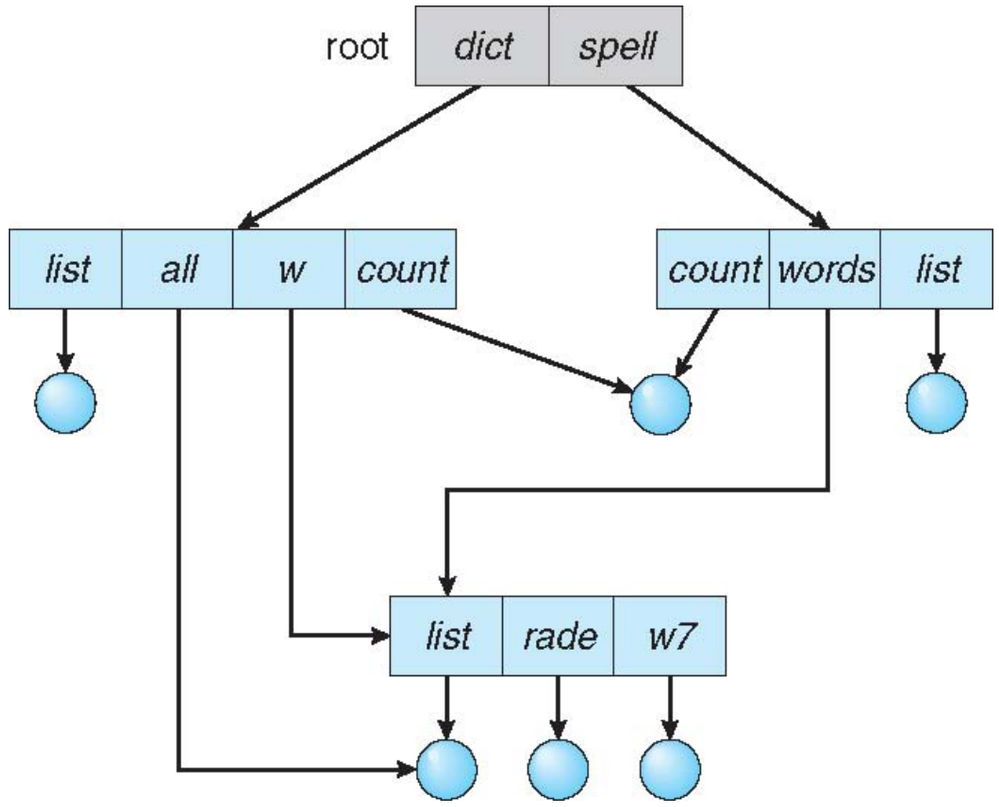
File System Implementation

- **Ch13.3: Disk and Directory Structure**
  - Acyclic Graph Directory Structure
- Ch14.4: Allocation Methods

Acyclic Graph Directory Structure

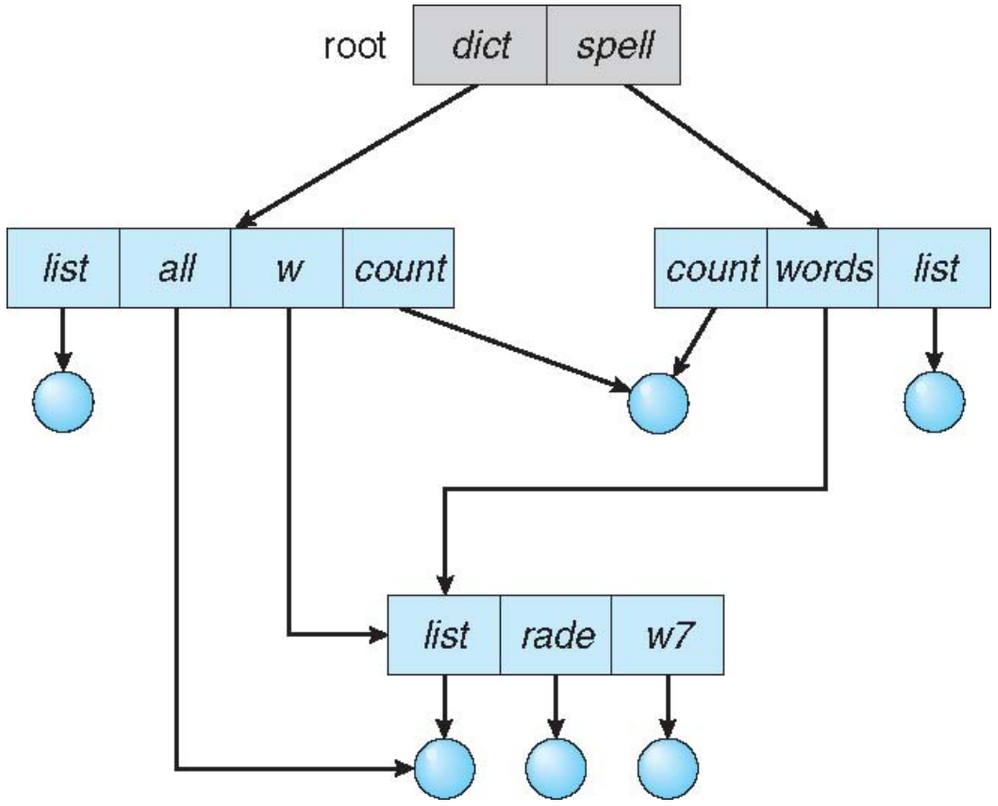
- An acyclic graph is a graph with no cycle and allows to share subdirectories and files
- It is a natural generalization of the tree-structured directory.
- If any user makes some changes in the subdirectory it will reflect in both subdirectories



- Advantages
  - Allowing file sharing
  - Searching is easy due to different paths.
- Disadvantages
  - Sharing the files via linking, in case of deleting it may create the problem
  - If the link is **softlink** then after deleting the file we left with a dangling pointer.
  - In case of **hardlink**, to delete a file we have to delete all the reference associated with it.

Cycles in the Directory Graph

- If we allow shared subdirectories and files
  - Directories can appear in multiple places in the file system we can get cycles
  - We don't want to fall into infinite loops if we traverse the file system e.g. to search for a file.



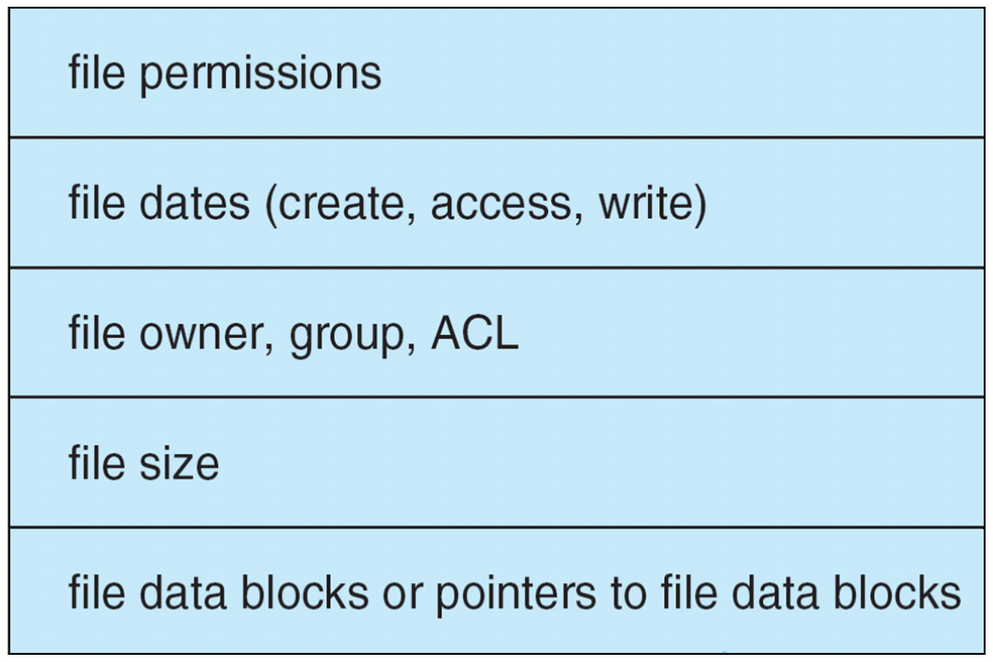
Acyclic Graph Directory Structure

We need some way of uniquely identifying directories.

In UNIX directories can only be linked with symbolic links and algorithms count symbolic links on directories to stop infinite recursion.

Terminology

- Per-file File Control Block (**FCB**): contains many details about the file, such as inode number, permissions, size, dates.
  - In UNIX this is the inode (Information node or Index node)
  - In NTFS this is the MFT (master file table) file record
  - In MS-DOS this is the directory entry



Unix Directory Entries

- **UNIX** keeps the file attributes and location information of each file in a separate structure – the **inode** (Information node or Index node).
- The inode also keeps a count of the number of *hard* links to the file.
- The inode table is an array of inodes stored in one or more places on the disk.
- UNIX directory isn't much more than a table of names and corresponding inode numbers.

NTFS Directory Entries

- **Windows** keep all file and folder information in the **MFT** (Master File Table).
  - Each file has at least one file record consisting of the attributes from before
  - Folders have an indexed table of file information.
  - Each folder entry includes the file name, a pointer to the file's MFT entry, plus the most commonly referenced file attributes e.g. created and modified dates, and length.
- So much of the information in the files MFT record is duplicated in the directory entry.
- This explains the hardlink behaviour discussed earlier.
- An MFT entry for a file or directory:

Standard Information	File or Directory Name	Security Descriptor	Data or Index	

File System Implementation

- Ch13.3: Disk and Directory Structure
- **Ch14.4: Allocation Methods**
    1. Contiguous Allocation
    2. Linked Allocation
    3. Indexed Allocation
    4. Index Block Scheme

Allocation Methods

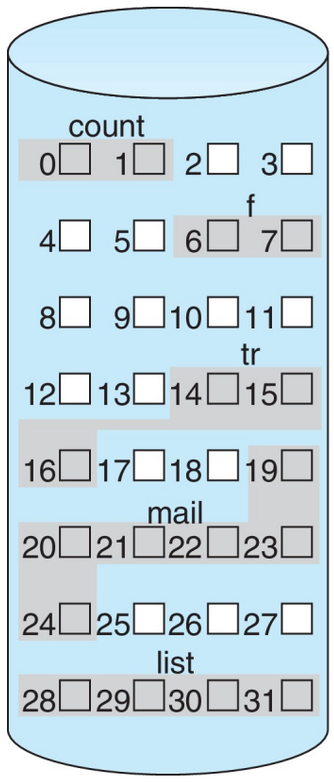
- Allocation Methods - how disk blocks are allocated for file?

- Whether it is in the directory entry or in some other data structure there must be a place on the disk which points to the blocks allocated to each file.

- This can be done in many different ways. It depends on how blocks are allocated to files.

Contiguous Allocation

- Each file occupies set of contiguous blocks
- Some early file systems always allocated files in contiguous blocks
  - Only the start block and the number of blocks needed to be stored
  - A table of block usage needs to be held for each file
- Algorithms to find a large enough hole
  - First fit
  - Next fit
  - Best fit
  - Worst fit
  - Buddy algorithm (10.8.1 for memory)
- First fit is commonly used



directory

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

Trade-offs: Contiguous Allocation

- Advantages:
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Great for sequential and random access

- Disadvantages:
  - External fragmentation – lots of holes outside files (and too small to hold bigger files)
  - How much space should be allocated for each file?
    - If we allocate too little it is a pain to increase the size.
    - If we allocate too much we are wasting space – internal fragmentation

- Extent-based Systems (e.g., NTFS) – collections of contiguous allocation

Linked Allocation

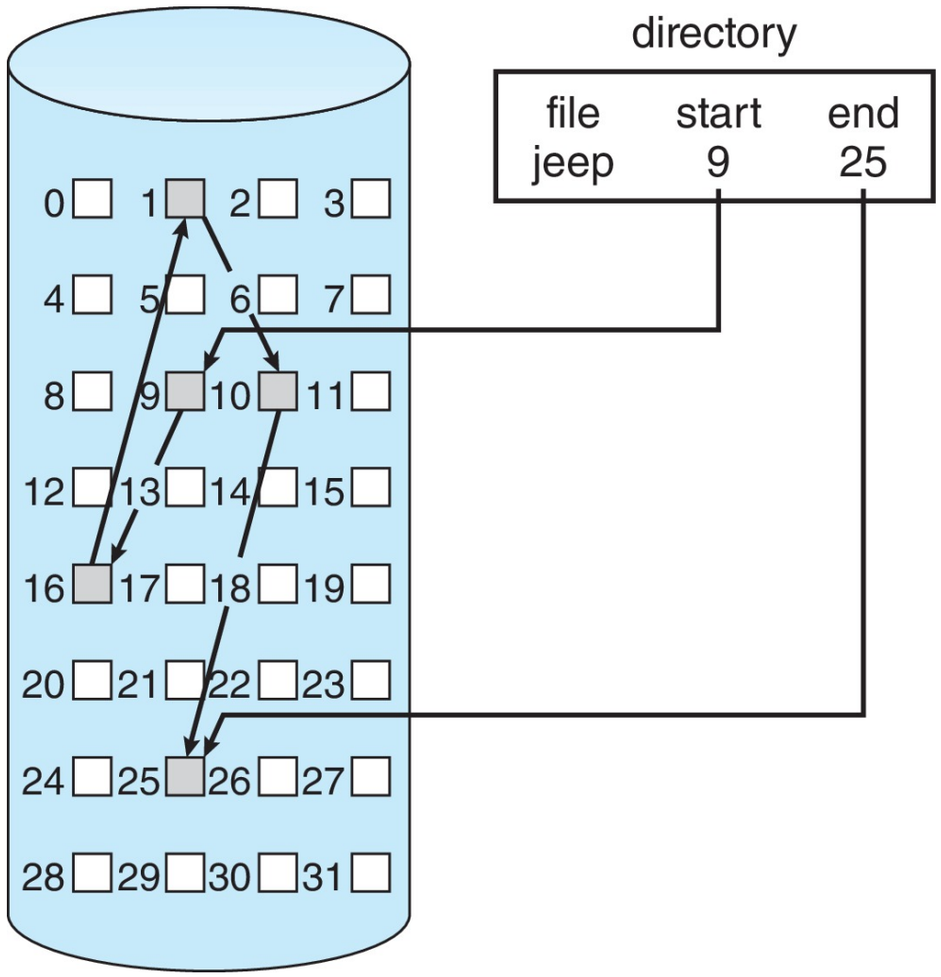
- Keep track of the blocks associated with each file as a **linked list of blocks**

- Each block contains pointer to next block
  - 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.

- Free space management system called when new block needed

Positioning Delay

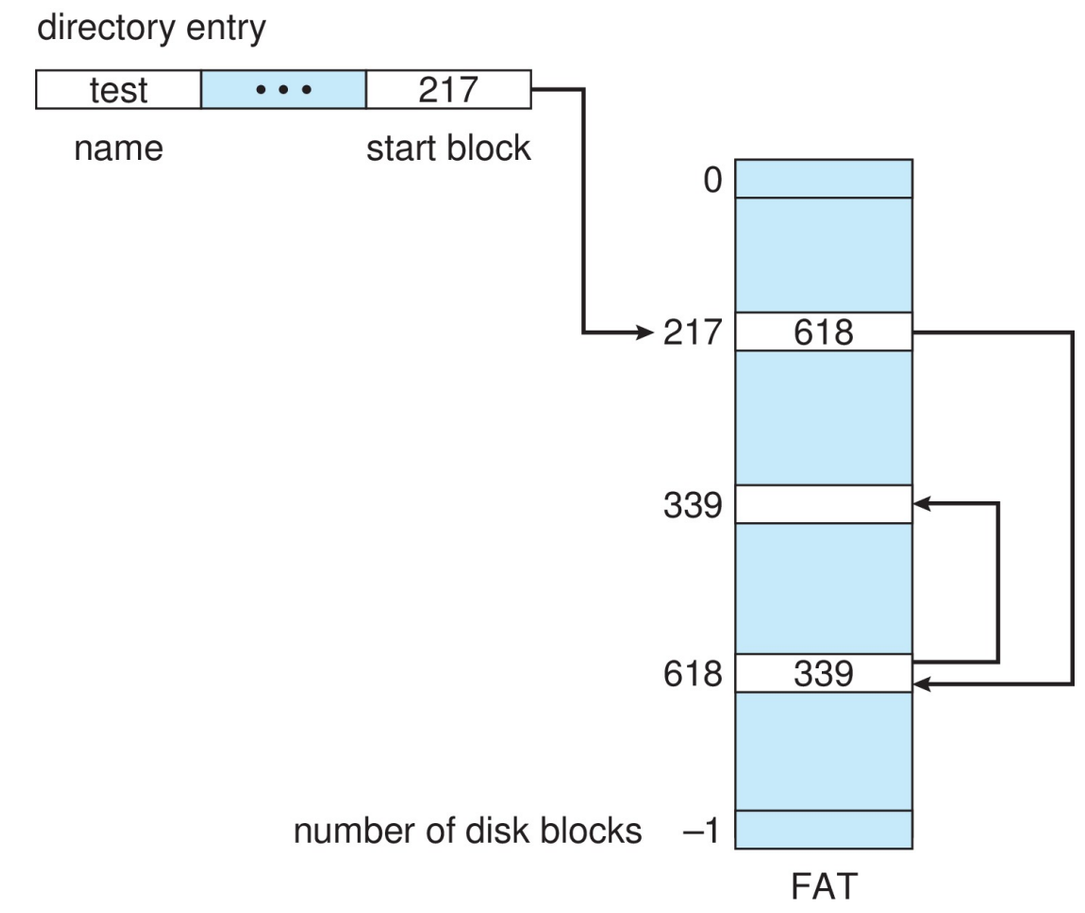


Trade-offs: Linked Allocation

- Advantages
  - Simple
  - No external fragmentation
  - Good for sequential
- Disadvantages
  - Bad for random access: Locating a block can take many I/Os and disk seeks
    - To directly access a particular block, all blocks leading to it have to be read into memory. So direct (random) access is slow (at least until pointer data is stored in memory).
    - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
    - 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 a file id and relative block number in every block.
  - Take up space
    - Because of the pointers each data block holds a little less data than it could. Only a problem with small blocks.

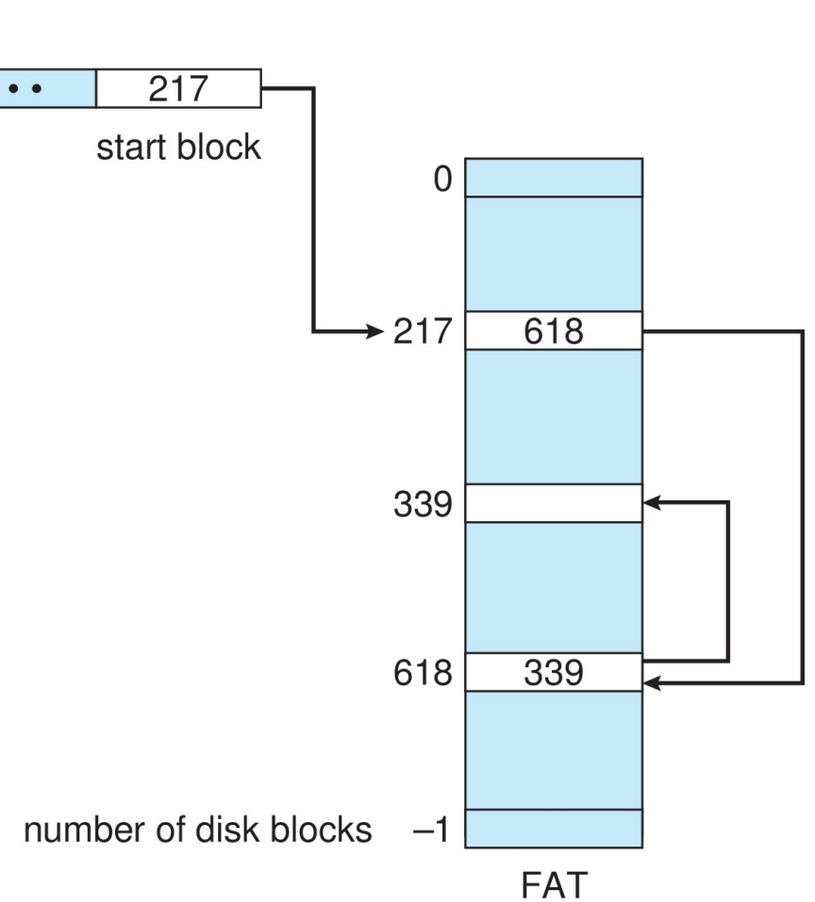
MS-DOS & OS2 FAT

- 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
- Increase 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 allocation units on the disk grew.



Trade-offs: FAT

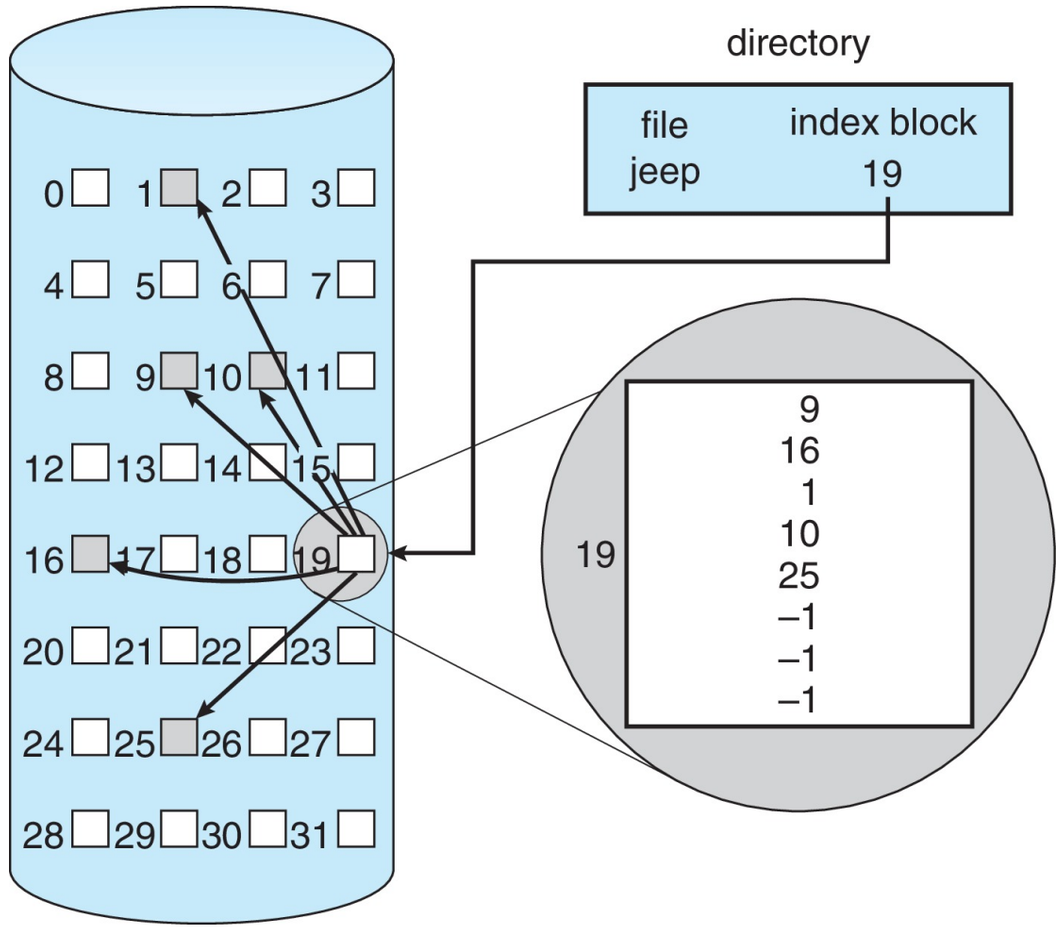
- Advantages
  - Accessing the FAT data for a file requires far **fewer disk accesses** than for normal linked access.
  - One block of FAT data might hold information for several blocks in the file.
    - If we have enough memory to cache the entire FAT we can determine the block numbers for any file with **no extra disk access**.
- Disadvantages
  - As FAT gets larger it is more difficult to cache the whole thing and so it becomes more common for a single block access to require multiple FAT block reads.



Indexed Allocation

- Each file has its own index block(s) of pointers to its data blocks

- Keep all block numbers in a contiguous table for each file.



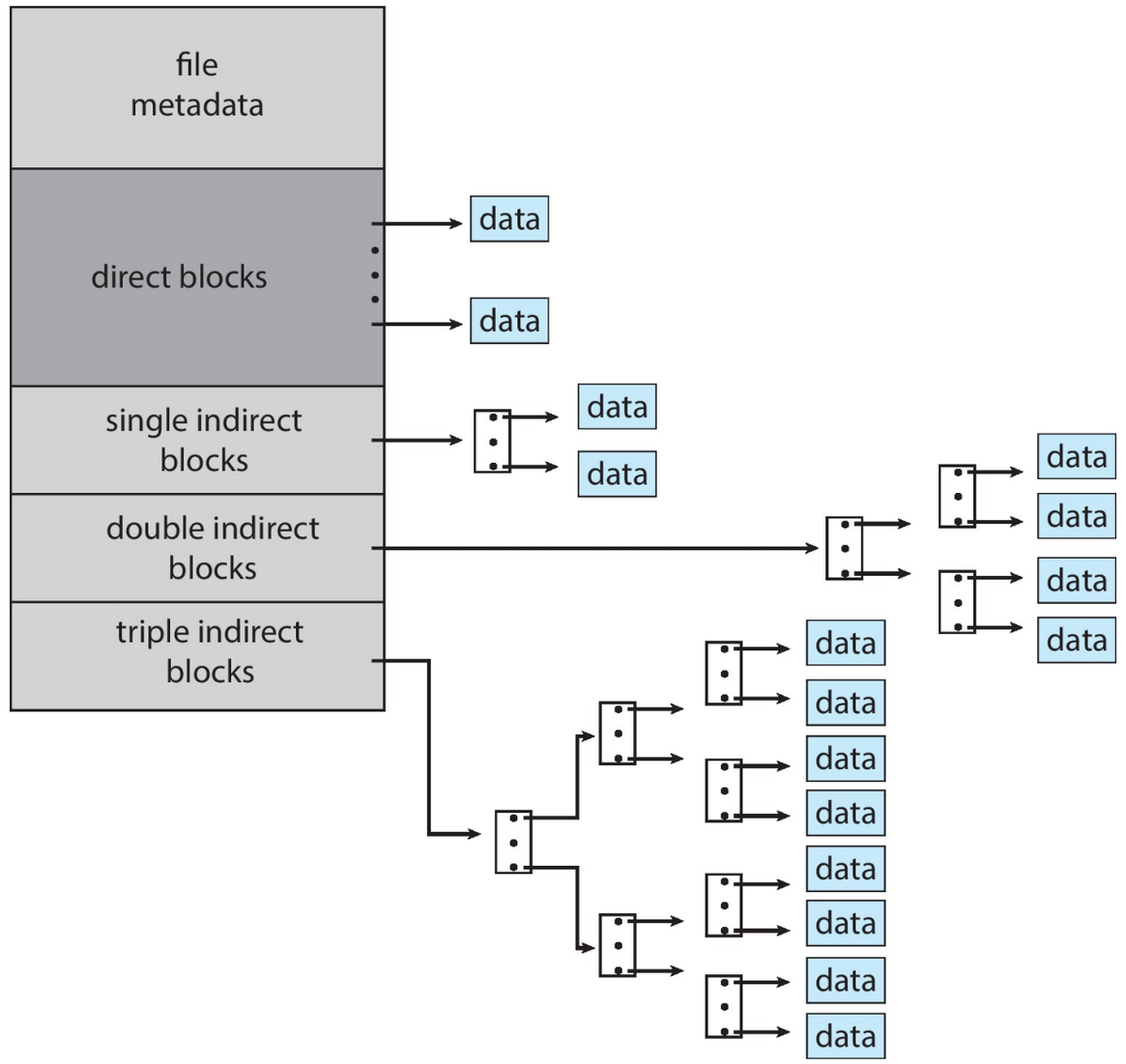
Trade-offs: Indexed Allocation

- Advantages
  - Good for direct access
  - No external fragmentation

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

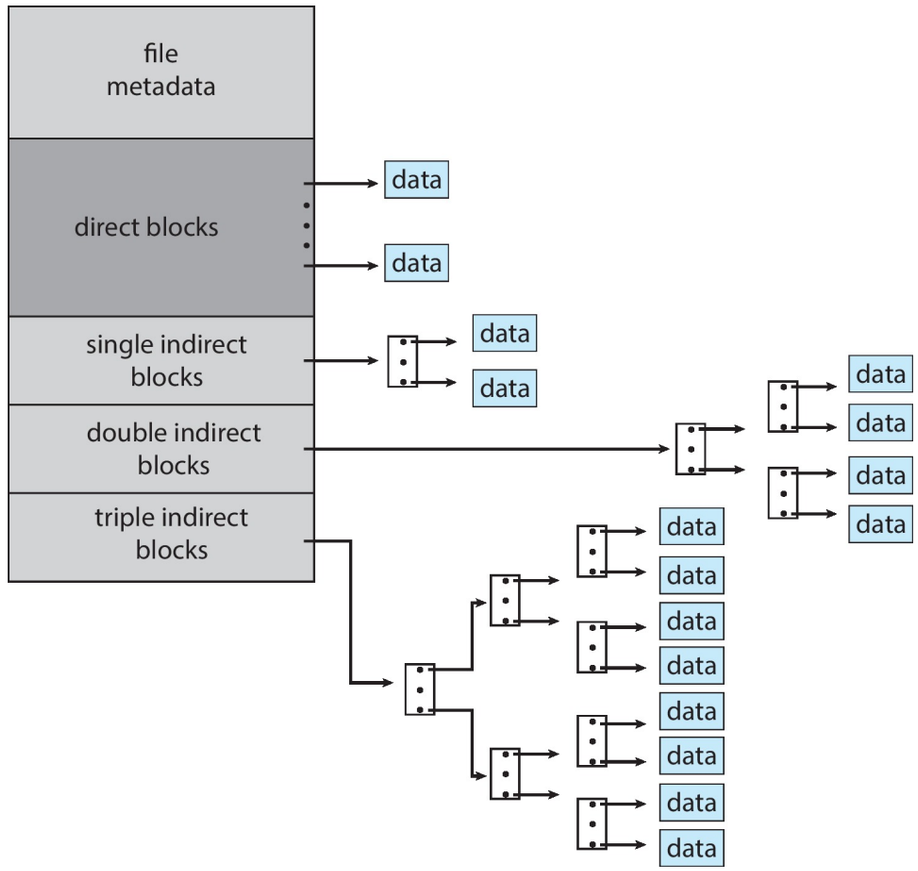
Index Block Scheme

- Combined Scheme: UNIX UFS
- 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 unless necessary.
- 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).
- We can have multiple levels of index blocks.
  - The first level points to index blocks.
  - The second level points to actual blocks.



Example: Index Blocks

- Consider a UNIX file system with
  - 12 direct pointers,
  - 1 indirect pointer,
  - 1 double-indirect pointer, and
  - 1 triple-indirect pointer in the i-node.
- Assume that disk blocks of size 8K, a block address of 4 bytes
- Q: What is the largest possible file that can be supported with this design?



- Answer:

```
Number of ptrs/block = 8K/4 = 2048
(12 * 8KB) + (2048 * 8KB) + (2048 * 2048 * 8KB) + (2048 * 2048 * 2048 * 8KB)
```

direct index block **indirect index block** **double-indirect index block** **triple-indirect index block**

NTFS Extents

- NTFS keeps all index information in the MFT. Rather than storing all block numbers it stores **extents**.
- A cluster is a number of blocks (2, 4, 8, etc.). An NTFS volume is seen as an array of clusters.
- An extent is a start cluster number and a length (number of clusters).
- e.g.

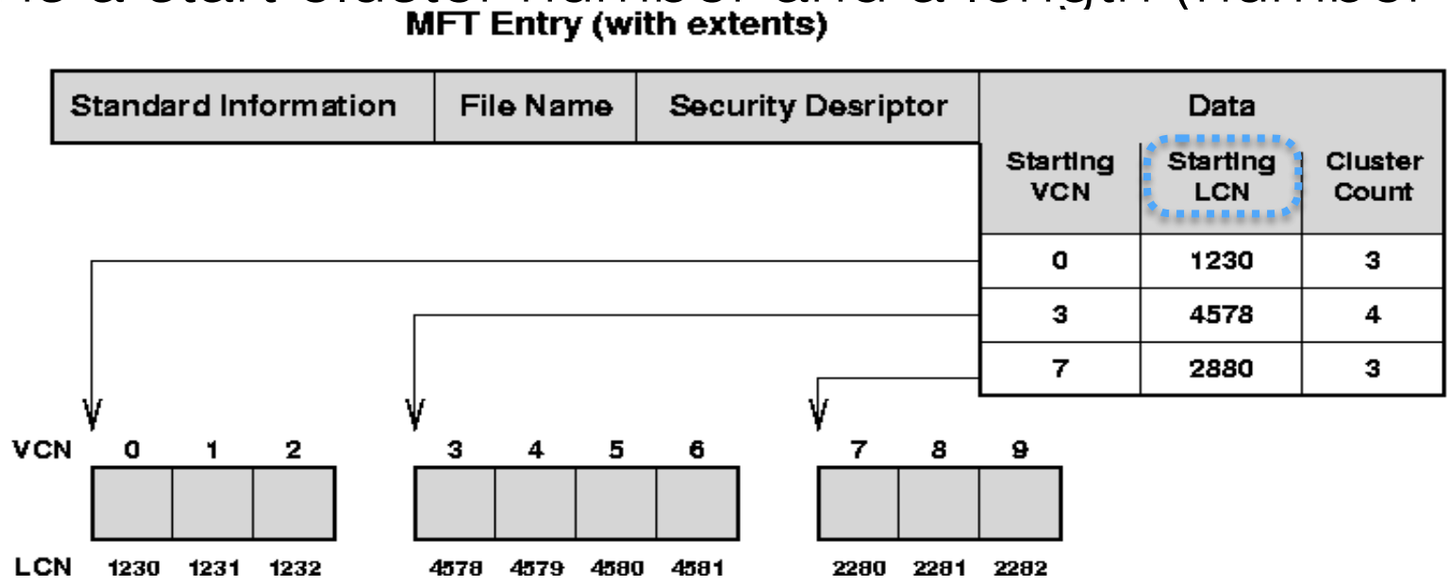


Diagram from http://www.cs.wisc.edu/~bart/537/lecturenotes/s26.html

Extra MFT entries are used as required.

File System Implementation

- Ch14.5: Free-Space Management
- Ch14.6: Efficiency and Performance