

Terminology

The things on the disk that hold information about files – file table entries (textbook calls them file control blocks FCB).

There are several file tables held in different types of memory and accessed by different entities (as we will see).

In this case I mean the on-disk representation (which could be a directory) that holds the file information, attributes and other pointers.

In UNIX this is the inode.

In NTFS this is the MFT file record.

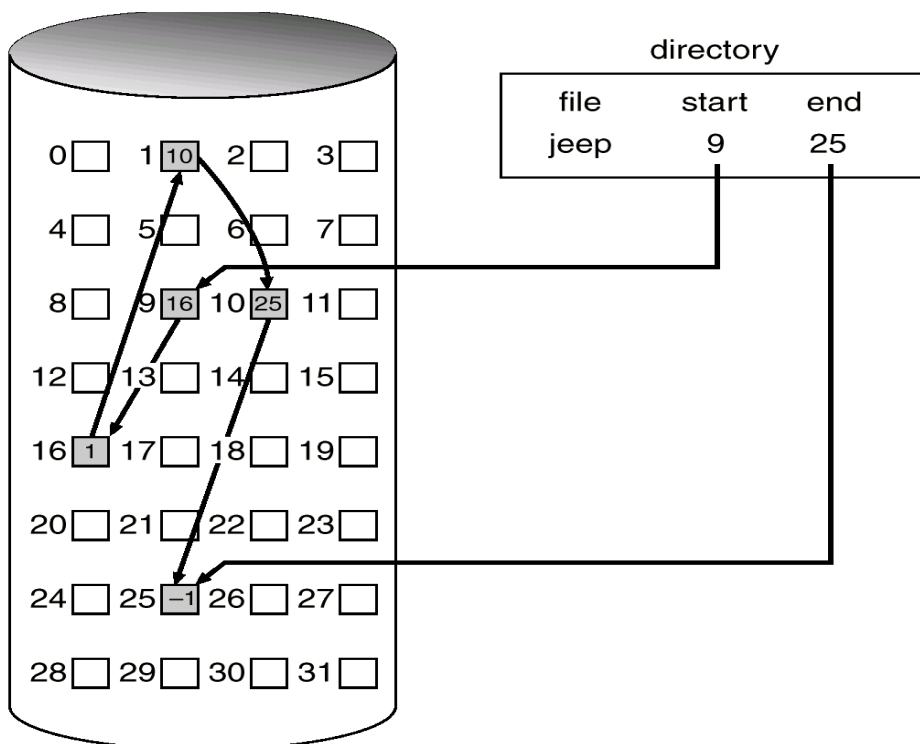
In MS-DOS this is the directory.

Linked allocation

The second major method (see lecture 16) to keep track of the blocks associated with each file is linked allocation.

A small amount of space is set aside in each block to point to the next (and sometimes previous as well) block in the file.

The directory entry holds a pointer to the first block of the file and probably the last as well.



Pros and Cons of linked allocation

Pros

- Simple.
- No external fragmentation.

Cons

- To directly access a particular block, all blocks leading to it have to be read into memory. So random access is slow (at least until pointer data is stored in memory).
- Damage to one block can cause the loss of a large section of the file (or damage to two blocks if doubly-linked).

We could store the filename and relative block number in every block.

- Because of the pointers each data block holds a little less data than it could. Only a problem with small blocks.

MS-DOS & OS2 FAT

The MS-DOS File Allocation Table (FAT) is a collection of linked lists maintained separately from the files. One entry for each block on the disk.

A section of disk holds the table.

Each directory entry holds the block number of the file's first block.

This number is also an index into the FAT.

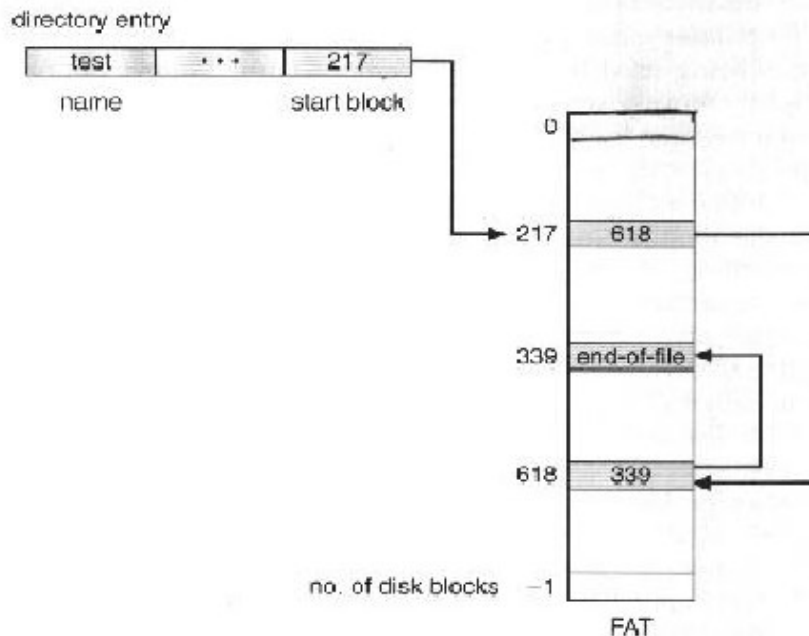
At that index is the block number of the second block.

This number is also an index into the FAT.

etc...

Over the years the number of bits used to index the FAT changed (9 to 12 to 16 to 32) as disks got larger. So the size of the FAT and the maximum number of clusters on the disk grew.

FAT advantages



Accessing the FAT data for a file requires far fewer disk accesses than for normal linked access.

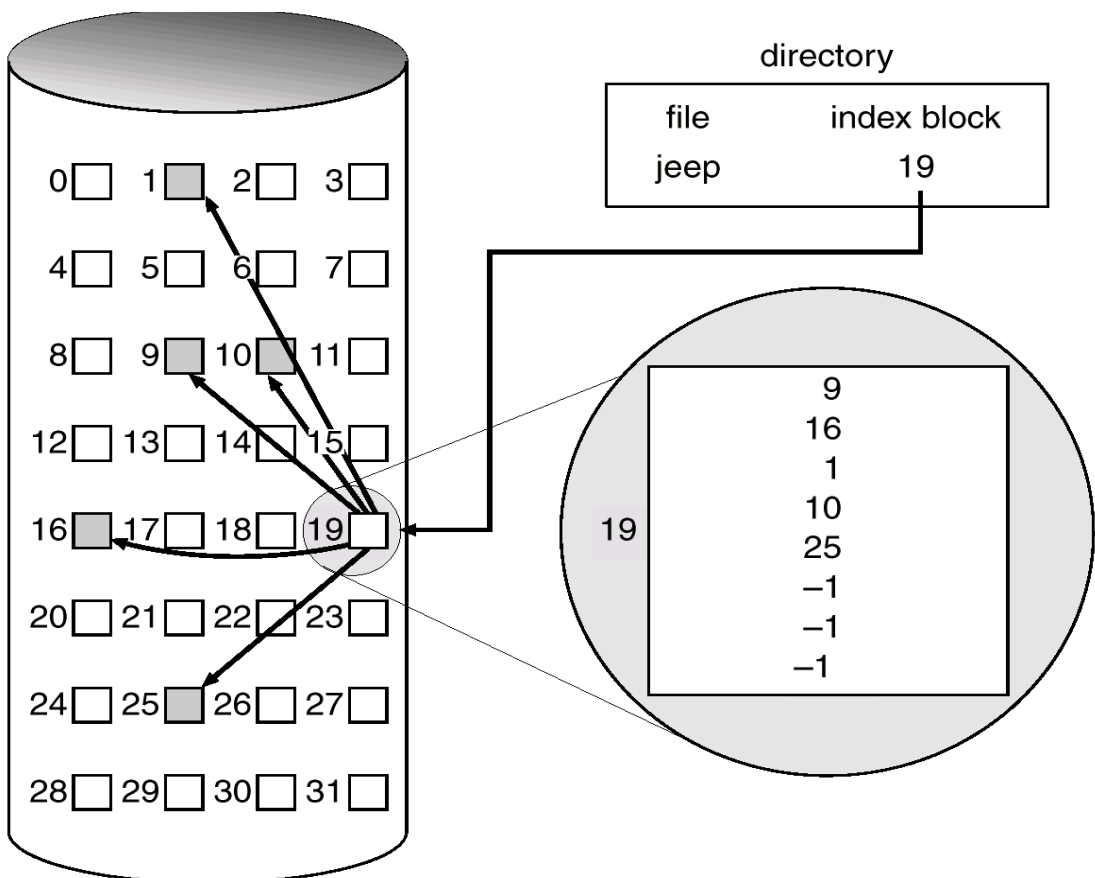
One block of FAT data might hold information for several blocks in the file.

If we have enough memory to cache the entire FAT we can determine the block numbers for any file with no extra disk access.

Unfortunately as the FAT gets larger it is more difficult to cache the whole thing and so it becomes more common for a block access to require multiple FAT block reads.

Indexed allocation

A partial way around this is to keep all block numbers in a contiguous table for each file.



Pros and Cons of Indexed Allocation

Pros

- Good for direct access.
- Information for each file is kept in one area.
- No external fragmentation.

Cons

- File size limited by the number of indices in the index table.
But we can extend this in a number of ways.
- If we lose an index block we have lost access to a whole chunk of the file.

Extending index blocks

If the file has more blocks than we can reference from the index block we need to have some way of connecting to more index blocks.

We can link index blocks together (like the linked allocation of files). Similar pros and cons.

We can have multiple levels of index blocks.

- The first level points to index blocks.
- The second level points to actual blocks.

e.g. If we have blocks of size 8K, a block address of 4 bytes and we have a two level system we can address files of up to 32 gigabytes.

The indirect index block points to 2048 index blocks.

Each index block points to 2048 actual file blocks.

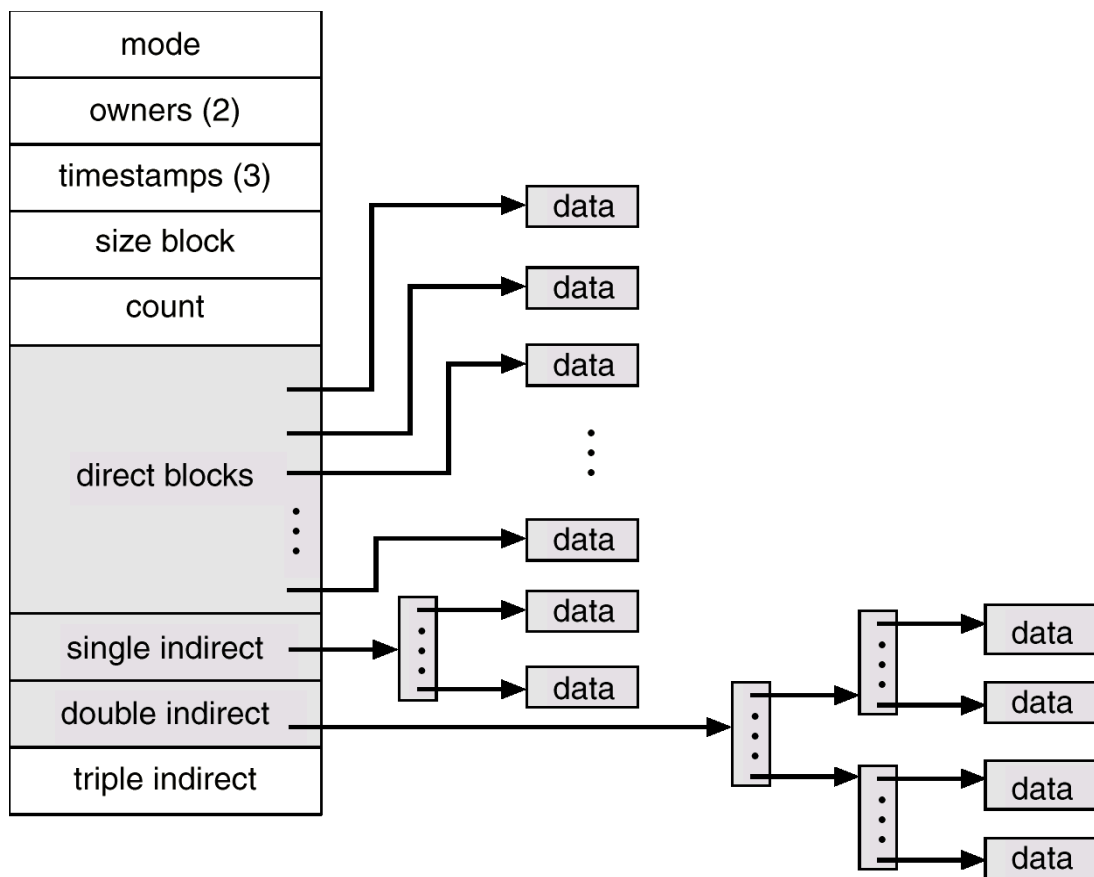
So we can have 4194304 actual blocks in a file.

How do we find the address of a particular block?

UNIX index block scheme

In order to minimise the number of blocks read to find the actual block (especially with small files) several versions of UNIX don't use extra indirect blocks until necessary.

NTFS goes one step further and stores small files $< 1\text{K}$ in the MFT file record itself.



Keeping track of free blocks

Whenever a new block is requested for a file we need to check to see if there is an empty block. How do we keep track of all the empty blocks?

There are lots of them – my 40Gb disk with 4K blocks has about 10 million blocks.

We could link the free ones together like one large empty file using the linked allocation method of slide 2.

With a linked list it is trivial to find the first free block.

But it is not efficient if we want several blocks at a time (especially if we want them to be contiguous).

We can maintain a bitmap (or bit vector).

Where each bit represents a used or free block (1 represents free).

For my disk this takes up more than a megabyte of space.

We can keep a list of start points and lengths.
A bit like RLE (run length encoding).

UNIX 4.3BSD file system

Early versions of UNIX maintained a device wide free list of blocks in a stack, whereby the next allocated block is the most recently returned one. What are the consequences of this?

Also all inodes were stored in one place. What are the consequences of this?

Cylinder groups were introduced in order to improve file access and reliability.

Cylinder groups - one or more consecutive cylinders (disk head seek time is minimised so access to all blocks in the same group is fast). Inodes and free lists stored with each cylinder group.

Each cylinder group has a copy of the superblock for the "file system" (UNIX terminology for a partition or disk device.)

Tries to keep blocks from the same file within the same cylinder group. But larger files are split over different cylinder groups.

Before next time

Read from the textbook

12.4 – Allocation methods

12.5 – Free-space management

12.6.1 - Efficiency