CS 423
Operating System Design: File Systems - I

Ram Alagappan

Acks: Prof. Tianyin Xu and
Prof. Shivaram Venkataraman (Wisconsin) for the slides.

DISKS → FILES

Plan

- This lecture: Files and FS API
- Next: File system implementation
- After: RAID/Other topics

What is a File?

Array of persistent bytes that can be read/written

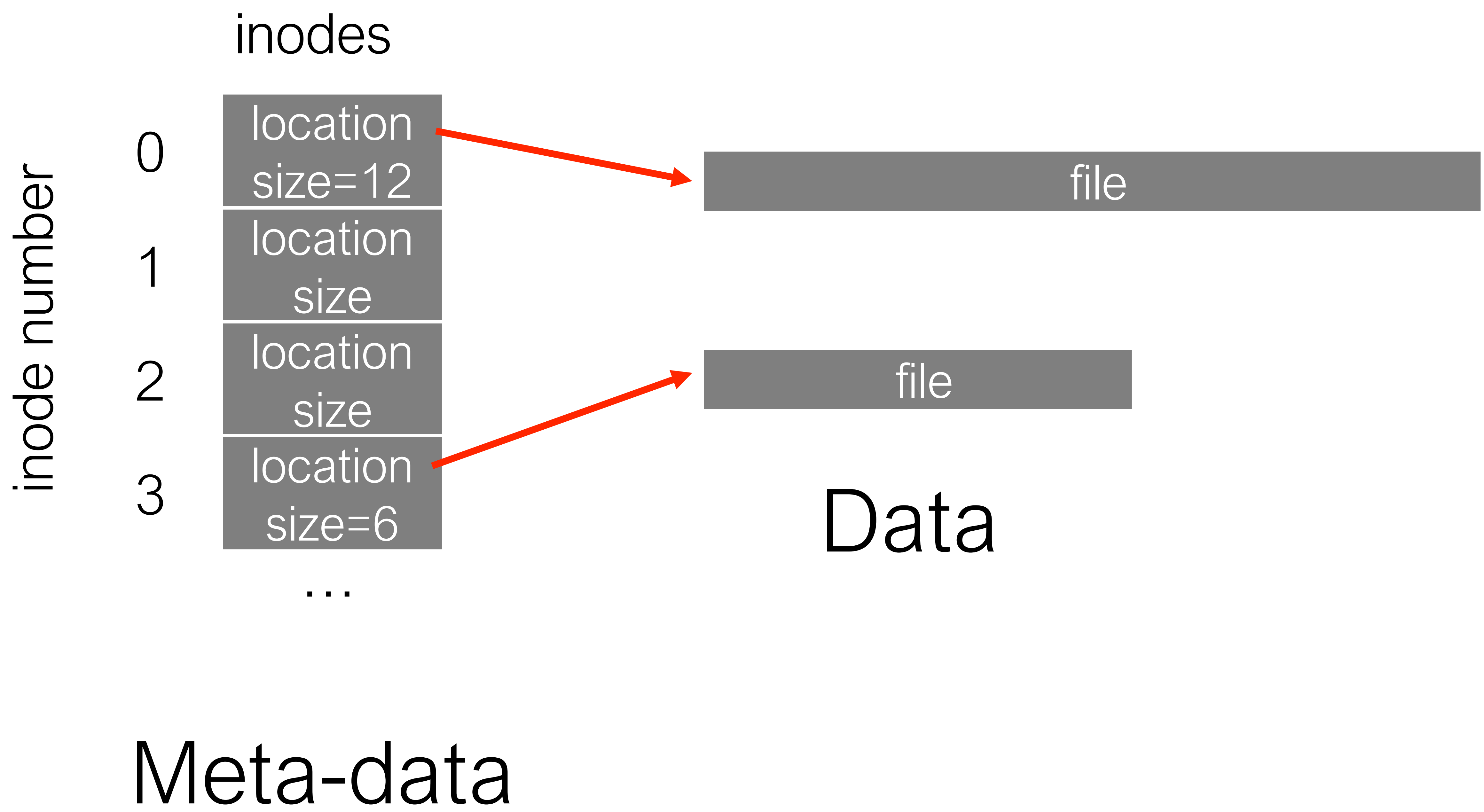
File system consists of many files

- Refers to collection of files
- Also refers to part of OS that manages those files

Files need names to access correct one

Three types of names

- Unique id: inode numbers
- Path
- File descriptor


inodes
inode number
0
1
2
3
location
size=12
location
size
location
size
location
size=6
...
file
file
Data
Meta-data

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?

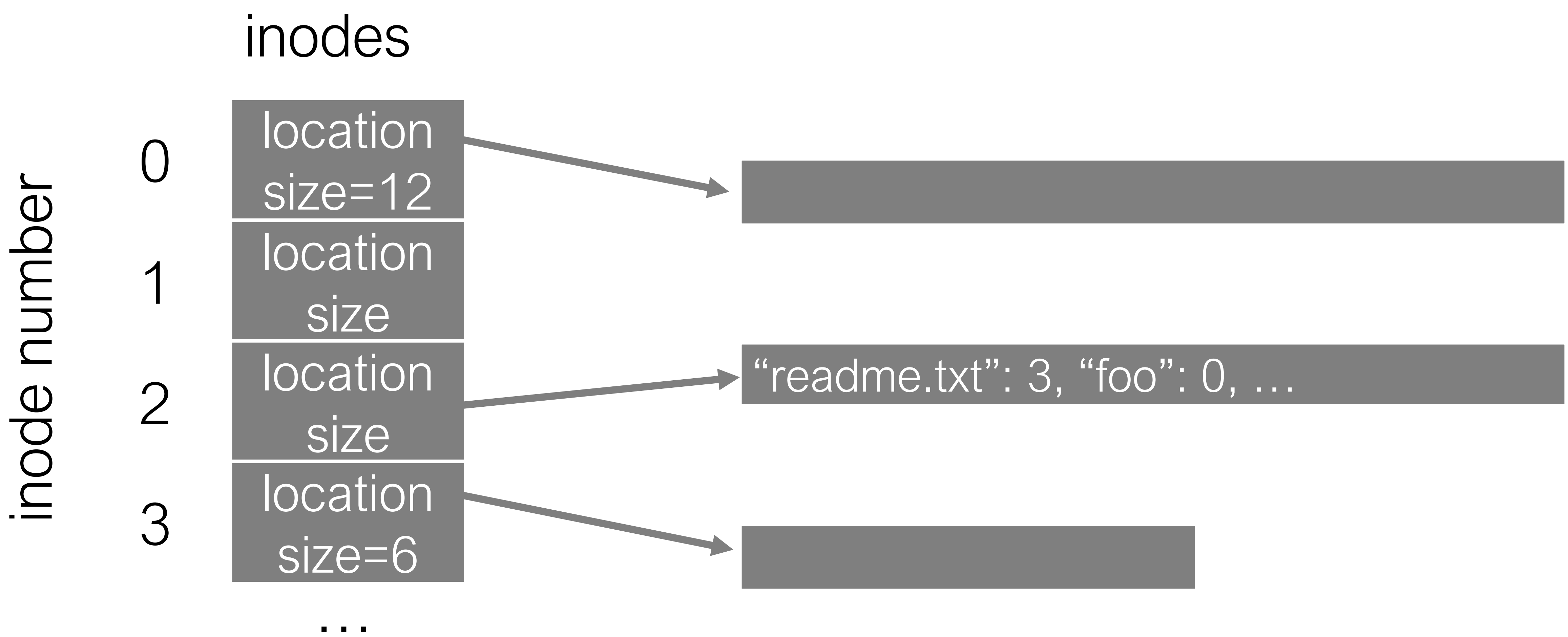
- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

Paths

String names are friendlier than number names

File system still interacts with inode numbers

Store *path-to-inode* mappings in a special file or rather a *Directory*!



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

Paths

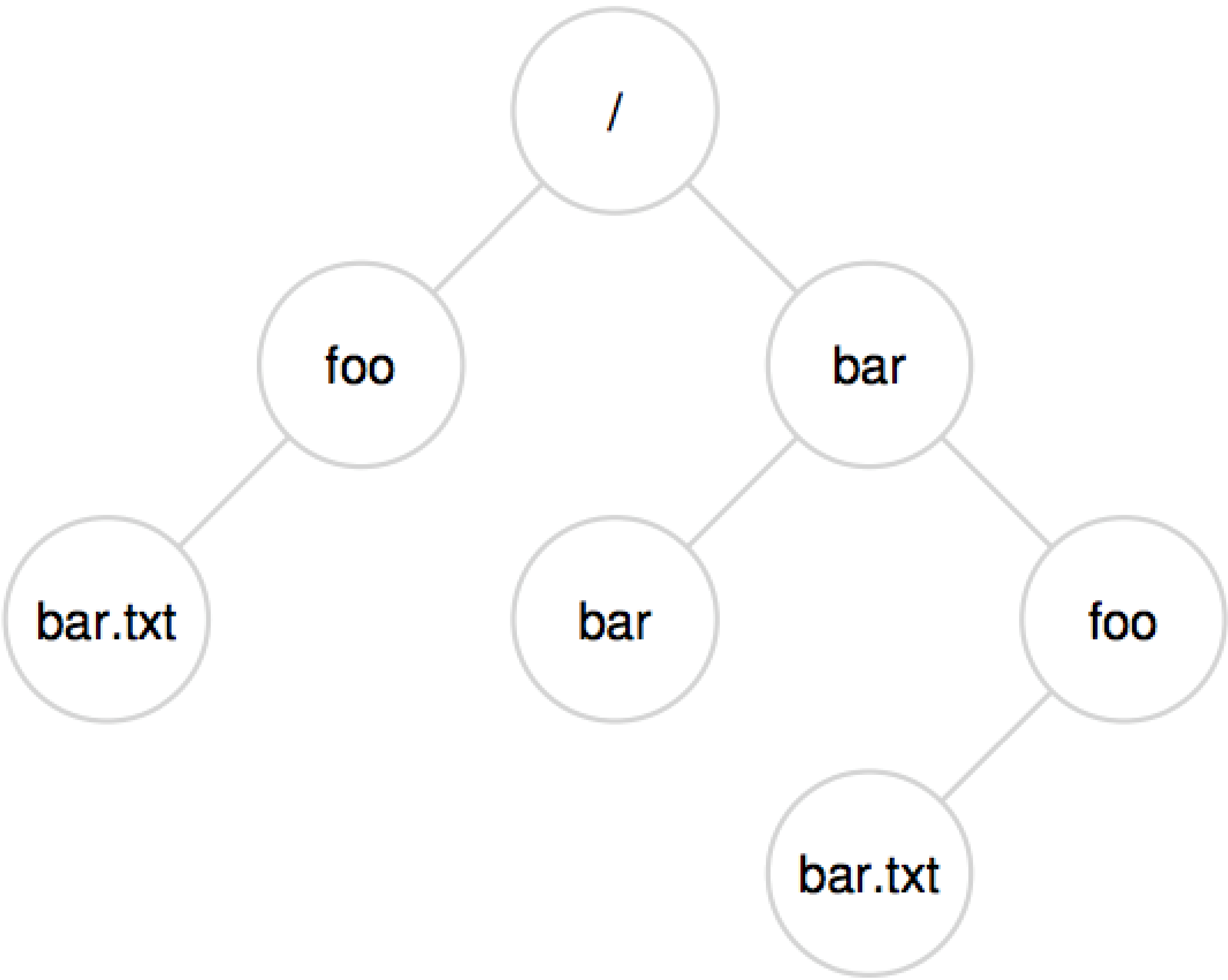
Directory Tree instead of single root directory

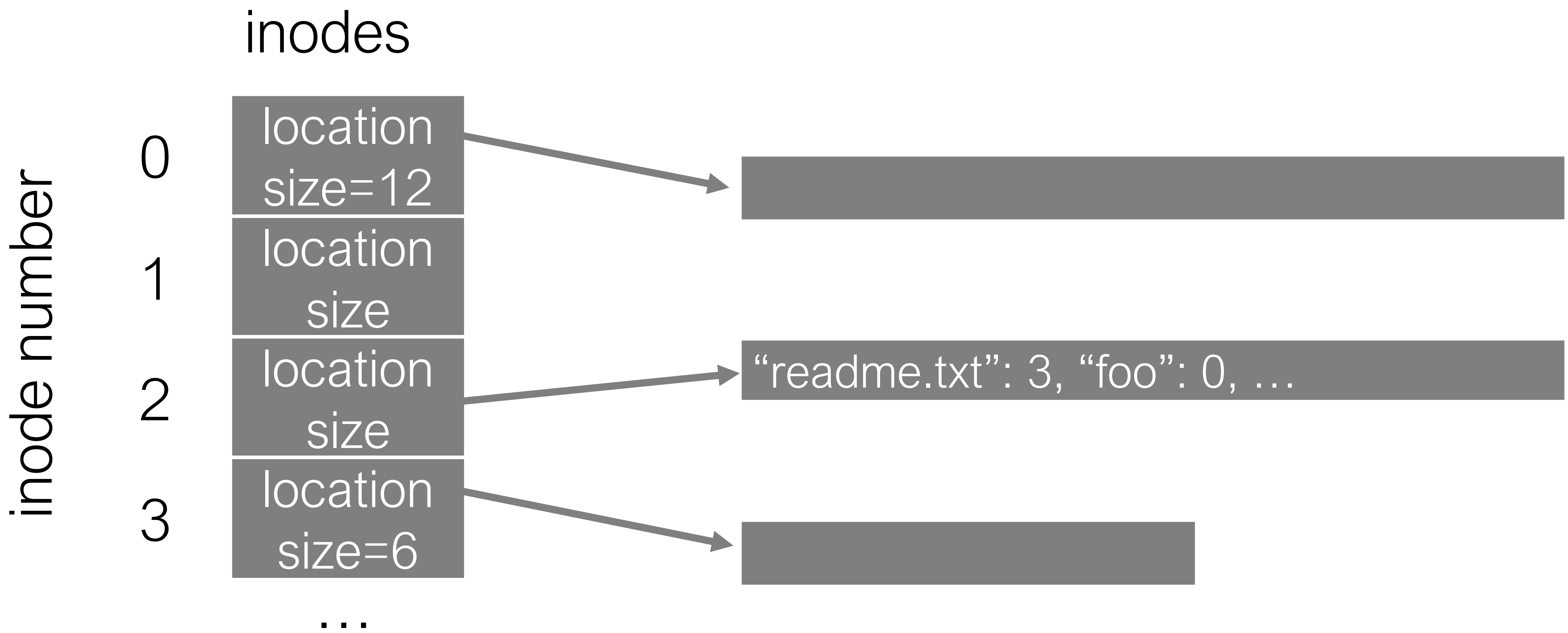
File name needs to be unique within directory

/usr/lib/file.so

/tmp/file.so

Store file-to-inode mapping in each directory





Reads for getting final inode called “traversal”

Example: read /hello

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 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

First time a process opens a file, what will be the fd in Unix/Linux?

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 {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

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

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

FD offsets

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

FD Offsets

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

LSEEK and READ

```
off_t lseek(int filedesc, 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.

Assume head is on track 1
Suppose we do lseek to X and the sector for X is on track 4
Where is head immediately after lseek?

Entries in OFT

When a process opens its first file (say whose inode is 10),

What will be the values in the file struct?

```
struct file {
   int ref;
   char readable;
   char writable;
   struct inode *ip;
   uint off;
};
```

What if another process opens the same file?

How will the values inside file struct change?

Shared Entries in OFT

Fork:

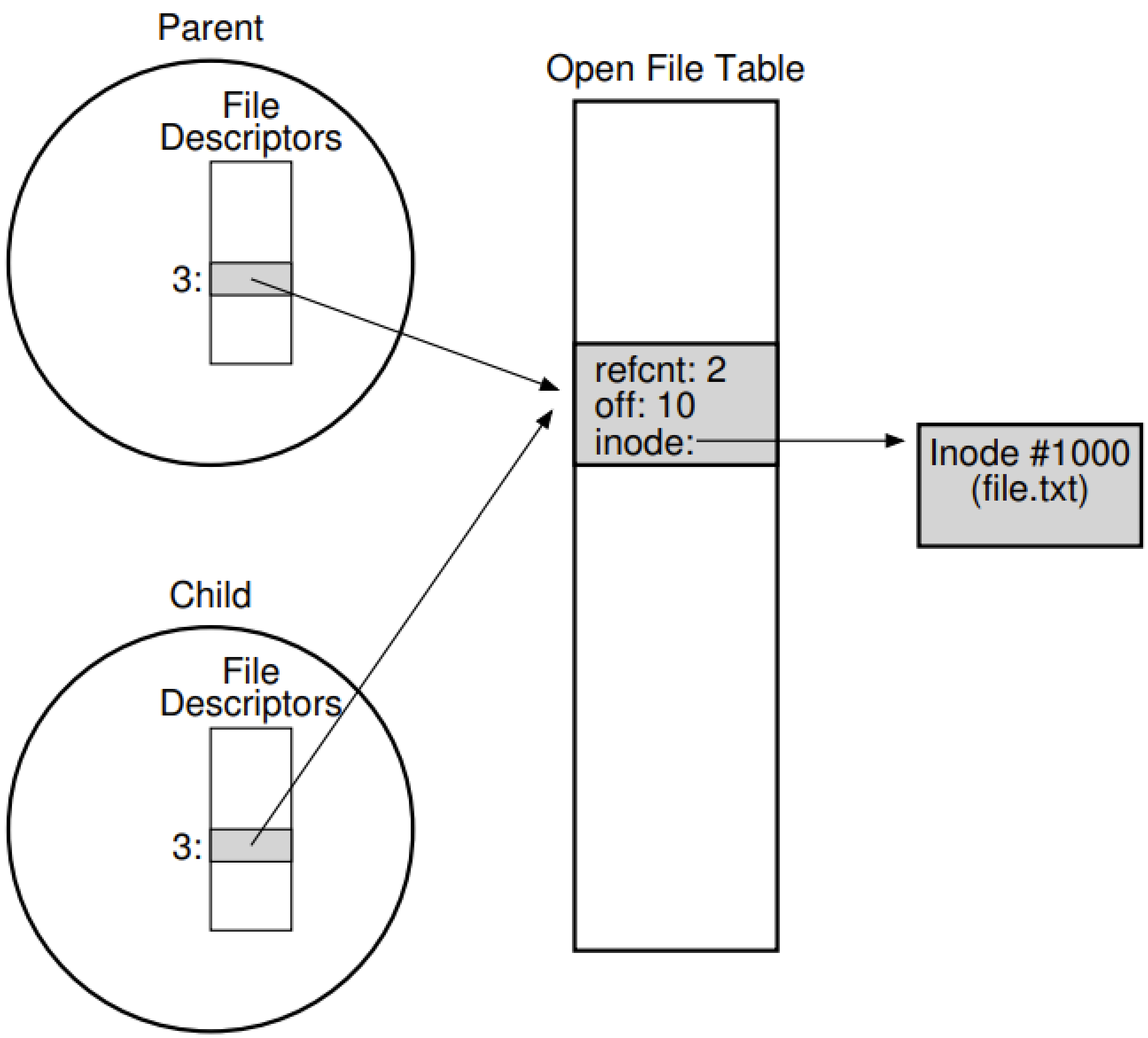
```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

What is the parent trying to print?

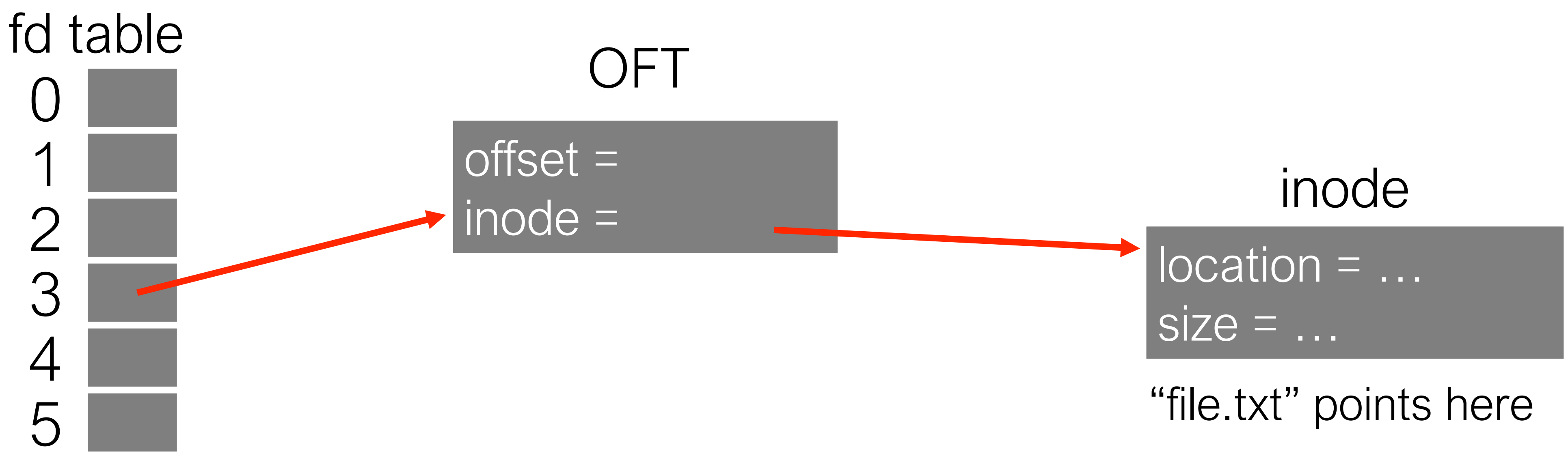
What value will be printed?

Shared Entries in OFT

What's happening here?



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
```

DUP

```
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);
```

How many entries in the OFT (assume no other process)?

Offset for fd1?

Offset for fd2?

Offset for fd3

Fsync

File system keeps newly written data in memory for a while

Write buffering improves performance (why?)

But what if system crashes before buffers are flushed?

fsync(int fd) forces buffers to flush to disk, tells disk to flush its write cache

Makes data durable

Rename

rename(char *old, char *new):

Do we need to copy/move data?

How does the FS implement this?

Does it matter whether the old and new names are in the same directory or different directories?

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 data

Even when renaming to new directory

What can go wrong if system crashes at wrong time?

Paths and Links

(Hard) Link

Inode has a field called “nlinks”

When is it incremented?

When is it decremented?

Deleting Files

What is the 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

Symbolic or soft links

A different type of link

Hard links don’t work with directory and cannot be cross-FS

touch foo; echo hello > foo;

Hardlink: ln foo foo2

Stat foo; what will be the size and inode?

Stat foo2; what will be the size and inode?

Softlink: ln –s foo bar

Stat bar; what will be the size and inode?

Atomic File Update

Say you want to update file.txt atomically

If crash, should see only old contents or only new contents

How to do?

FILE SYSTEM IMPLEMENTATION

VSFS

Very Simple File System

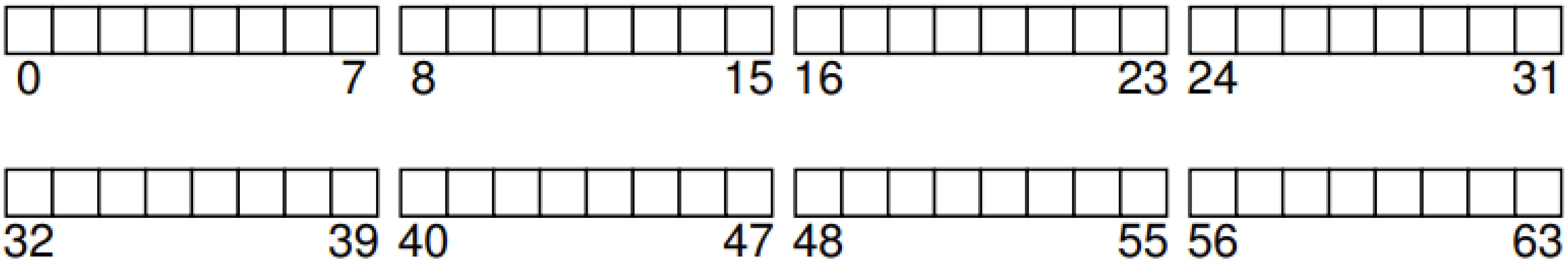
Two aspects:

Data structures – how are files, directories, etc stored on disk

Access methods – how are high-level operations like open, read, write mapped to these DS operations

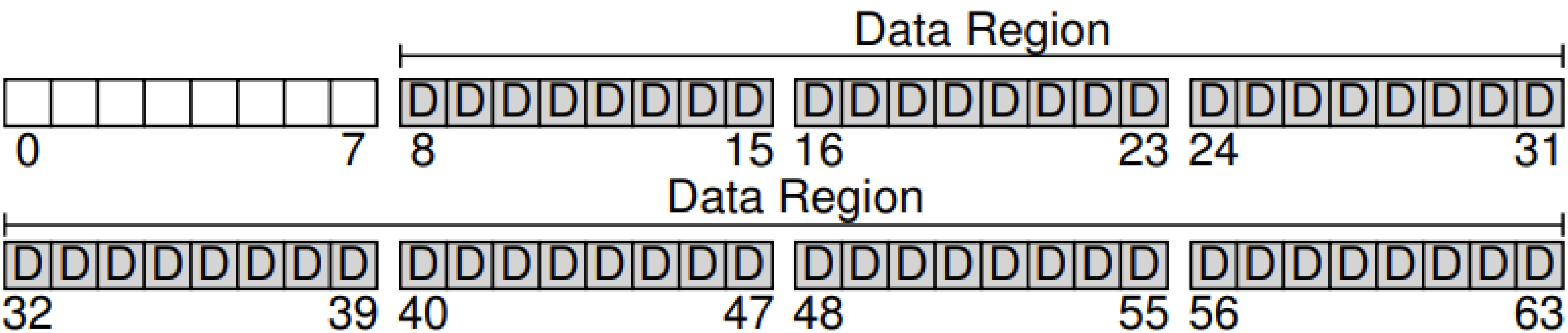
VSFS

Assume a small disk partition with 64 blocks

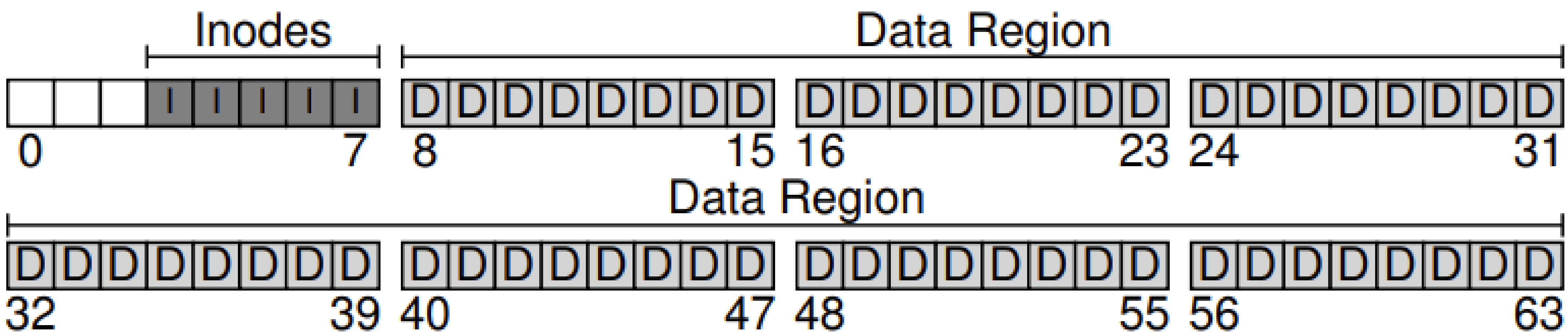


Data and metadata – most space must go for data blocks

VSFS – Data blocks



VSFS – Inodes (metadata)



Called the inode table

With 256-byte inodes, we can store 16 inodes in a block, so totally 80 files can be stored in VSFS

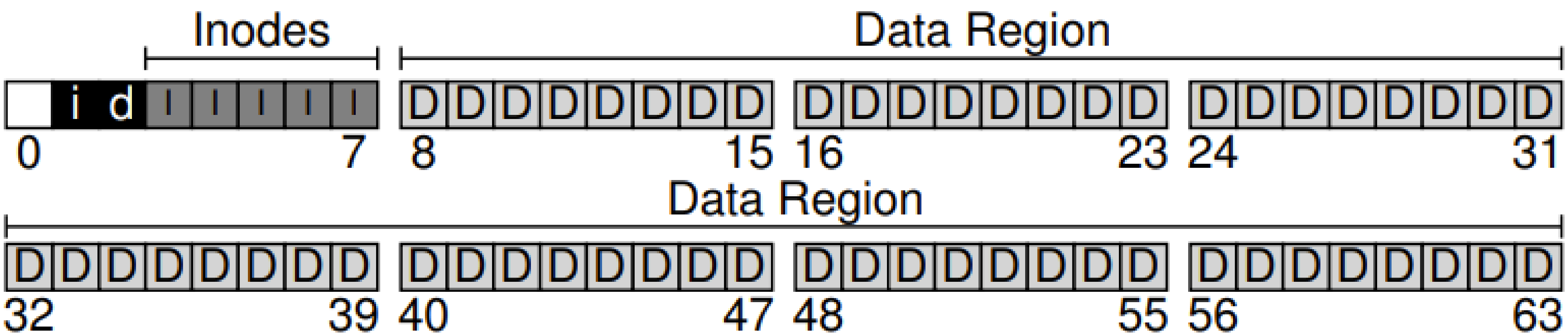
But we can simply scale VSFS to a larger disk

VSFS – Bitmaps (metadata)

Need allocation structures

Free lists – linked list is an option

Most commonly used: bitmap (ib, db)

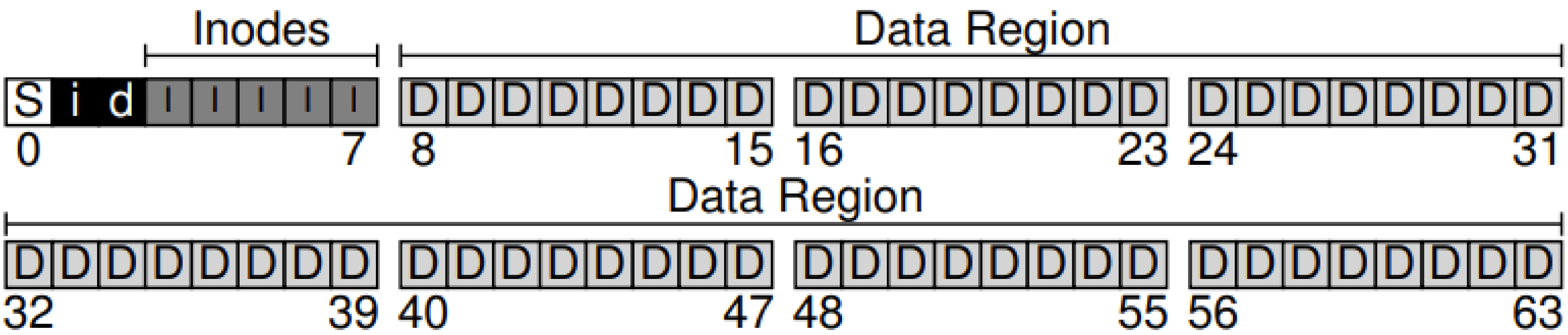


We actually don't need a block for bitmap

VSFS – (metadata)

What's stored in the first block?

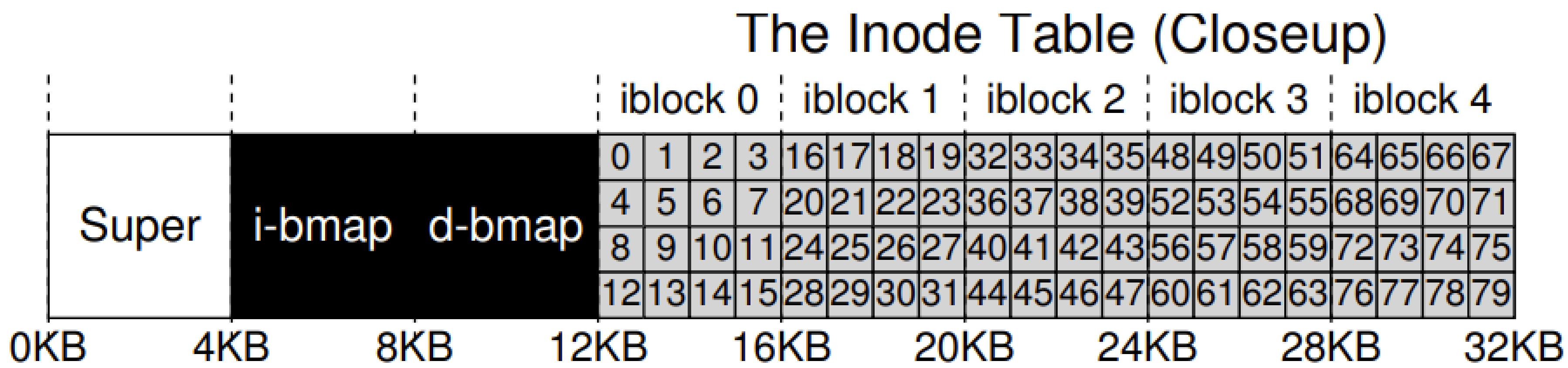
VSFS – Superblock (metadata)



How many data blocks, how many inode blocks

Inode table starting block #

INODE

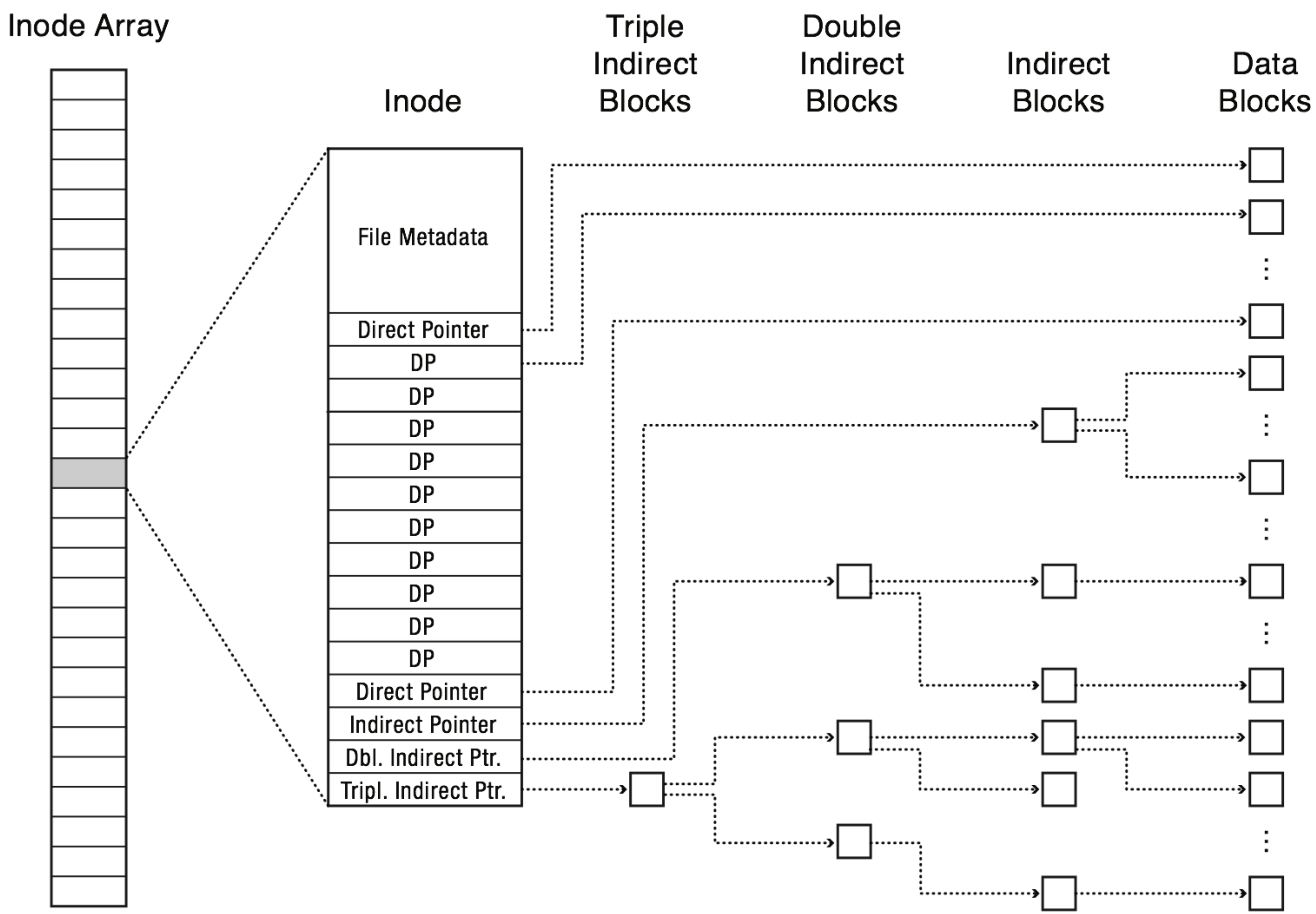


Implicitly know the block/sector number

INODE

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Direct and Indirect Pointers



Extent based approach

No pointer for every block

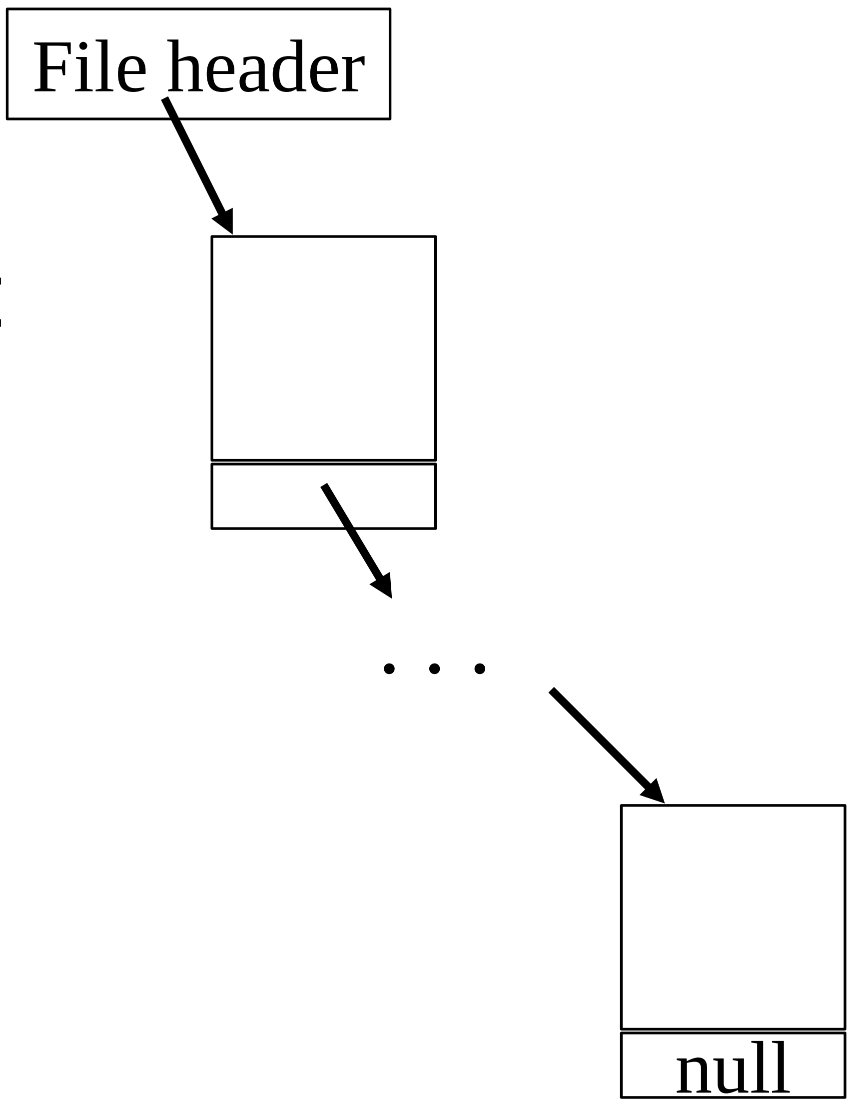
<Starting block, num blocks>

Adv compared to pointer approach?

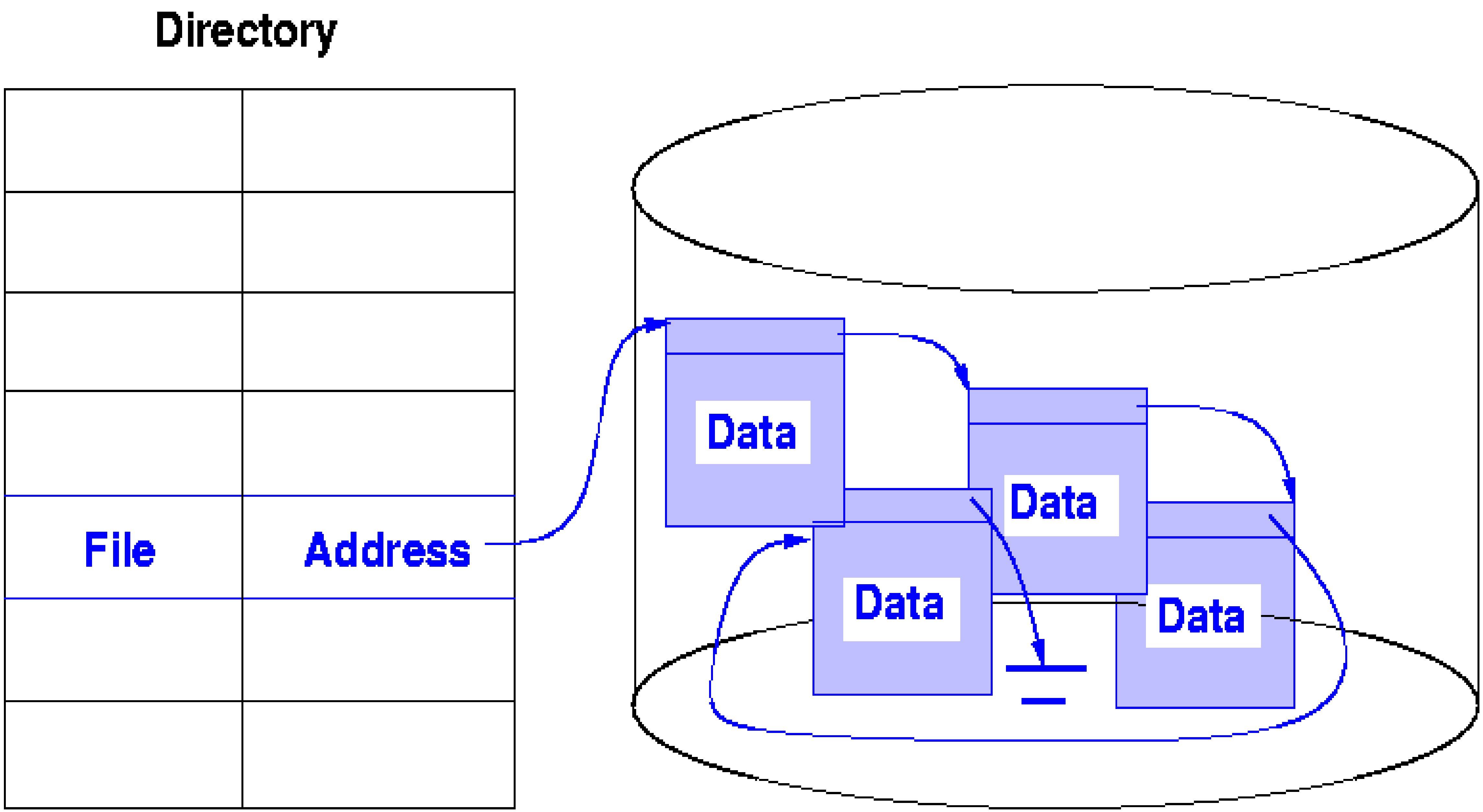
Cons?

Linked Files

- File header points to 1st block on disk
- Each block points to next
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
- Cons
  - random access: horrible
  - unreliable: losing a block means losing the rest



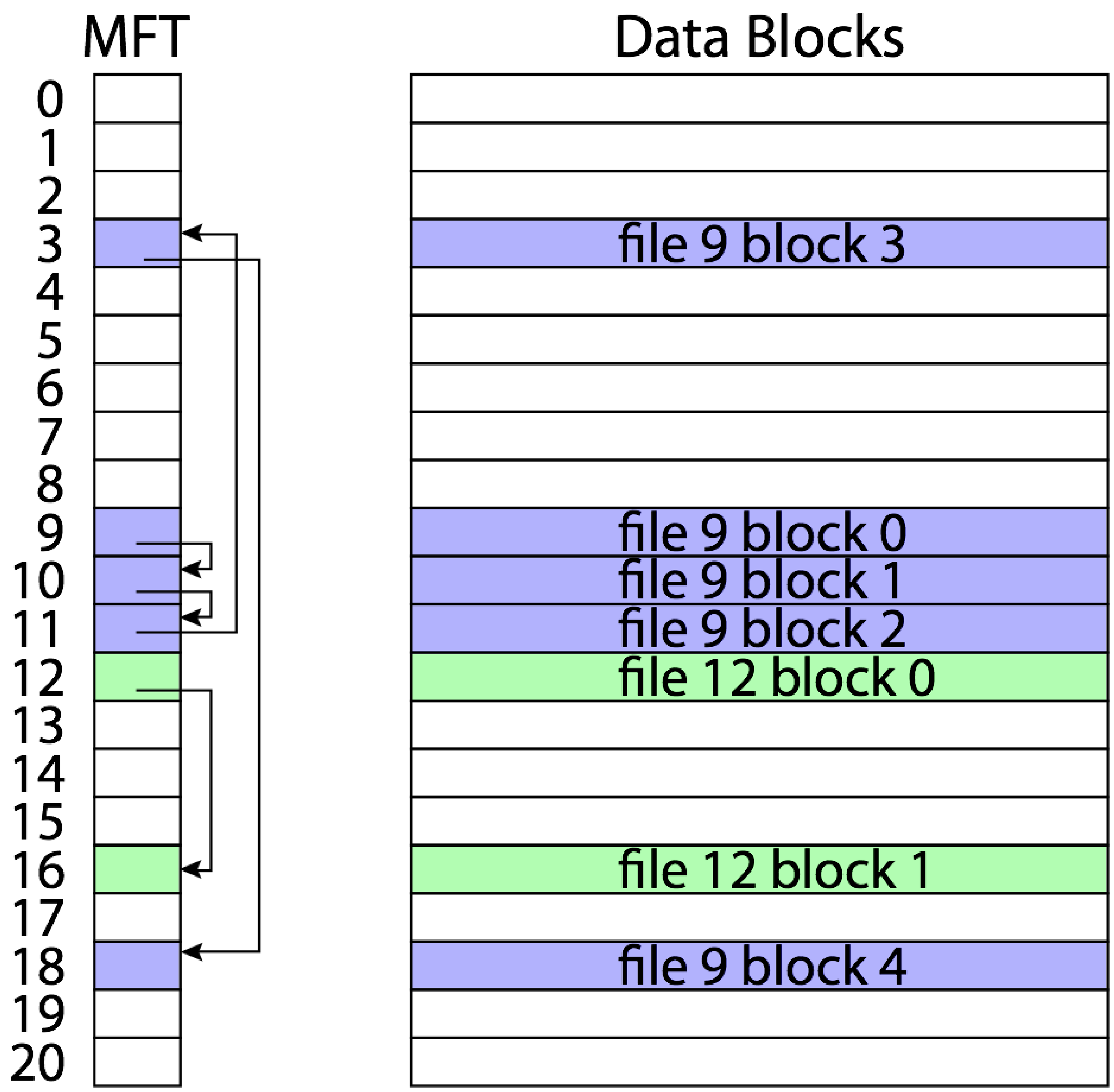
Linked Allocation



MS File Allocation Table (FAT)

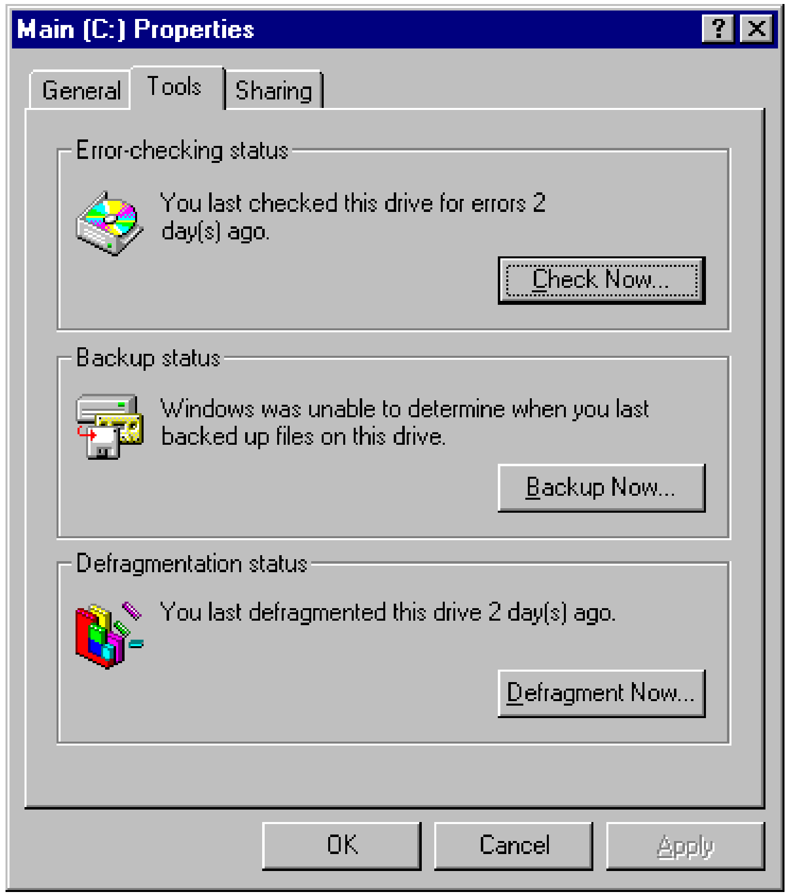
- Linked list index structure
  - Simple, easy to implement
  - Still widely used (e.g., thumb drives)
- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks

MS File Allocation Table (FAT)



MS File Allocation Table (FAT)

- Pros:
  - Easy to find free block
  - Easy to append to a file
  - Easy to delete a file
- Cons:
  - Small file access is slow
  - Random access is very slow
  - Fragmentation
    - File blocks for a given file may be scattered
    - Files in the same directory may be scattered
    - Problem becomes worse as disk fills



Small files: Inlined

- Really small files
- No need to have a separate data block
- Inline them into the inode – can access with fewer disk accesses

Directory Organization

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

What is the inode of this directory?

Where is the directory's content stored?

Next Lecture

Continue more with FS internals and implementation

Also, FFS/LFS/Journaling

Beyond: RAID