CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Indexed Allocation

Contiguous allocation led to external fragmentation

Linked allocation fixed that, but could not support efficient direct access due to cost of following pointers on disk

FAT fundamentally similar to linked allocation, but much more efficient

Next, look at alternative to linked allocation

Still split files into blocks and allow them to be scattered across disk

Indexed allocation brings all pointers for particular file into one location

Index block -- where all pointers stored

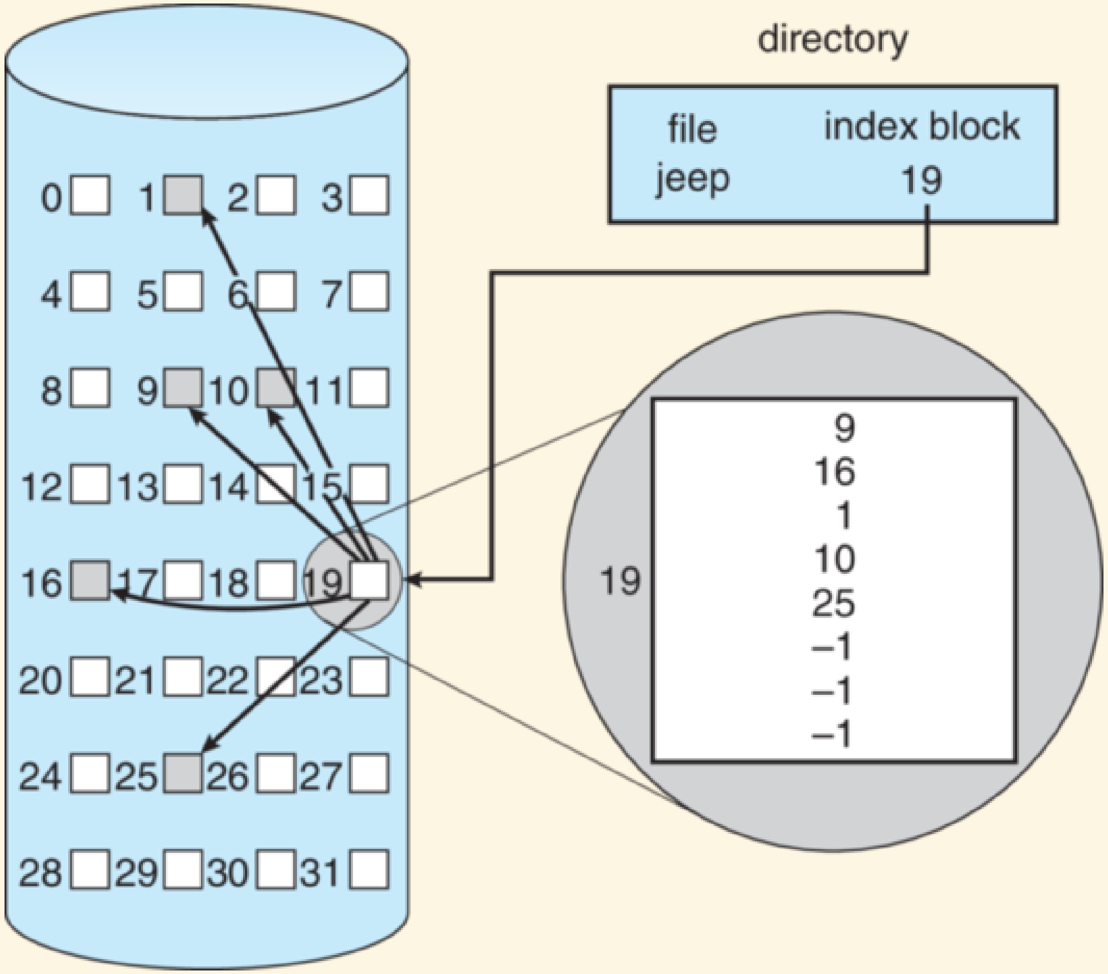
Sounds similar to FAT at first, but substantial changes occur because index block is per file rather than per file system

Each file has index block, stored as array of disk-block addresses

`i`th entry of index block is location of `i`th block of file

Direct access becomes very simple again

Find pointer in `i`th location of array and read location indicated by pointer



directory
file
index block
jeep
19
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
19
9
16
1
10
25
–1
–1
–1

Initially, every entry in index block is `null`

Easy to grow file -- whenever new block is written to, set corresponding entry in index block to pointer

No external fragmentation and easy direct access

Downside to indexed allocation is index block itself

Index block must be fixed size to make access simple

This creates upper bound on size of file because limited number of pointers stored

To allow for large files, index block must be large enough to store sufficient number of pointers

Wasteful for small files because most pointers will not be used

Index block results in more storage overhead than linked list because more pointers stored than necessary

Downsides of index block

- fixed upper bound on file size
- increased pointer storage overhead

Question becomes: how large to make index block?

Larger index block allows larger files but results in increased storage overhead

Methods exist to allow compromise between large vs small index blocks

In **linked scheme**, index block holds small number of pointers to file blocks and one pointer to next index block

Small files use just one index block, but index blocks can be linked to allow files of arbitrary size

Still have some wasted space because each index block holds, e.g., 100 pointers, which may not all be used

Random access in indexed allocation with linked scheme is somewhat slower, but still faster than linked allocation

One link must be followed per index file, rather than per block or cluster, and index files hold several pointers

With **multilevel index**, first-level index block points to set of second-level index blocks instead of directly to file blocks

To access specific file block, first access first-level index block to find location of corresponding second-level index block

Then, access second-level block, which holds actual address of file block

First-level index holds pointers to second-level indicices, many of which may be null if not needed

Overhead per file is now

- one first-level index block, and
- (n/k) second-level index blocks

where n is number of blocks in file and k is number of blocks addressable from single second-level index block

Additional overhead necessary to store first-level index block, but allows smaller files to take less space (fewer second-level indices) and larger files to take more (more second-level indices)

Still allows fixed size of file, but can be much larger

If each second-level block stores k pointers to file blocks and first-level block stores k pointers to second-level blocks, maximum file size is k * k blocks

Additional access overhead results from multilevel index

For each access to file block, first must access first-level index block, then corresponding second-level index block, before finally having address to access actual file block

So far, have been discussing two-level index

Could instead have second-level index point to third-level index that points to file blocks

Or, second-level -> third-level -> fourth-level -> file blocks

Each time degree of indirection increases, same tradeoffs made

Maximum file size increases

Overhead for small files increases

Number of accesses to get to particular block increases

Assuming all levels of index files hold k pointers, then as level of indexing increases from 1 to 2 to 3

- maximum file size increases from k to k^2 to k^3 file blocks
- minimum number of pointers allocated increases from k to 2k to 3k
- disk accesses to read file block increase from 1 to 2 to 3

Unix-based file systems use **combined scheme** that takes ideas from both linked scheme and multilevel index

Fixed number of pointers (e.g., 15) stored directly in inode

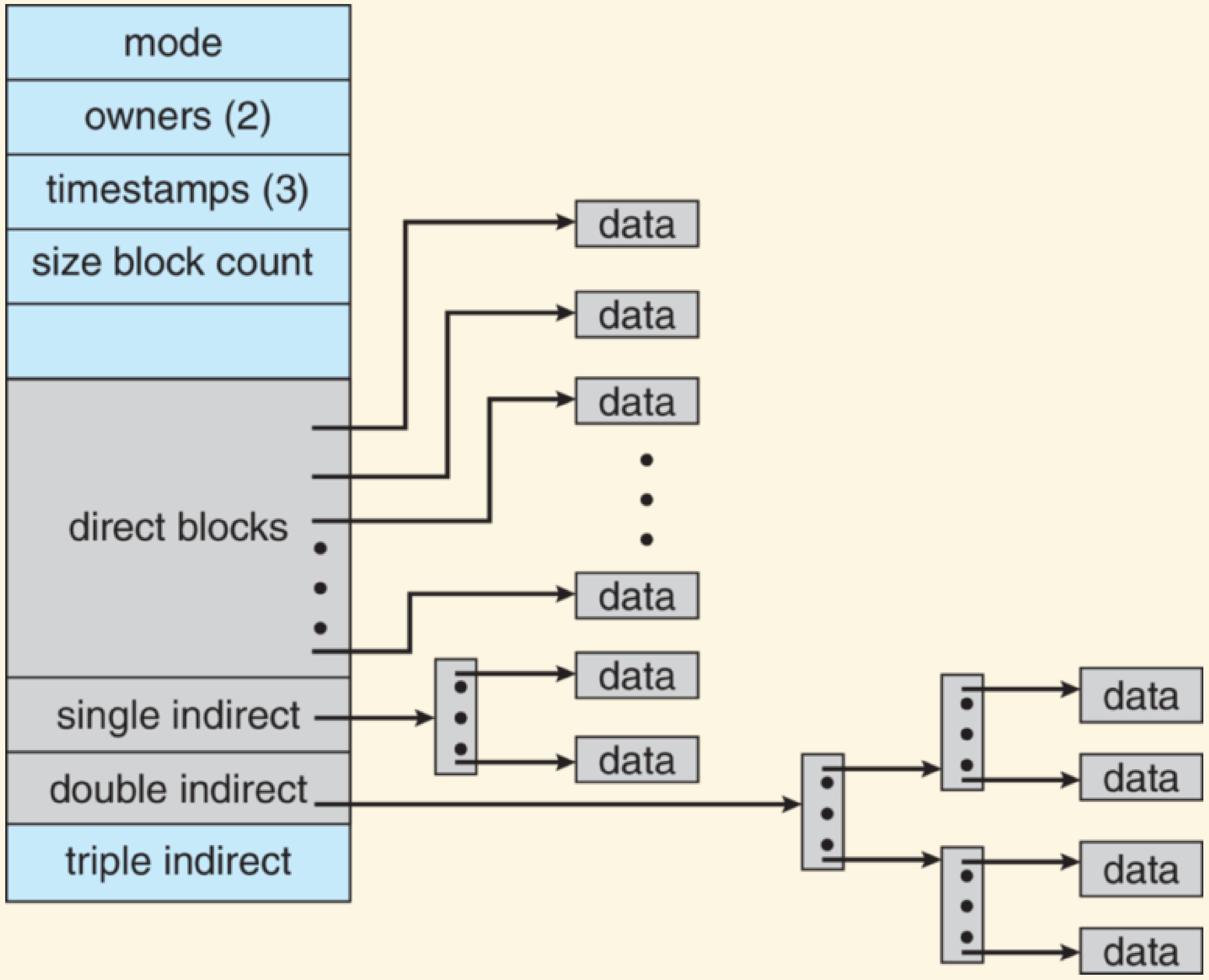
Pointers are of different types: some point directly to file blocks, others act as multilevel indices of various levels

For example, with 15 pointers stored, 12 pointers point directly to file blocks (just like one-level index block)

Next pointer is **single indirect block** -- pointer to index block that stores pointers directly to file blocks (like two-level index block)

Pointer 14 is **double indirect block** -- pointer -> pointers -> pointers -> file blocks (like three-level index block)

Final pointer is **triple indirect block** -- pointer -> pointers -> pointers -> pointers -> file blocks (like four-level index block)


mode
owners (2)
timestamps (3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

In this scheme, small files require little storage and can be accessed quickly through direct access to blocks

Files can grow quite large, but require higher levels of indirection as they do so

Maximum file size still technically fixed, but extremely large