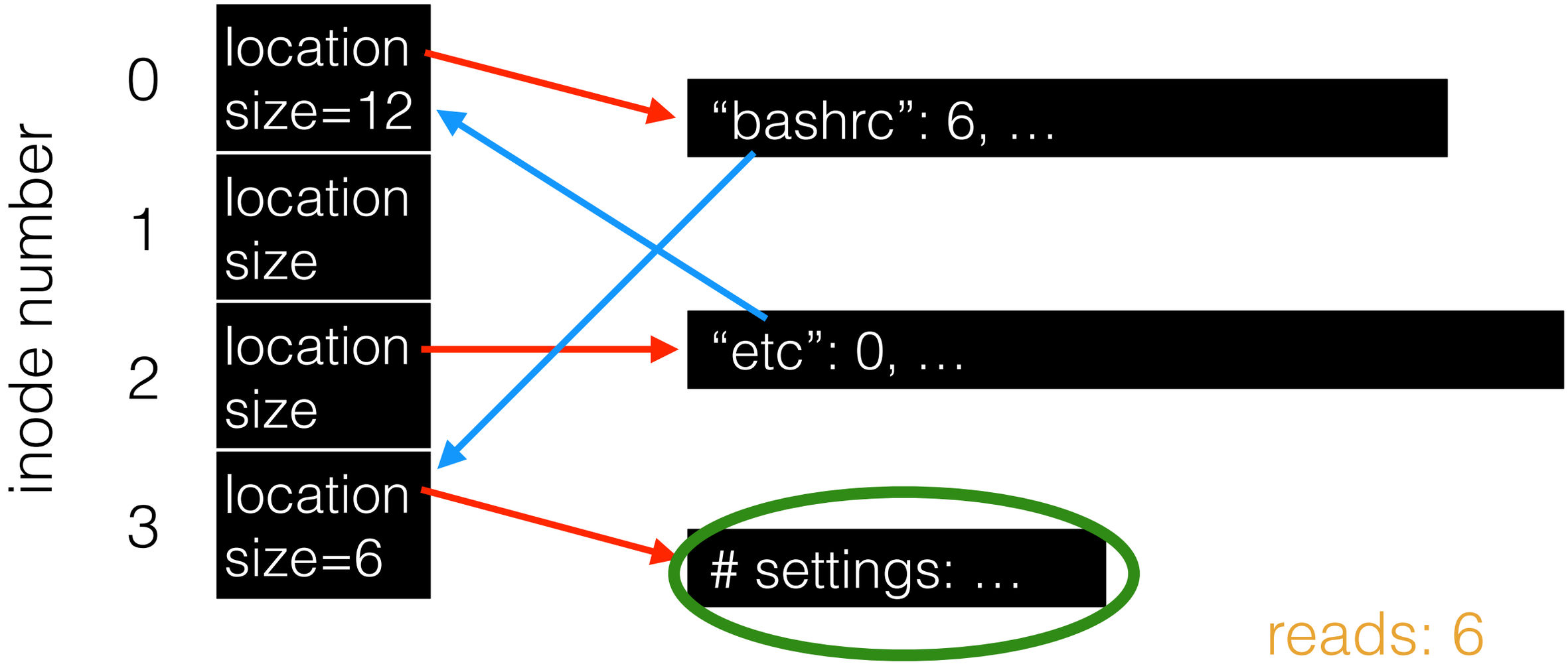
read **/etc/bashrc**



reads: 5

Directories and Files

read **/etc/bashrc**



reads: 6

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?

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)

File Descriptor (fd)

- Idea: do traversal once, and store inode in descriptor object. Do reads/writes via descriptor.

File Descriptor (fd)

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

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.

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)

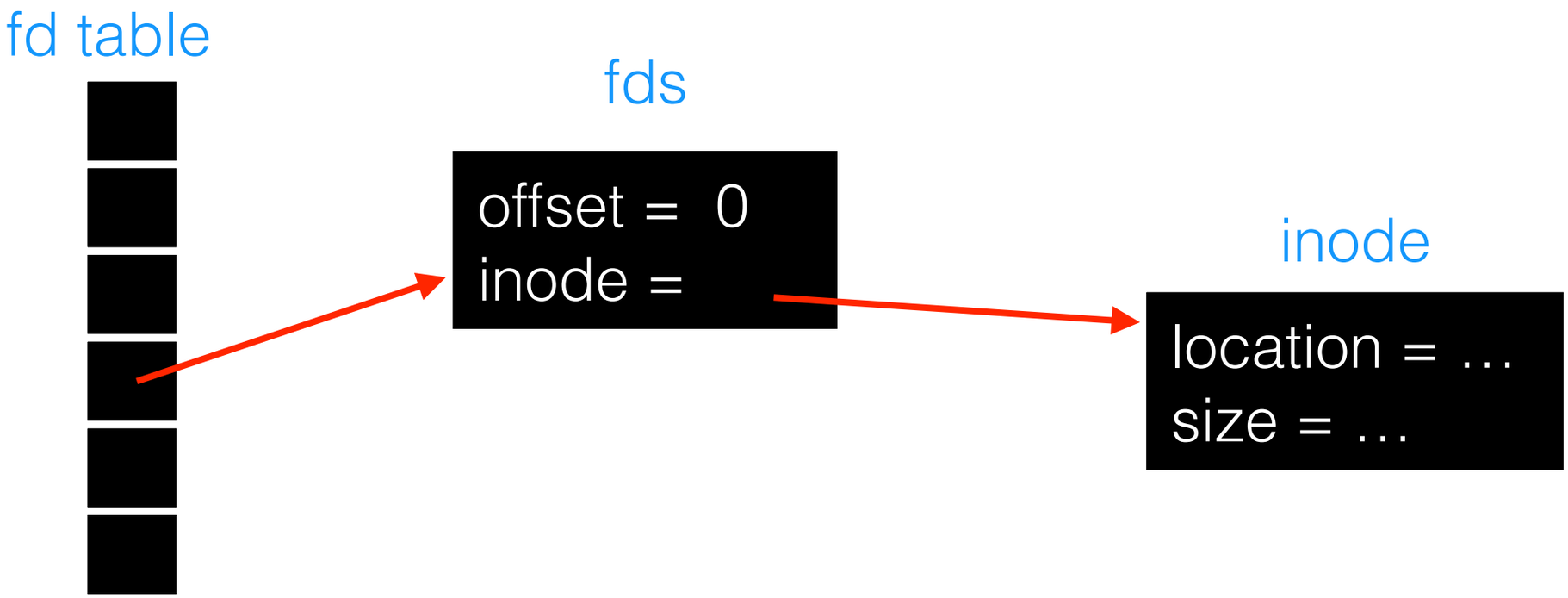
```
struct file

  struct inode  ip
  uint off                                  //Offset};

// Per-process state
struct proc

  struct file  ofile[N]                     // 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
```

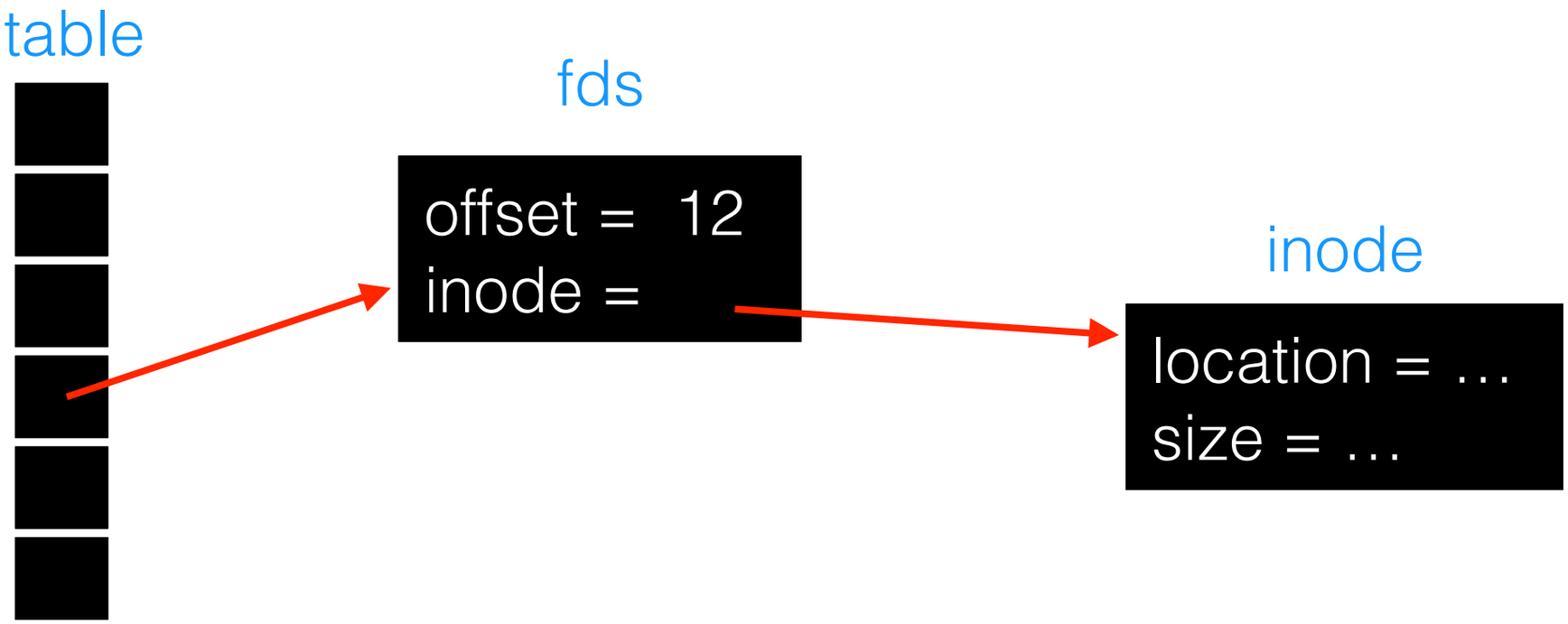
Code Snippet

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



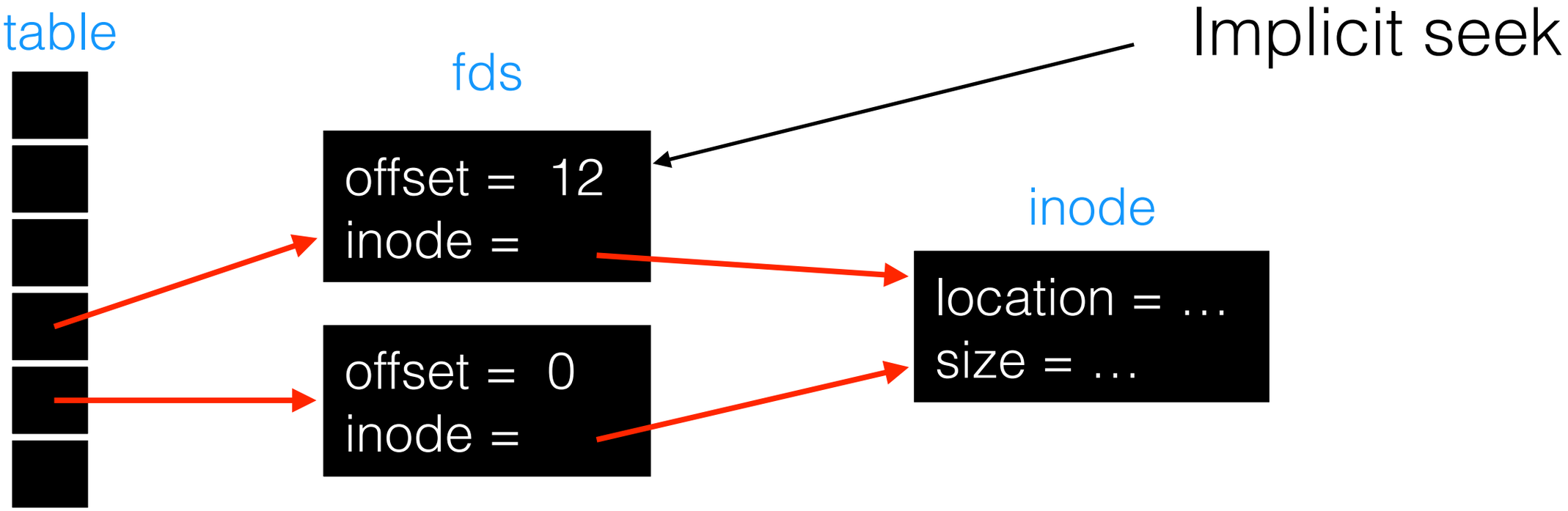
Code Snippet

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



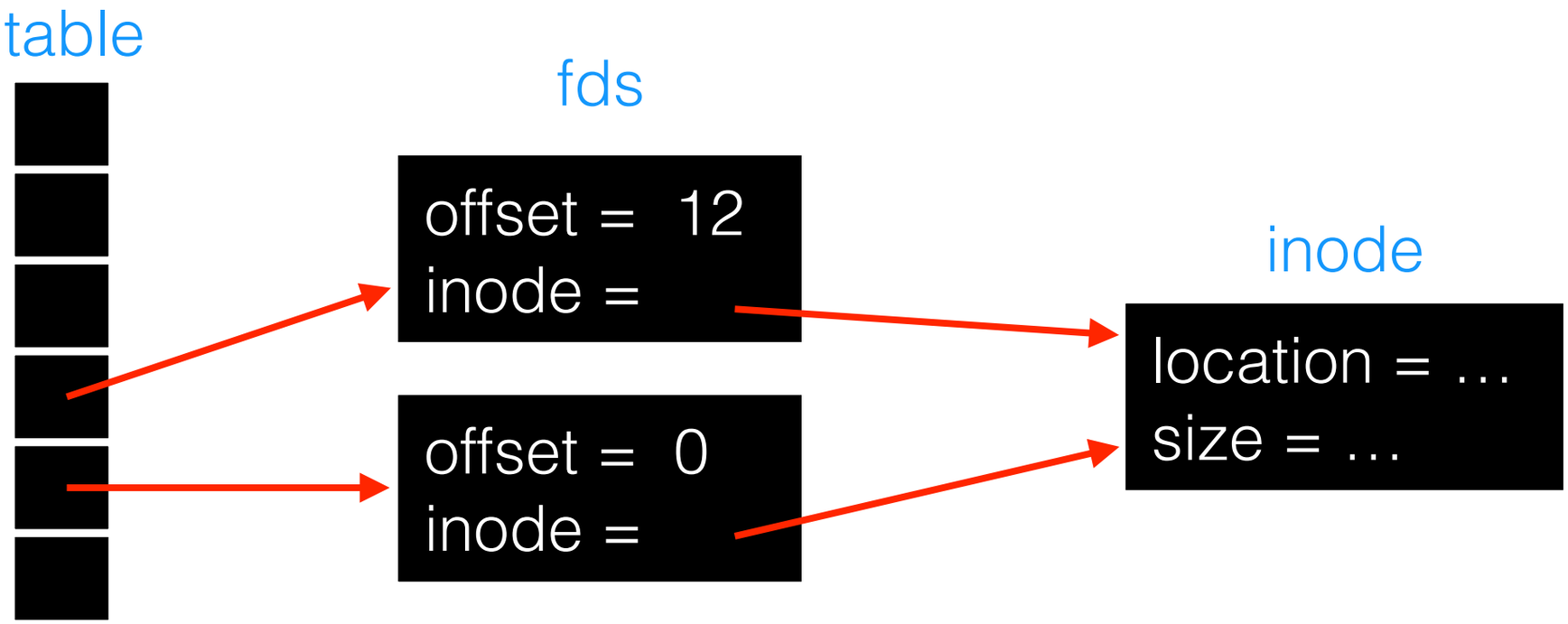
Code Snippet

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



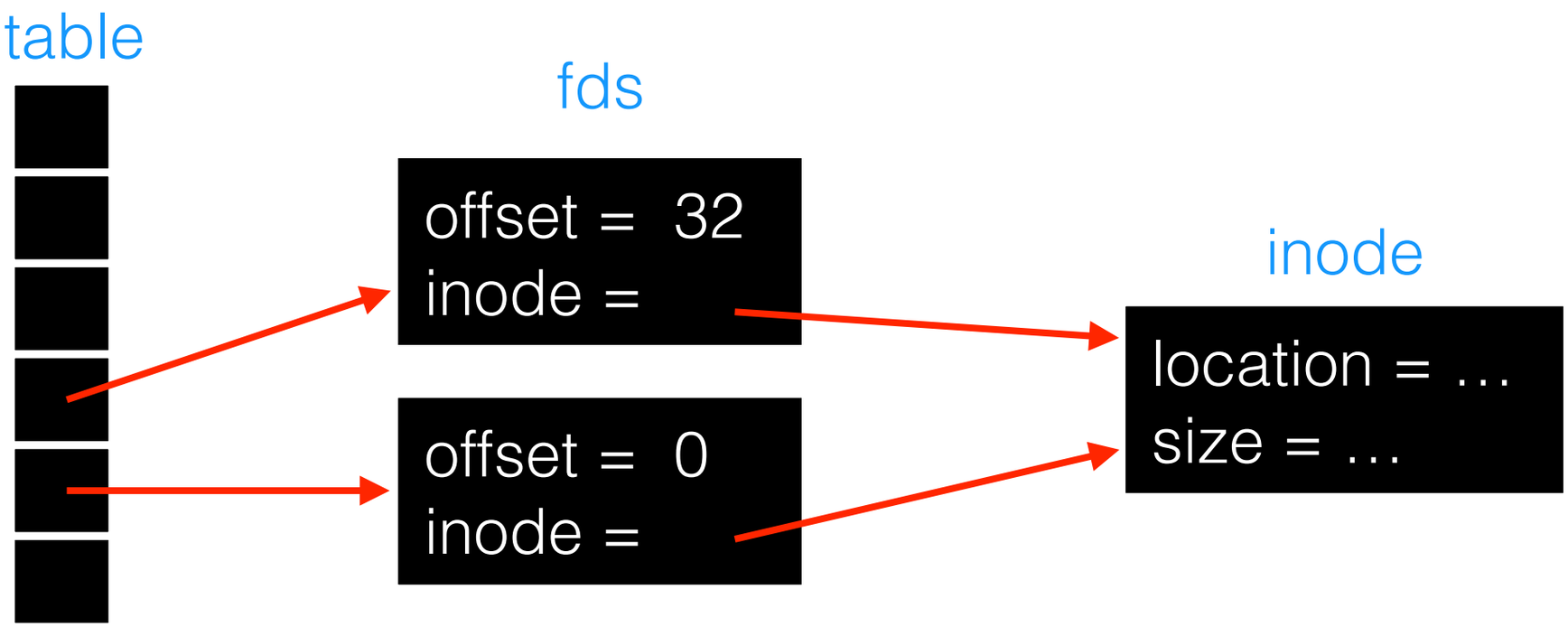
Explicit Seek (use LSEEK)

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



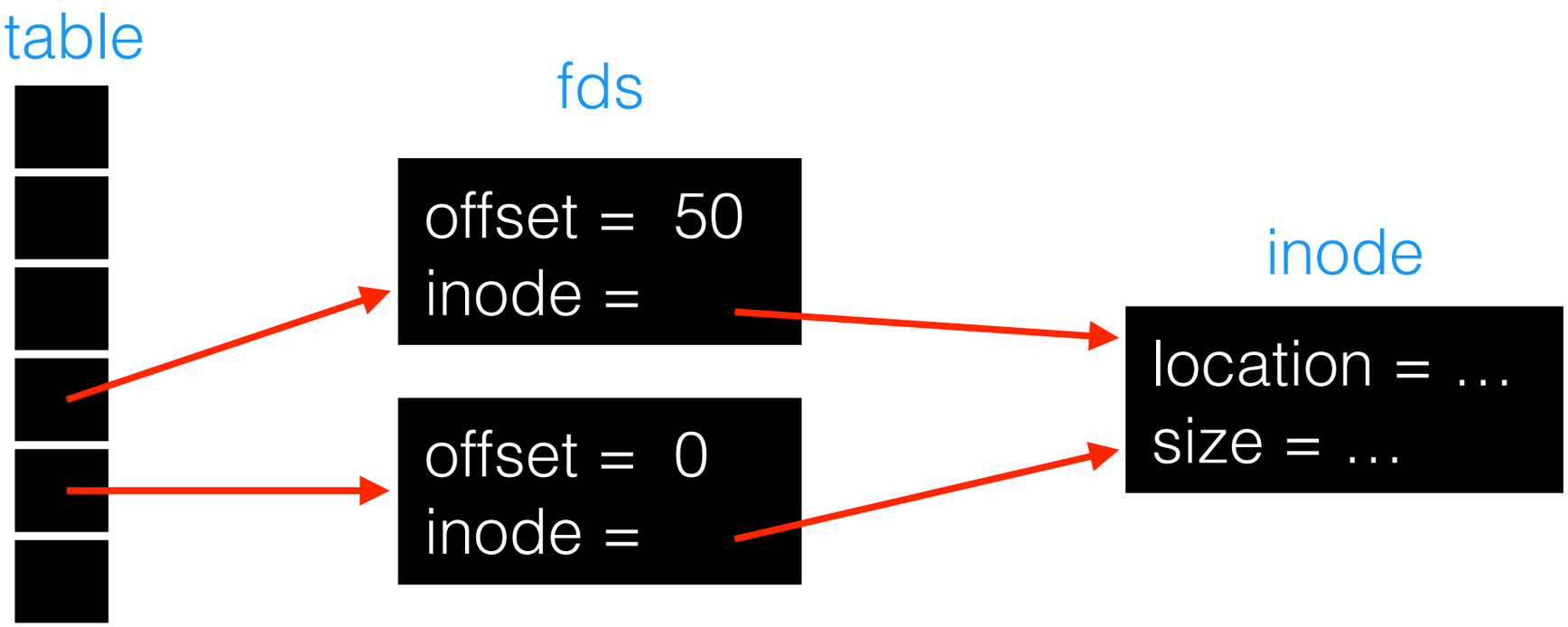
Explicit Seek (use LSEEK)

```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
lseek(fd1, 20, SEEK_SET);
```



Explicit Seek (use LSEEK)

```
int fd1 = open(“file.txt”); // returns 3
read(fd1, buf, 12);
lseek(fd1, 20, SEEK_SET);
lseek(fd1, 30, SEEK_CUR);
```



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
- traverse once
- different offsets

Strace on Common Operations

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)  = 3
read(3, "hello\n", 4096)     = 6
write(1, "hello\n", 6)  = 6 // file descriptor 1: standard out
hello
read(3, "", 4096)       = 0 // 0: no bytes left in the file
close(3)
...
prompt>
```

Demo2.sh

fsync

fsync

- Write buffering improves performance (why?).

fsync

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

fsync

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

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 it's write cache too.

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 it's write cache too.

fsync

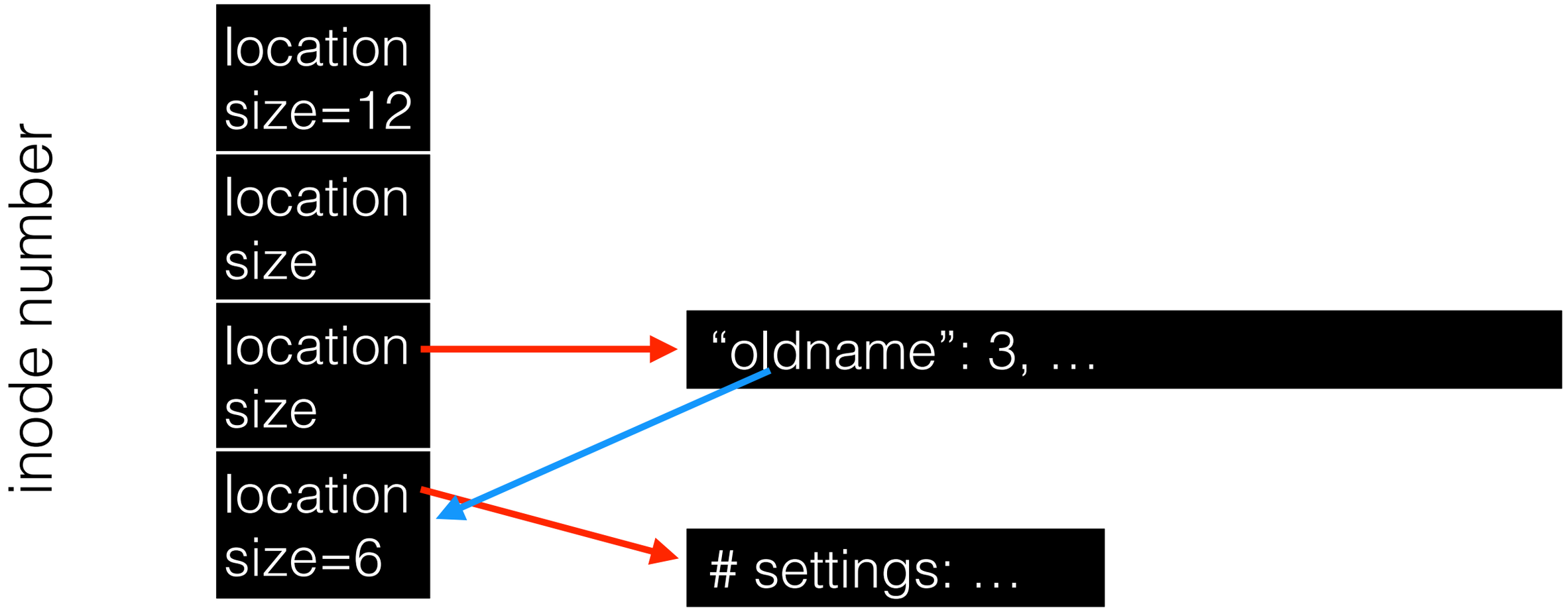
- Write buffering improves performance (why?).
- But what if we crash before the buffers are flushed?

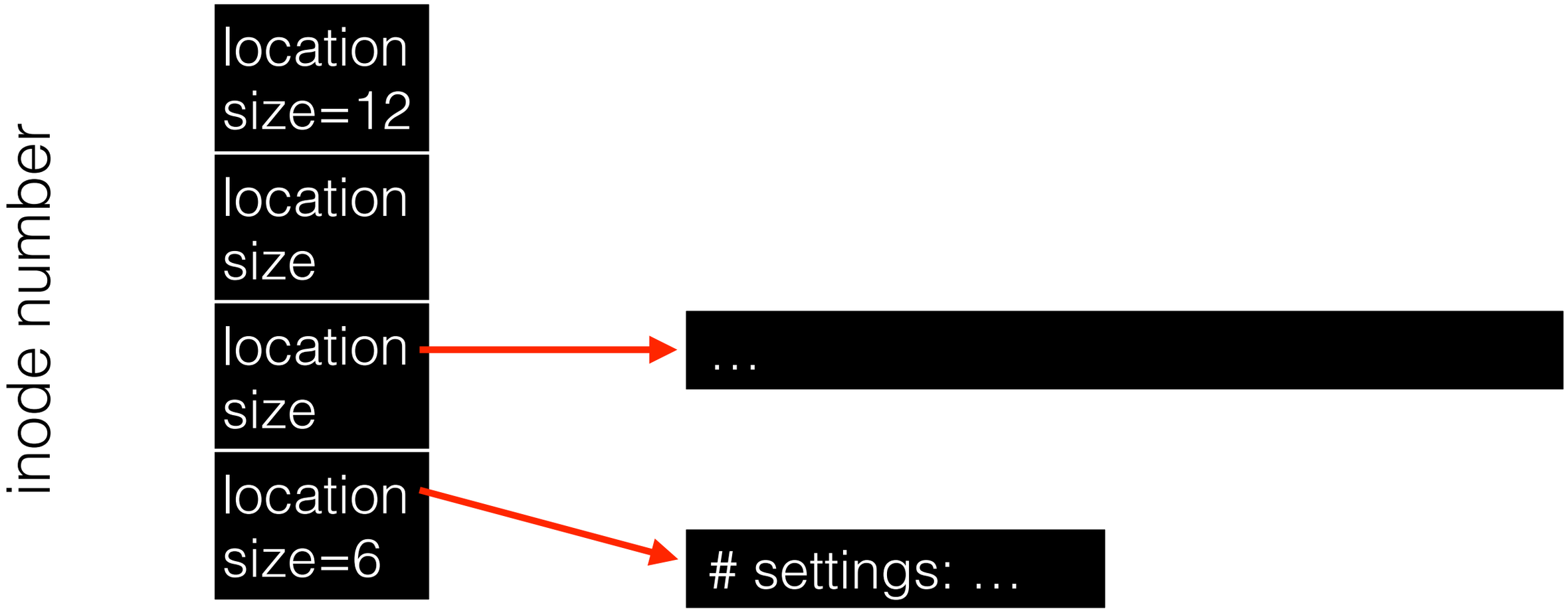
- fsync(int fd) forces buffers to flush to disk, and (usually) tells the disk to flush it's write cache too.

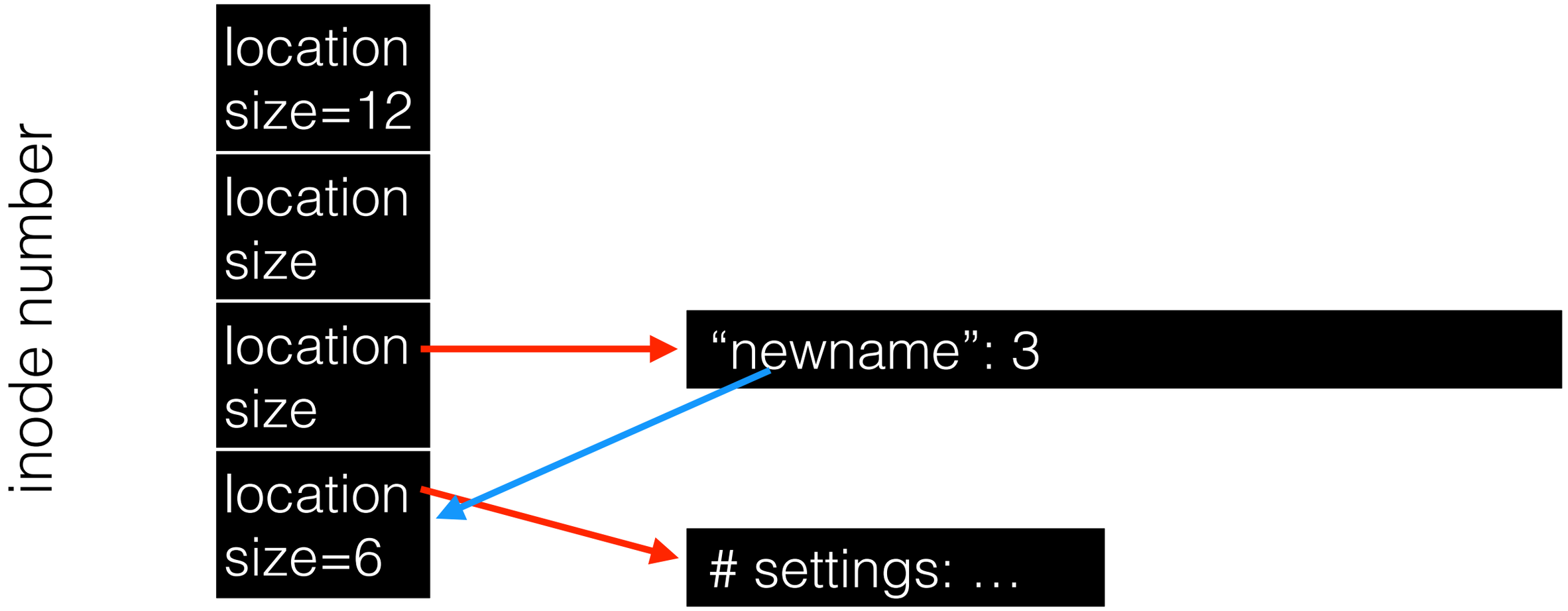
- This makes data durable.

rename

rename(char *old, char *new):
- deletes an old link to a file
- creates a new link to a file


inode number
location
size=12
location
size
location
size
location
size=6
“oldname”: 3, …
settings: …


inode number
location
size=12
location
size
location
size
location
size=6
…
settings: …


inode number
location
size=12
location
size
location
size
location
size=6
“newname”: 3
settings: …

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

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.

Let's Learn About Link Before Unlink
Hard Link

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

Let's Learn About Link Before Unlink
Hard Link

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

link(old pathname, new one)

Let's Learn About Link Before Unlink
Hard Link

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

link(old pathname, new one)

- Link a new file name to an old one

Let's Learn About Link Before Unlink
Hard Link

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

link(old pathname, new one)

- Link a new file name to an old one
- Create another way to refer to the same file

Let's Learn About Link Before Unlink
Hard Link

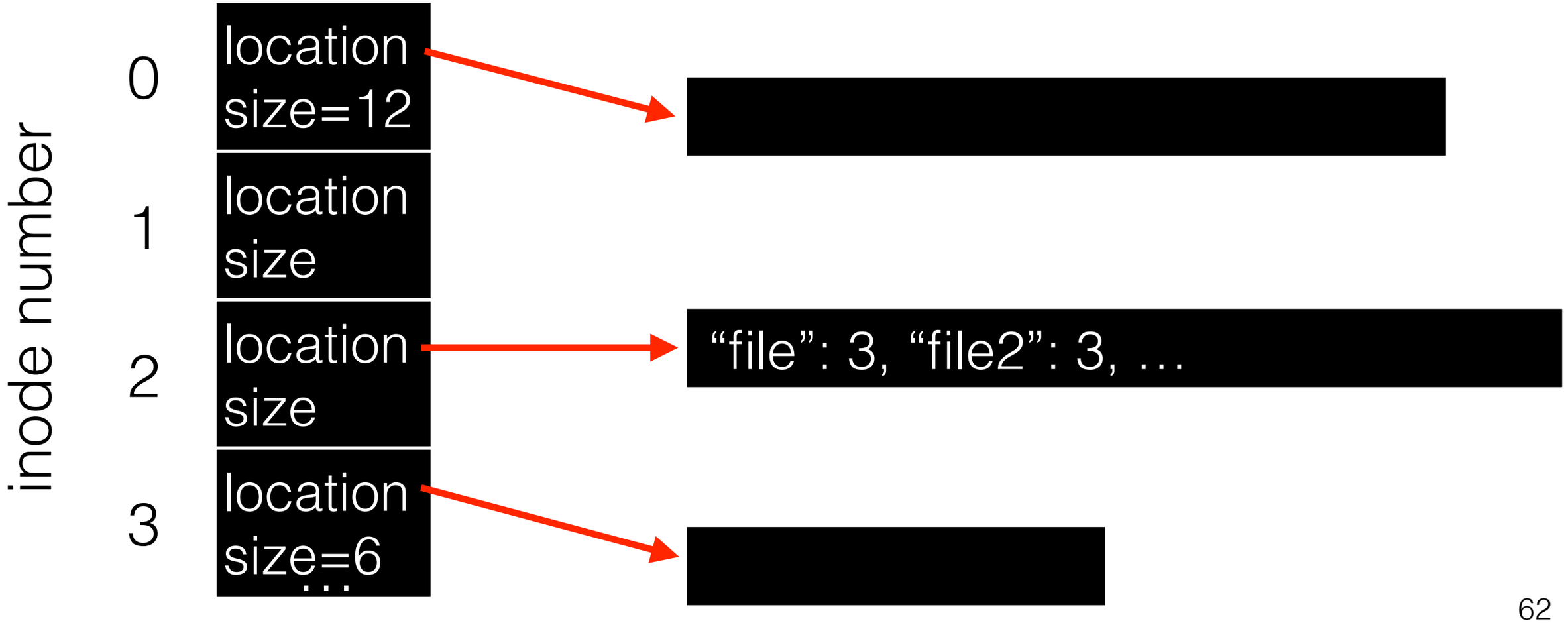
```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

link(old pathname, new one)

- Link a new file name to an old one
- Create another way to refer to the same file
- The command-line link program : ln

Let's Learn About Link Before Unlink
Hard Link

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```



Let's Learn About Link Before Unlink
Hard Link

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

Create another entry in the directory pointing to the same node