File Systems

Chapters 13 and 14

The File

- What's a *file*?
  - You know, it's a contiguous sequence of values (bytes)
  - Typically maintained on secondary storage … for non-volatility.
  - We use them to represent all kinds of things
    - Text, executable programs, images, music files, special formats used by particular programs … um, core files for when our programs crash, game save files, empty files, archives containing a bunch of other files, log files, configuration files, … probably a bunch of other things I'm not thinking of right now.

File Attributes

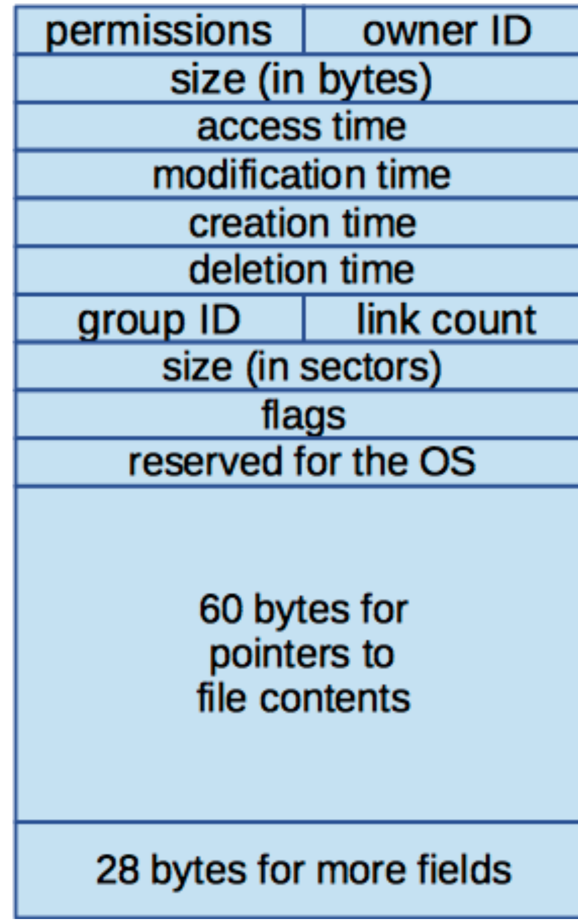
- A file is more than just contents
- Normally, we'll expect a collection of *file attributes* for each one
  - *Name* : a string identifying the file to a human user
  - *Location* : where to find file contents on the device
  - *Size* : how many bytes the thing contains
  - *Protection* : some kind of access list, including ownership
  - *Timestamps* : e.g., when the file was created, accessed, modified
  - *Type* : what kind of information the file contains (this one deserves special attention)

File Type via Filename Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

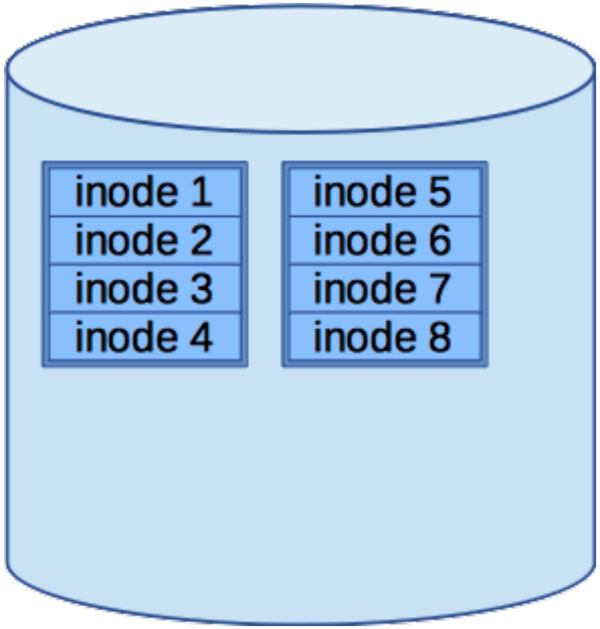
Representing Files

- Unix file systems like ext2 represent each file via an inode
- A fixed-sized per-file record
- Containing some things we'd expect
  - 16-bit owner ID
  - 16-bit group ID
  - Permissions (with 4 extra bits)
  - 32-bit timestamps



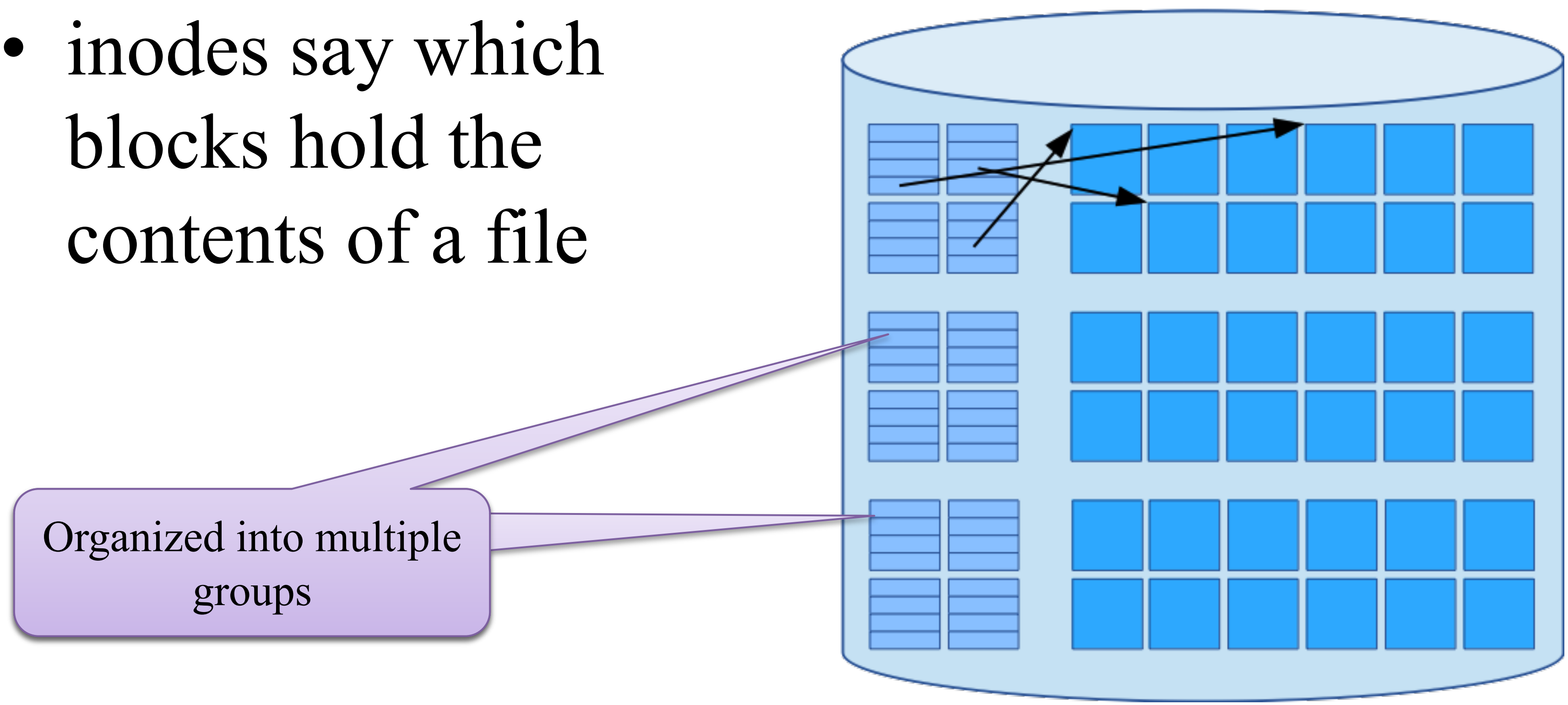
Storing inodes

- Disk blocks are represented as one or more sectors (e.g., a 4096-byte block)
- An inode might typically require 128 bytes.
- This is smaller than a typical block
  - so, multiple inodes are packed together in a range of blocks.
- Every inode has a unique index (counting from 1)



inode and Data Storage

- Some blocks are used for storing inodes
- Others for storing data.
- inodes say which blocks hold the contents of a file



File Operations

- As programmers, what kinds of things do we expect to do with files?
- I'm glad you asked. The OS provides a lot of system calls just for working with files.
- Create/read/write/seek/truncate individual files
  - int open(path, flags, mode)
  - ssize_t read(fd, buffer, count)
  - ssize_t write(fd, buffer, count)
  - off_t lseek(fd, offset, whence)
  - int ftruncate(fd, length)
- Tell the OS when you're done with a file (so it can stop buffering)
  - int close(fd)

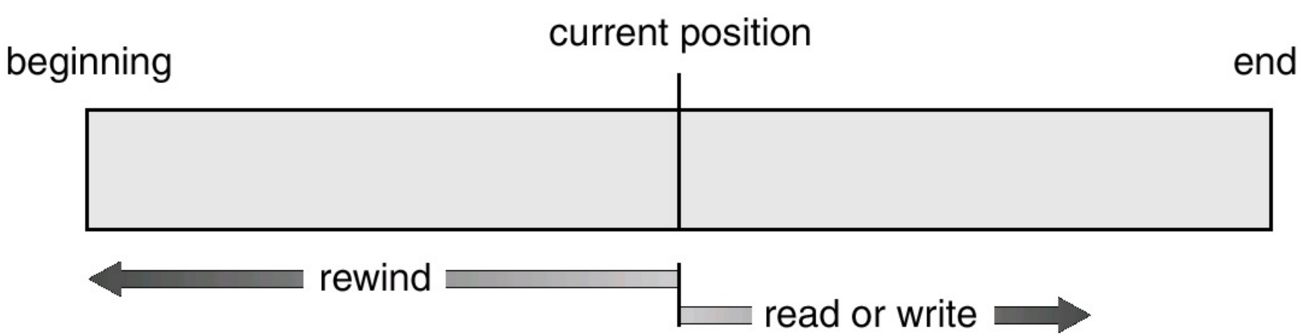
File Operations

- Query protection and other information about files
  - int fstat(fd, struct stat *buffer)
  - int fchmod(fd, mode)
  - int fchown(fd, owner, group)
- Execute programs stored in files
  - int execl(path, arg0, arg1, …, NULL)

File Access Methods

- Processes may want to use files for different things and use them in different ways
- *Sequential Access* : Access file contents in order
  - Maintain a current position, advanced as we read
  - What would this API look like?

```
read nextBytes
write nextBytes
rewind
```



File Access Methods

- Direct access
  - Can read/write to an arbitrary location on each access
  - What would this API look like?

```
read n, nextBytes
write n, nextBytes
```

```
seek to n
read nextBytes
write nextBytes
```

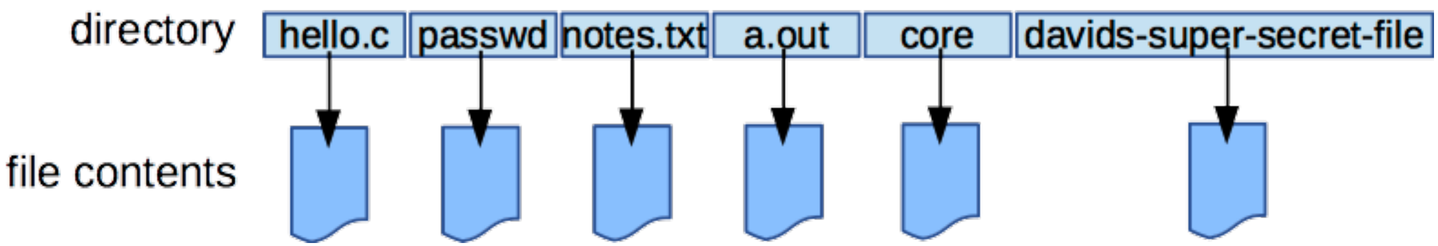
```
n = relative block number
```

Meet the File System

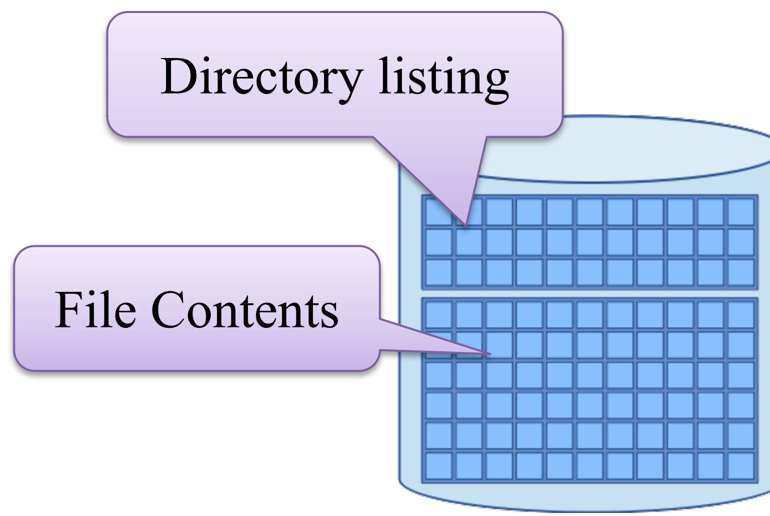
- It’s a data structure, built from fixed-sized blocks on secondary storage
- It maintains
  – A collection of files, each with its own contents
  – A name for each file, typically along with other metadata
  – Some type of directory structure, to help organize files.

Directory Structure

- Mapping names to file contents is really a job of the file system.
- How about something simple, a *single-level directory*



- Maybe easy to implement
- Some file systems are structured this way.
- … but, could be a limitation if we need lots of files and users.

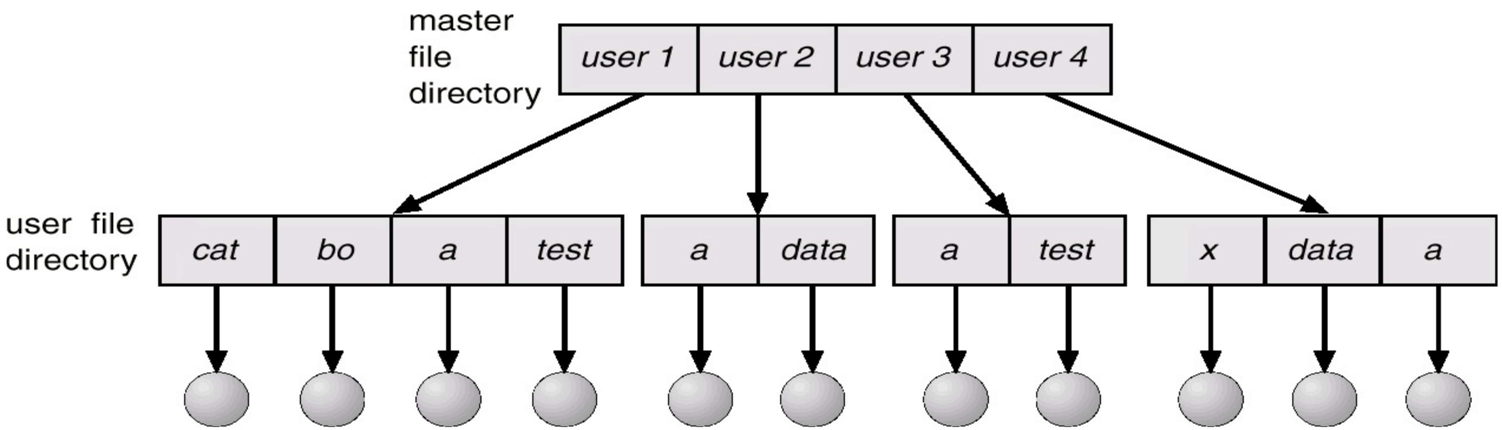


Directory Structure

- A more sophisticated directory structure can offer some advantages
- Naming flexibility
  - We don’t need to compete for a single shared directory
- Grouping
  - We can organize files based on what they’re for
- Efficiency
  - Maybe we never need a single directory with, say, 10,000 files in it

Two-Level Directory

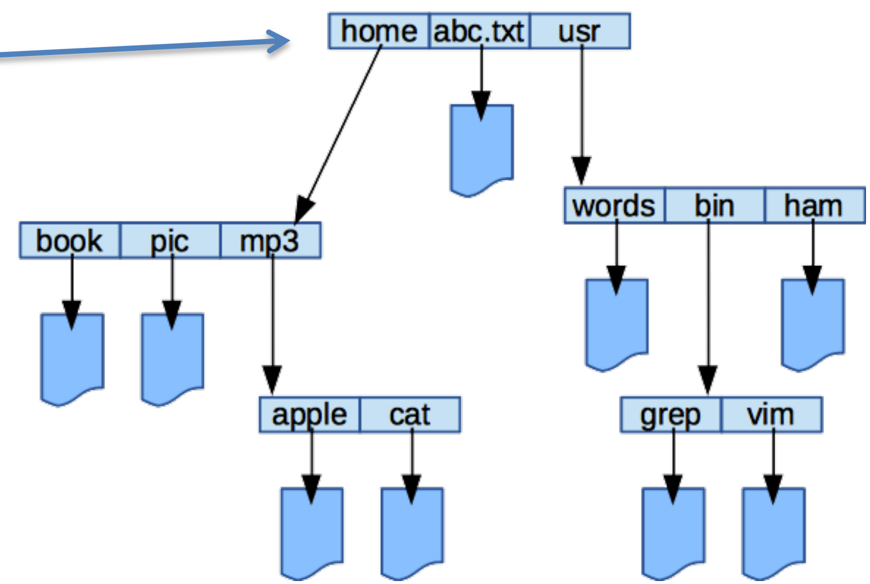
- Separate directory for each user



- Path name
  - “user1/test/hw5.exe”
- Can have the same file name for different user
  - “user1/test/hw5.exe” and “user2/test/hw5.exe”
- Efficient searching
- No grouping capability

Tree Structured Directory

- How about a *tree-structured directory*
- Now, a directory is just something else that can show up in a file system.
- Naming gets more interesting
  - *root directory*
  - *current directory*
  - *absolute path name*
  - *relative path names*
- Efficient searching
- Grouping capability



Tree Structured Directory

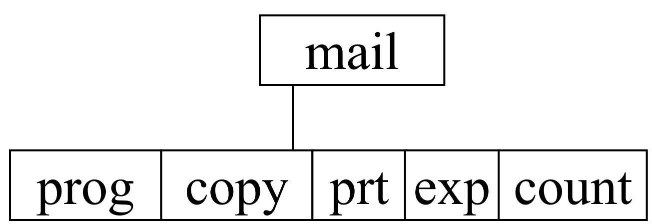
- Creating a new file is done in current directory
- Delete a file

  rm <file-name>
- Creating a new subdirectory is done in current directory

  mkdir <dir-name>

  Example: if in current directory /mail

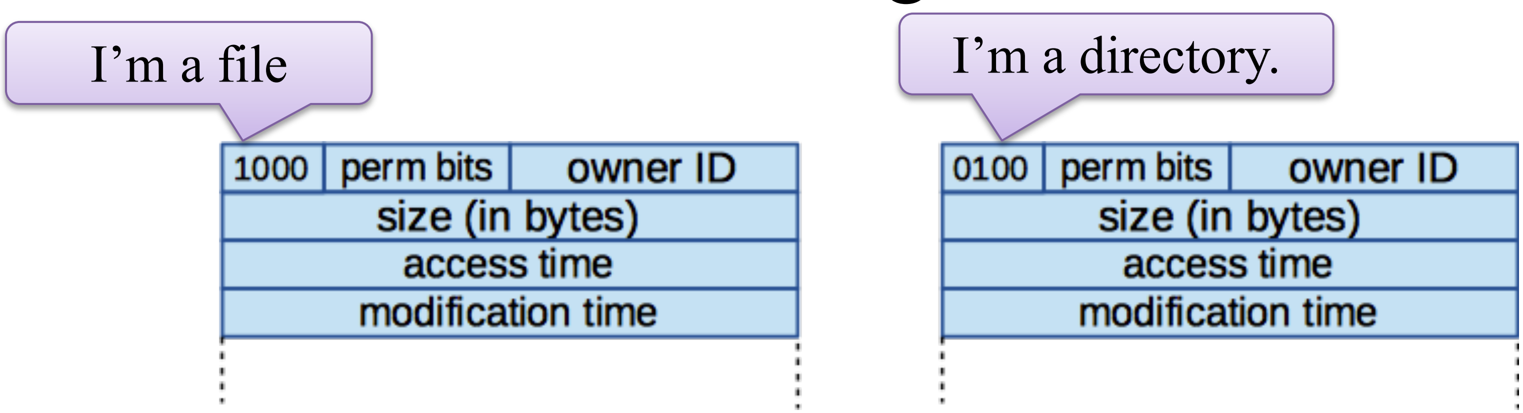
  mkdir count



Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

inodes for Directories

- inodes have this covered. The type field indicates what this thing is:



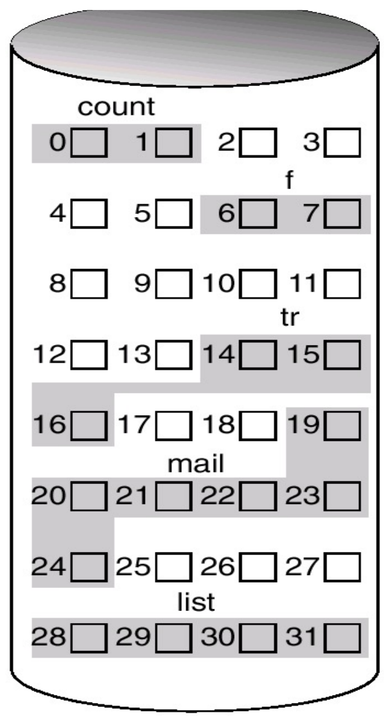
- Contents of the directory are stored just like file contents.

Organizing File Contents

- We’ve been vague on how we’re going to keep up with the blocks representing a file/directory’s contents.
- That’s called the *allocation method*
- We have some choices here.
  - Contiguous
  - Linked
  - FAT (really the same thing as linked, but better)
  - Indexed

Contiguous Allocation

- Each file occupies a contiguous sequence of blocks on the disk
- Simple, each file only requires a starting location and length ☺



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

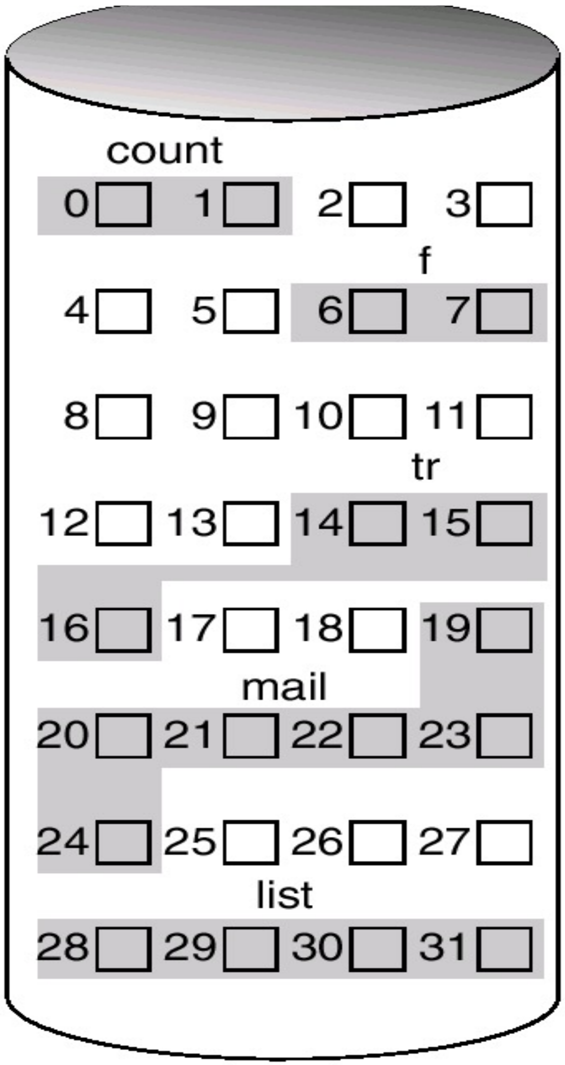
Contiguous Allocation

- Storage overhead
  - No per-block storage overhead
  - But we'll still have fragmentation (what kind?)
- Sequential access performance (accessing a file from start to finish) ☺
- Direct access performance (accessing a file in an arbitrary order) ☺
- Hard to find space for a new file ☹ (it's basically the storage-allocation problem)
- Hard to permit files to grow ☹ (without moving stuff around)

Mapping Locations into Contiguous Allocation

- A file is just a logically contiguous sequence of bytes
- The OS has to find the right byte in the right block, it's like address translation
- This is easy with contiguous allocation
  - Pretend a is the file location desired by some process
  - We want to compute the block number, q, and displacement, r
  - Let block size = 512 bytes.
  - Which block do you need? That's just $q = a / 512$
  - Which byte in that block, that's just $r = a \% 512$

Fun with Contiguous Allocation Address Translation



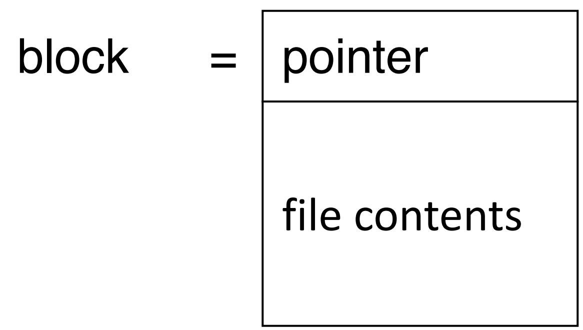
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- If block size = 512
- Where’s byte 514 of file “count”
- Where’s byte 1034 of file “tr”

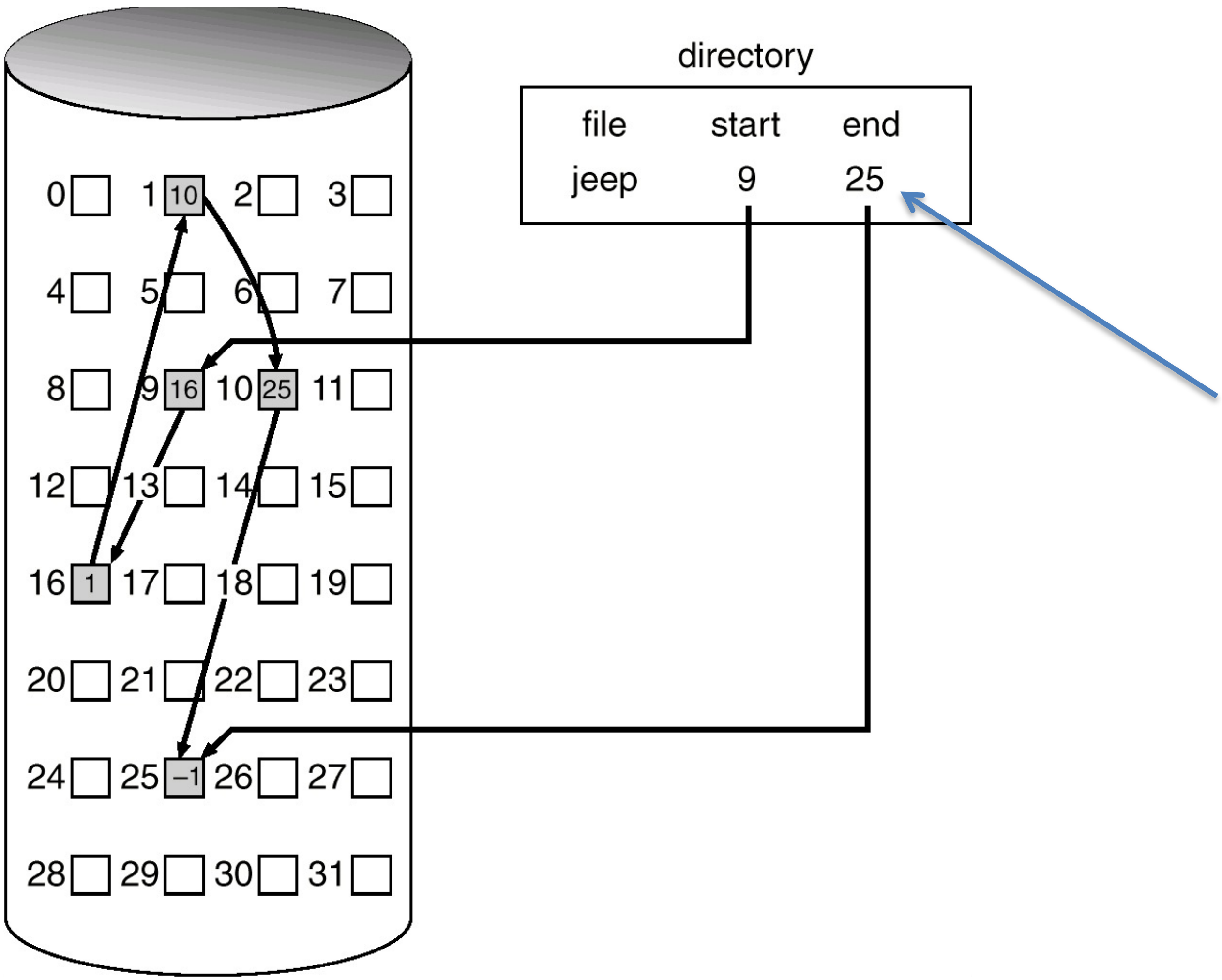
Linked Allocation

- Of course, contiguous allocation makes it hard for files to grow. Alternatives?
- Let each block contain a pointer to the next block in the file



- In a 512-byte block, we're using maybe 4 bytes for the pointer
  - Consider, choices like this may limit maximum file system size

Linked Allocation



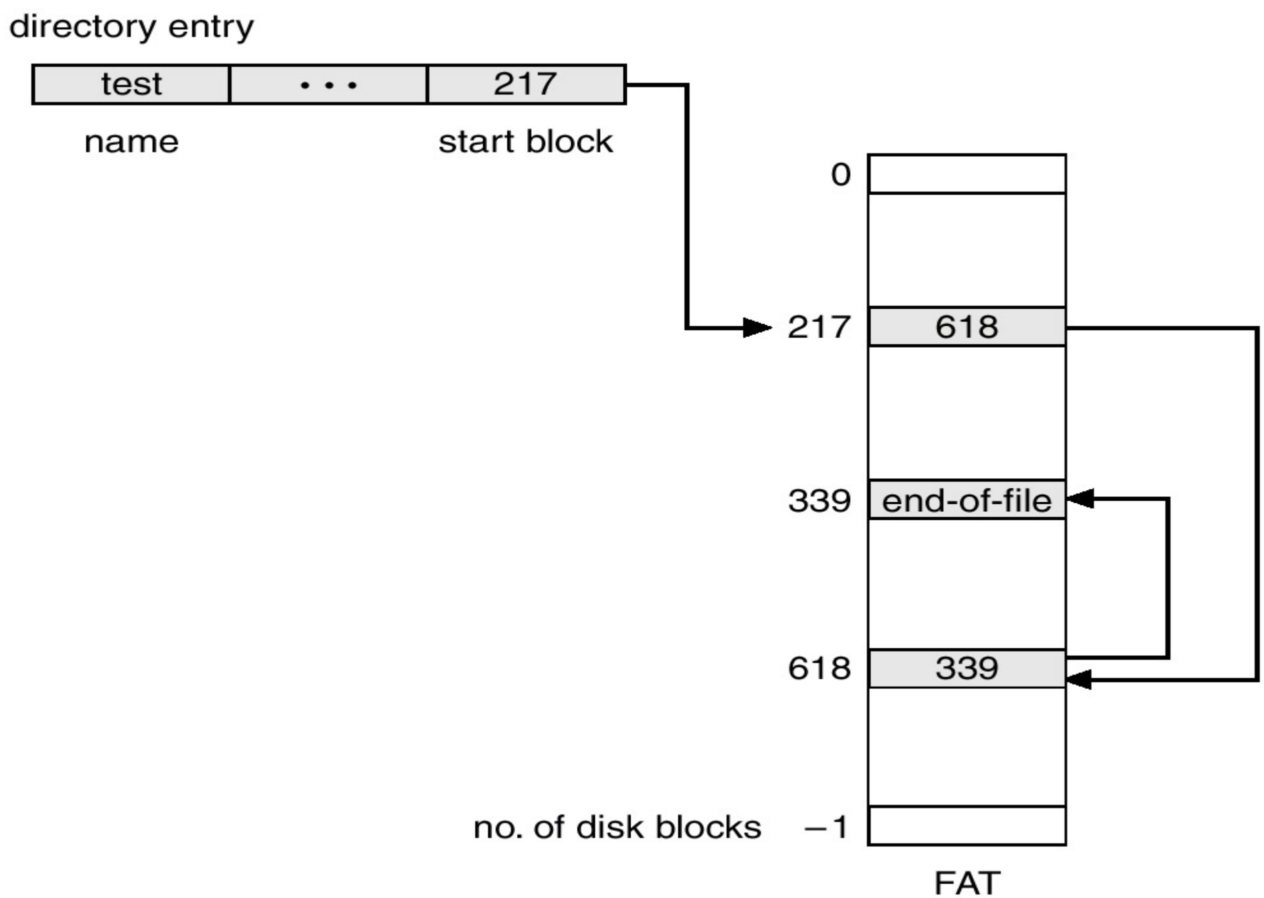
Why are we keeping a start and an end pointer?

Linked Allocation

- Still kind of simple, just need a starting address ☺
- Easy to find space and to grow files, any block can be used ☺
- Just a little per-block storage overhead 😐
  - File system may use *clusters* to reduce this
- Direct access performance ☹
- Mapping, consider the per-block overhead (same parameters as before)
  - Block number $q = a \,/\, 508$
    - But, you have to follow links to actually get that block
  - Offset into block $r = 4 + a \,\%\, 508$

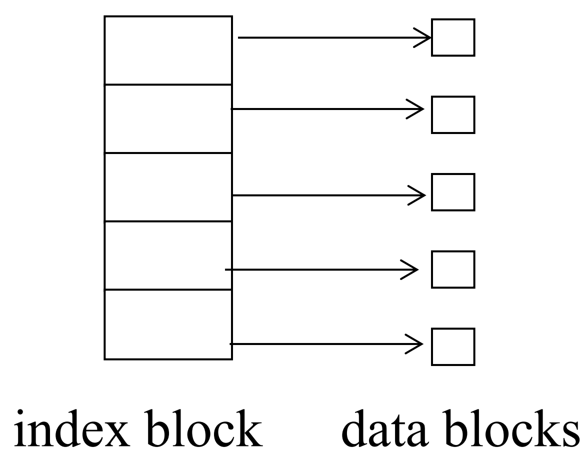
File-Allocation Table (FAT)

- Disk-space allocation used by MS-DOS.

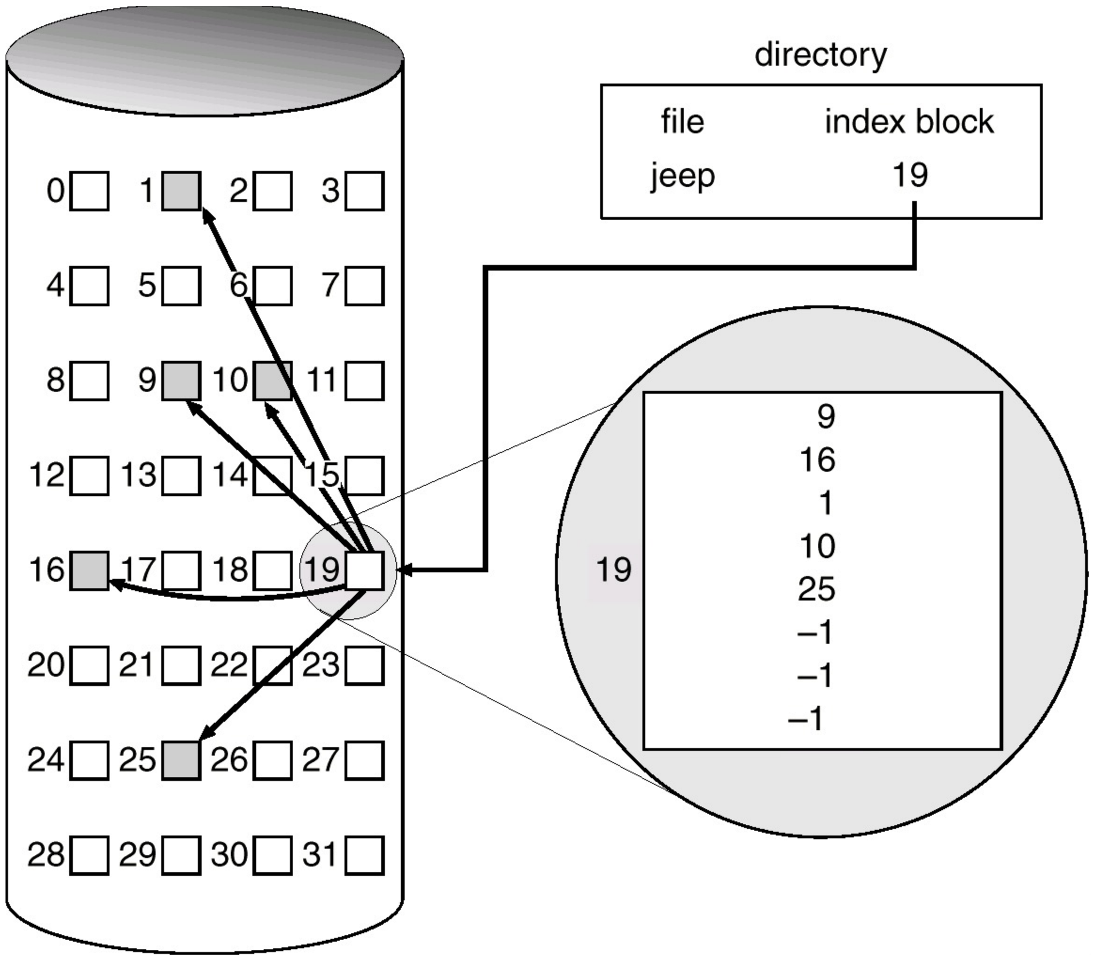


Indexed Allocation

- Start with linked allocation …
- But, bring all the pointers for a particular file together into an *index block*



Indexed Allocation



Indexed Allocation

- As with linked, files can grow into any block
- Need to look at the index block to find the data block
  - Additional performance overhead 😐
  - Additional storage overhead (a least one block) ☹
- Direct access performance 😐 to ☺
- Mapping
  - Block location: $q = index[\ a / 512\ ]$
  - *Block displacement:* $r = a \% 512$

Indexed Allocation Geometry

- Given a 512-byte block size
  - If each pointer is 4 bytes

    $512/4 = 128$ ptr

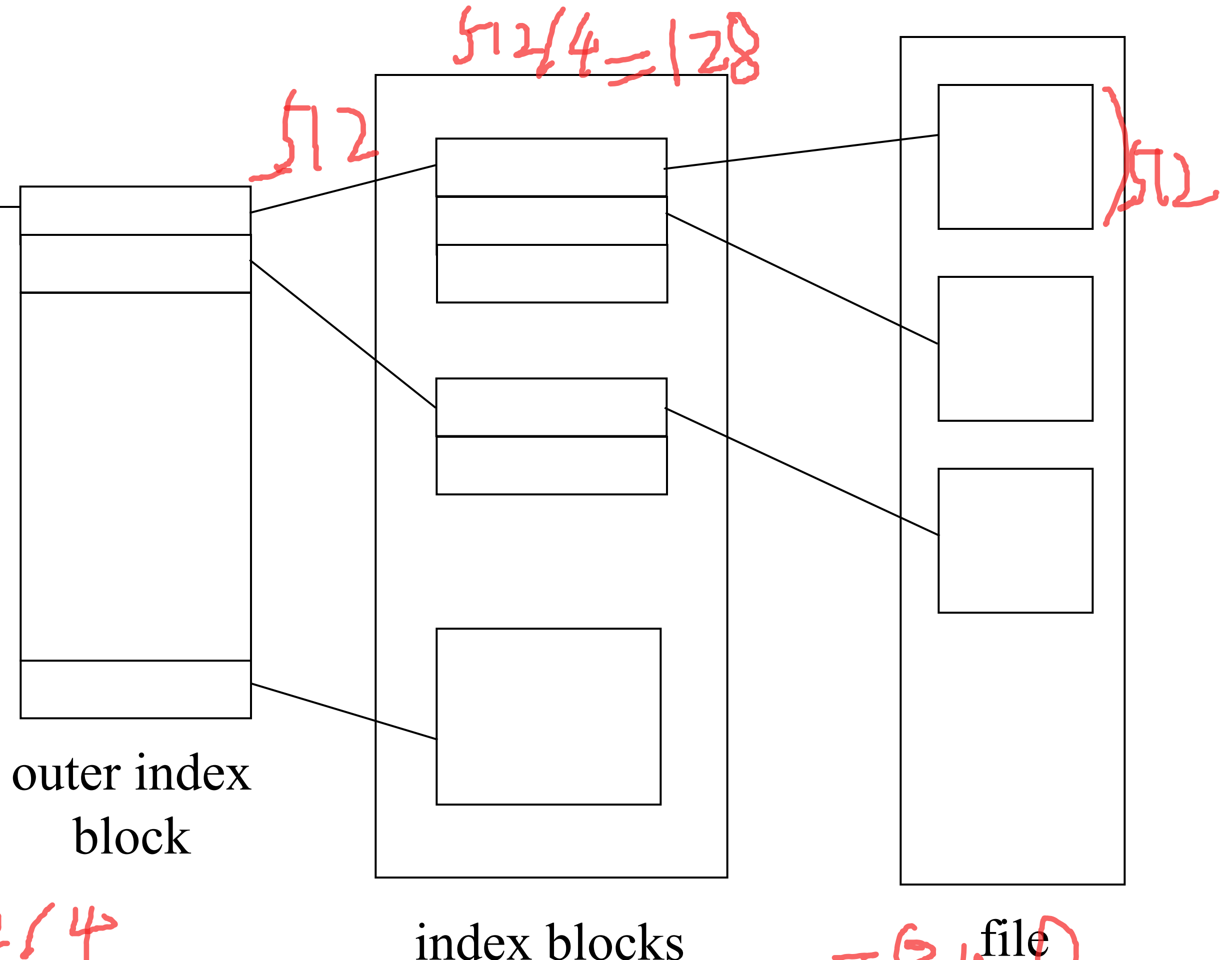
  - What's the largest file we can support?

    $128 \times 512B = 2^7 \times 2^9 = 2^{16} B = 64KB$

    9 → Data Block

  - That's not too big.
  - we could link index blocks together
  - Or, we could use a multi-level index.

Two-Level Indexing



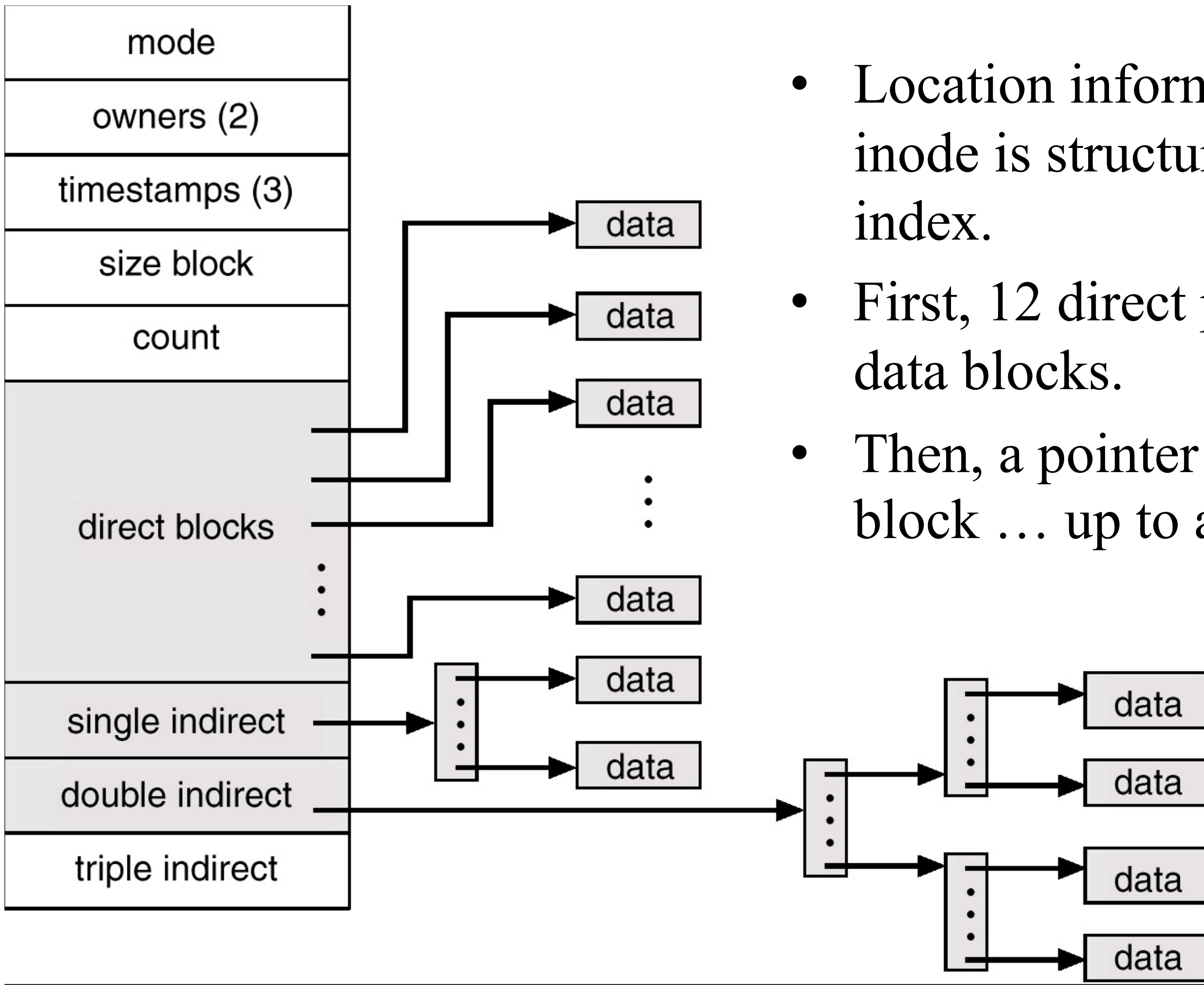
- An outer index block, each line pointing to a block of an inner index

512

- Each line of the inner index points to a data block

512/4 =128 pt

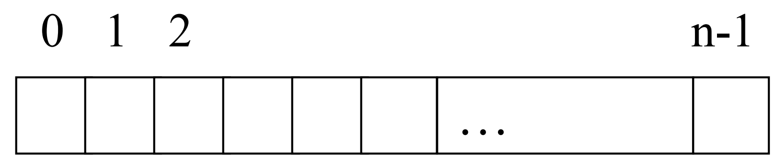
Indexing via inodes



- Location information in an inode is structured like an index.
- First, 12 direct pointers to data blocks.
- Then, a pointer to an index block … up to a 3-level index.

Free-Space Management

- We need to be able to quickly find unused blocks
  - Maybe with a bias toward contiguous sequences of blocks
- We could use a bitmap, with a bit for each block



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

- Note, with FAT, the pointer array can do double duty as a bitmap of free blocks

Linked Free Space List on Disk

- Nice, only free blocks are used to maintain the list
- So, no space wasted

- Probably a good idea to keep this list sorted
- Then, we can find contiguous free space