CSCI315 – Operating Systems Design

Department of Computer Science
Bucknell University

File System Implementation 2

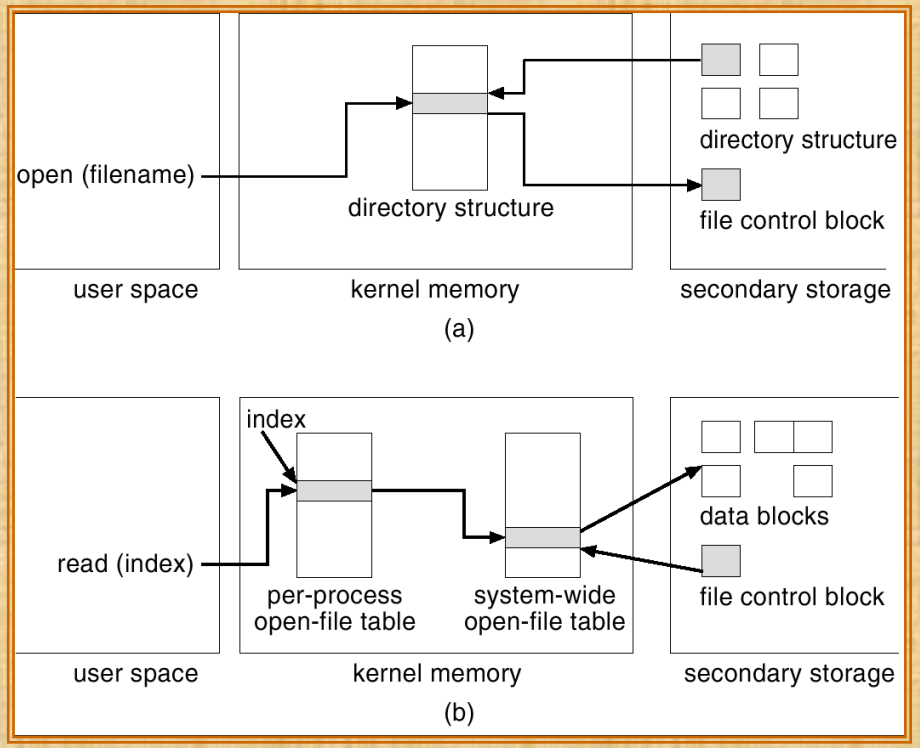
Ch 14.4-14.5

This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.
Xiannong Meng, Fall 2021.

Review

- We discussed the file control block (FCB) in Linux, that is the **inode**.
- We also discussed general directory structure in Linux.
- Today we will look at some other implementation issues.
  - Virtual file system, and
  - File block allocation strategies

In-Memory File System Structures



file open

file read

Per-process open-file table part of PCB

Directory Implementation

The directory is a **symbol table** that maps file names to file control block (inode) which has pointers that lead to the blocks comprising a file.

- **Linear list** of file names with pointer to the data blocks:
  - simple to program, but…
  - time-consuming to execute.
- **Hash Table:**
  - decreases directory search time,
  - *collisions* – situations where two file names hash to the same location,
  - fixed size.

An Example

- Consider ‘open(“hello.txt”, O_RDONLY)’ in
  - http://www.eg.bucknell.edu/~cs315/F2021/meng/code/files/file-stream.c
- Where does the file “hello.txt” and “file-stream.c” reside?
- On any remote school Linux computer!
  - linuxremote.bucknell.edu
  - Use “df” to find out (you may have to use “grep” to look for the string ‘cs315’
- Implication?
- File system has to work with networked files

Examples of “df” Results

```
File  Edit  View  Search  Terminal  Help
Filesystem                                                     1K-blocks
      Used  Available Use% Mounted on
devtmpfs                                                        32889408
         0   32889408   0% /dev
tmpfs                                                           32903432
      5772   32897660   1% /dev/shm
tmpfs                                                           32903432
   3292376   29611056  11% /run
tmpfs                                                           32903432
         0   32903432   0% /sys/fs/cgroup
/dev/mapper/centos-root                                         47158788
  14127676   33031112  30% /
/dev/sda1                                                        1038336
    302552     735784  30% /boot
tmpfs                                                            6580688
         4    6580684   1% /run/user/987
unixspace.bucknell.edu:/vol/vol1/home/linux/accounts/users/jdreese  10307921536
5894831424 4413090112  58% /home/jdreese
tmpfs                                                            6580688
       360    6580328   1% /run/user/3593
:
```

df | less
on linuxremot

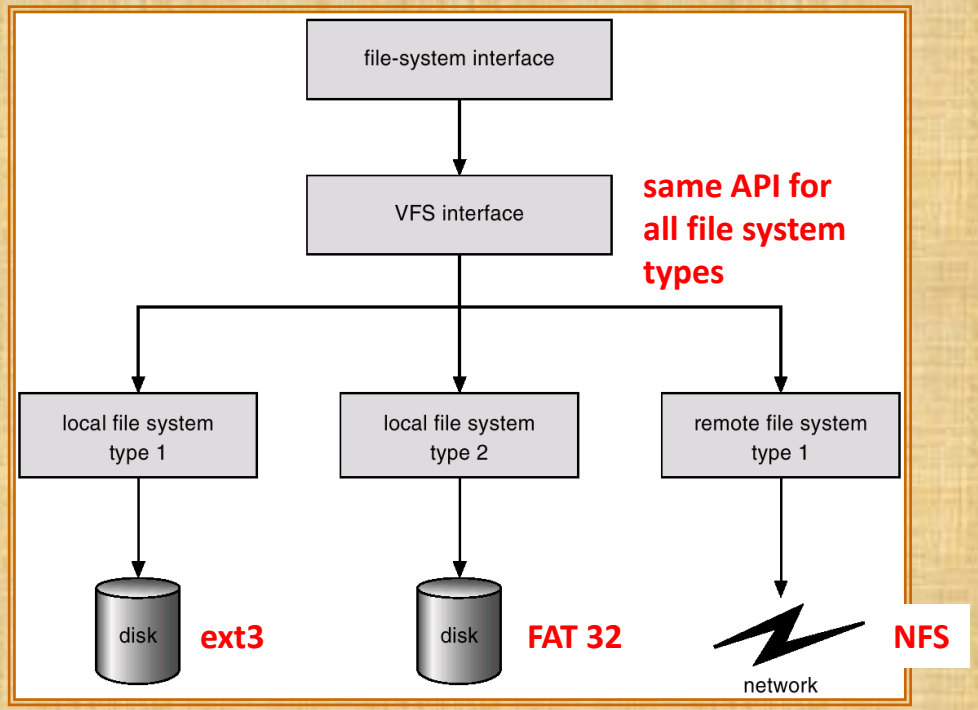
df | grep cs315
on linuxremote

```
xmeng@linuxremote2:/home/cs315/F2020
File  Edit  View  Search  Terminal  Help
[bash xmeng@linuxremote2 F2020]$ df | grep cs315
unixspace.bucknell.edu:/vol/vol1/home/linux/accounts/users/cs315    10307921536
5894847808 4413073728  58% /home/cs315
[bash xmeng@linuxremote2 F2020]$
```

Virtual File Systems

- **Virtual File Systems (VFS)** provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



Virtual File System Implementation

- For example, Linux has four file object types:
  - *inode, file, superblock, dentry*
    - *dentry: directory entry, e.g., /usr/bin/ls, both usr and bin are dentries*
    - *https://github.com/torvalds/linux/blob/master/include/linux/dcache.h*
- VFS defines set of operations on the objects that must be implemented, inode -> vnode
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:
    - **`int open()`** -- Open a file
    - **`int close()`** -- Close an already-open file
    - **`ssize_t read() --`** Read from a file
    - **`ssize_t write()`** -- Write to a file
    - **`int mmap()`** -- Memory-map a file

https://unix.stackexchange.com/questions/4402/what-is-a-superblock-inode-dentry-and-a-file

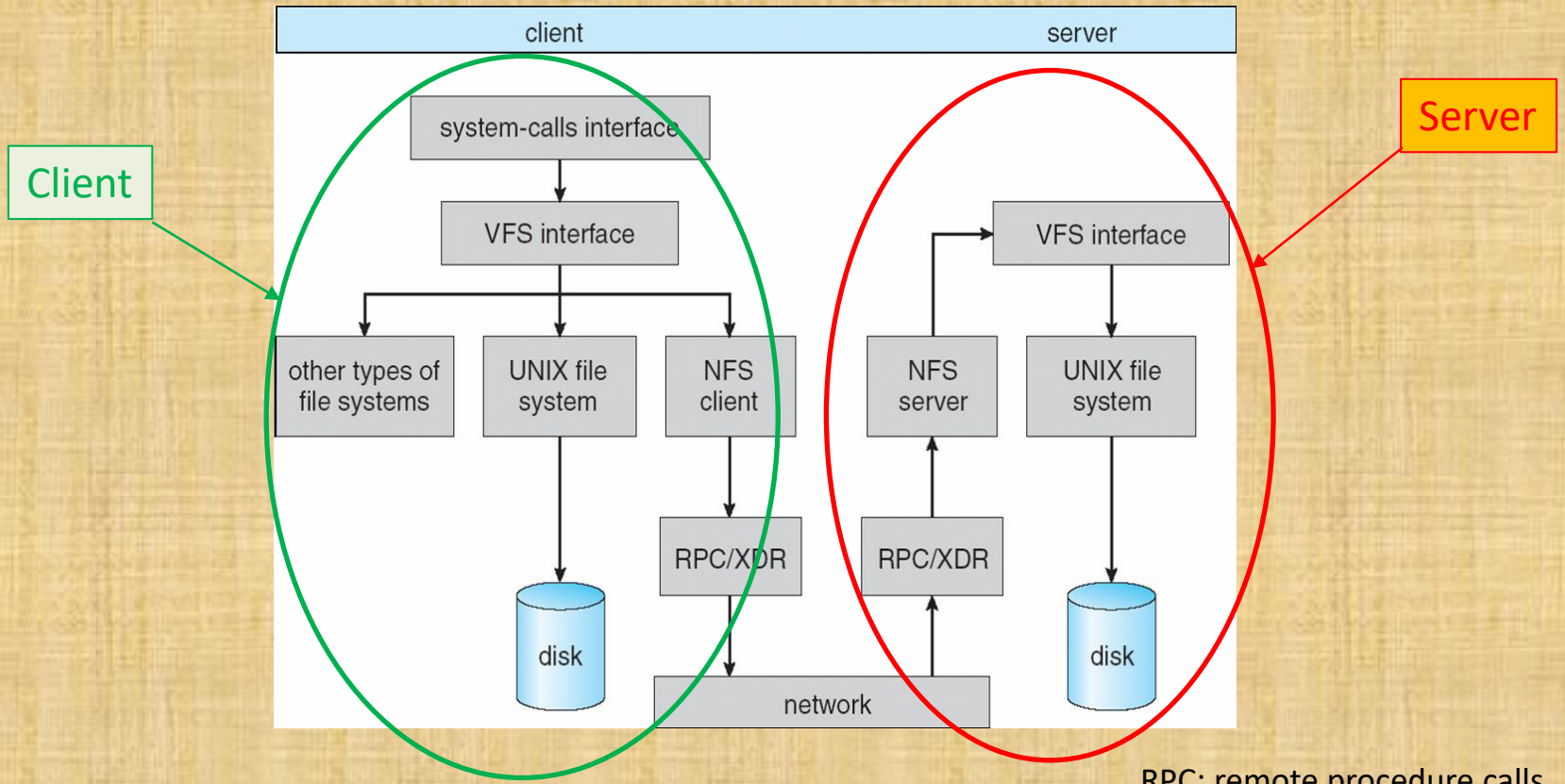
NFS (Network File System) Protocol

- Provides a set of remote procedure calls (RPC) for remote file operations.
- Early versions of NFS servers are **stateless**; each request has to provide a full set of arguments; NFS V4 is very different, stateful, supporting many more operations
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of File System Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture



RPC: remote procedure calls
XDR: external data representation

Allocation Methods

An **allocation method** refers to how disk blocks are allocated for files. We'll discuss three options:

- Contiguous allocation,
- Linked allocation,
- Indexed allocation.

A PC Disk Example

https://commons.wikimedia.org/wiki/File:Laptop-hard-drive-exposed.jpg

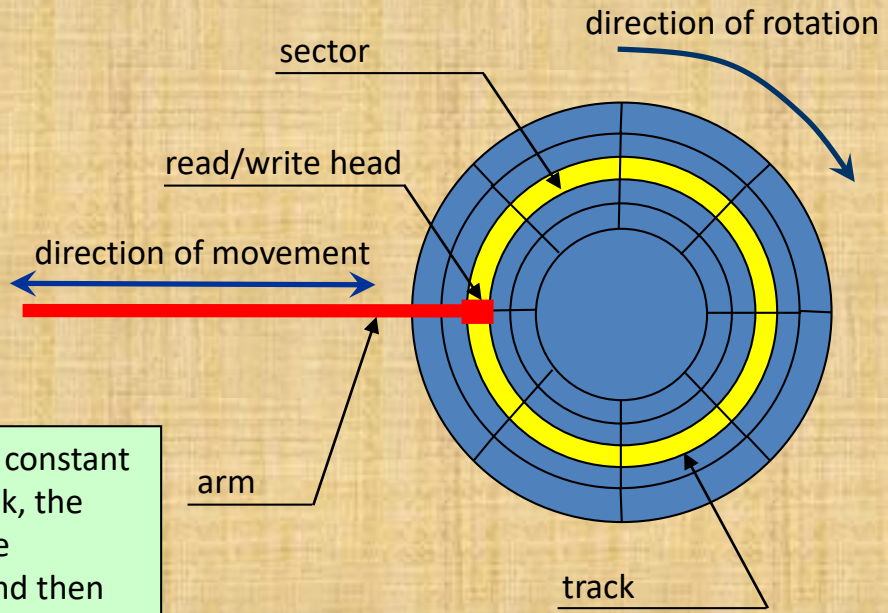
Disk Structure

Points to consider:

Sector sizes (number of bits per sector) are fixed in most disks, which means the data density is lower on outside tracks.

Newer formats, e.g., *zone-bit-recording*, uses variable size sectors so sectors have similar data density.

The disk rotates at a constant speed. To find a block, the head is moved to the appropriate track, and then the correct sector is found as the disk rotates.



Organization of a disk surface

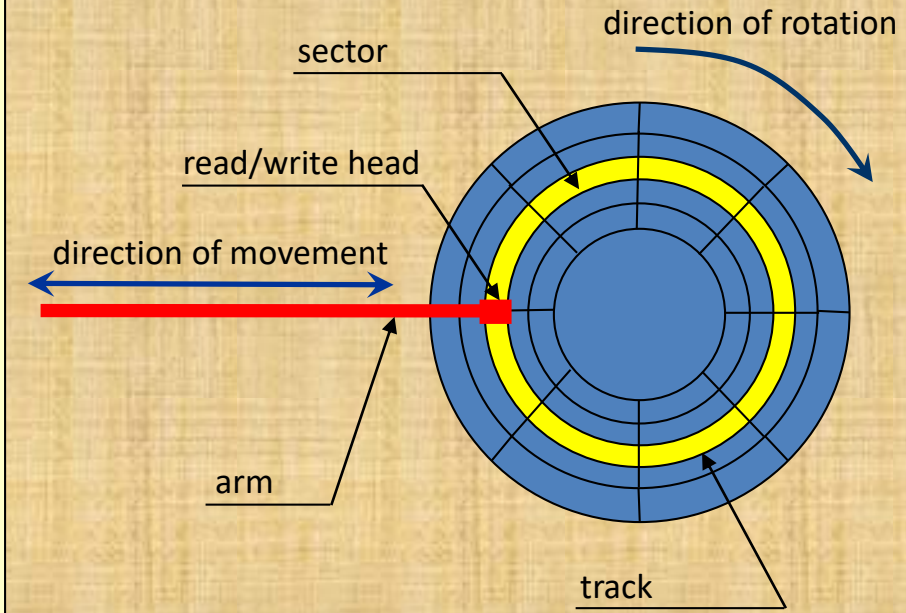
Disk Structure

The disk rotation is given in rotations per minute (RPM).

The time to find a track is proportional to the distance the head must travel.

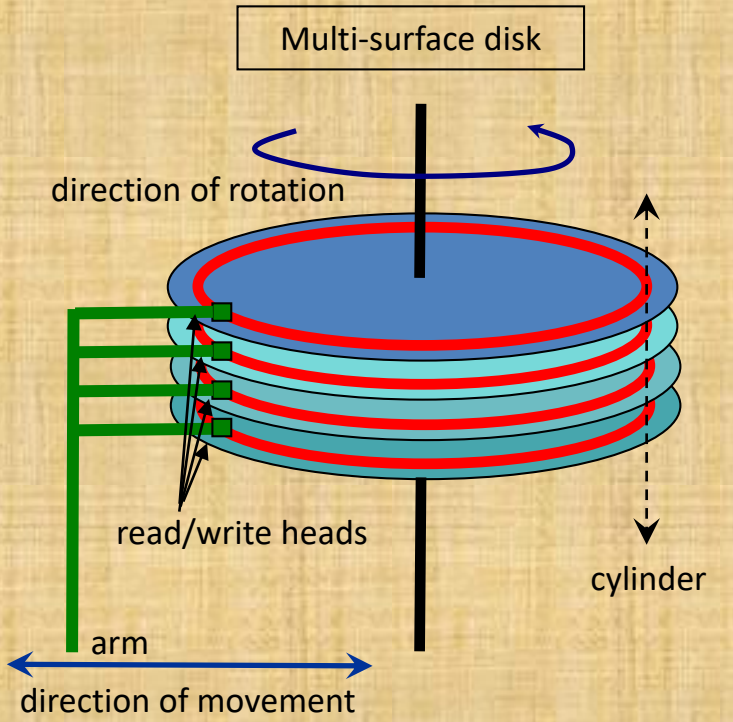
The average time to find a sector within a track is roughly **half the time for a full rotation**.

Question: If the time to move from track i to track $(i+1)$ is given by δ, assuming that the disk head is at track 0 (all the way out), could you calculate the time to get to sector 4 in track 5?



Organization of a disk surface

Disk Structure



A cylinder is the collection of all the same tracks across all the multiple disk surfaces.

There is a time associated with turning heads on and off so that a different surface can be accessed. We call this overhead the **head-switching** time.

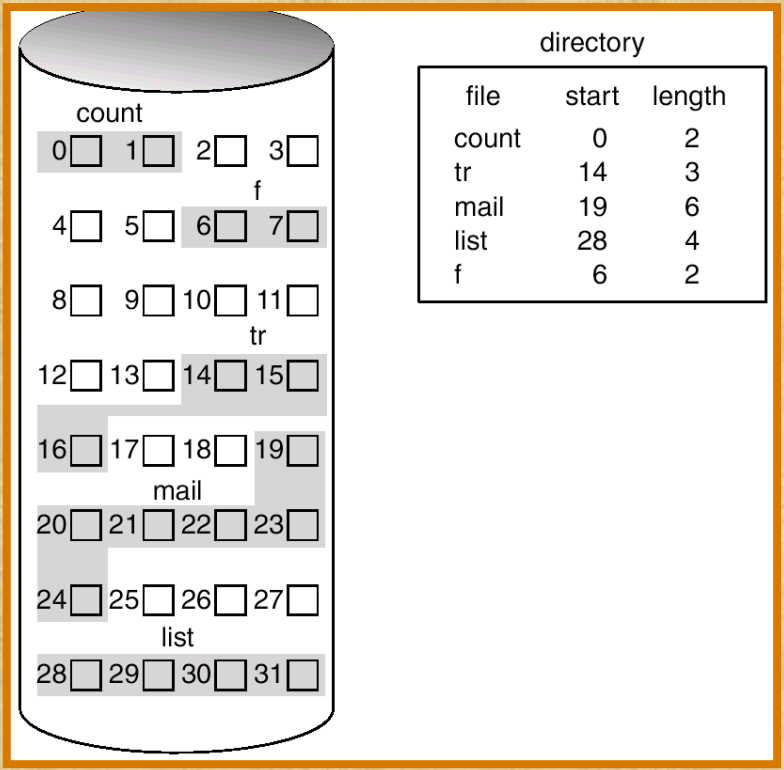
The time to move the arm to read another cylinder is due to the mechanics of the arm. It is certainly much large than the head-switching time, which is due to electronics only.

Question: How should one organize data across multiple surfaces to minimize access overhead?

Contiguous Allocation

- Each file occupies a set of **contiguous blocks** on the disk.
- Simple: only starting location (block #) and length (number of blocks) are required.
- Suitable for **sequential** and **random** access.
- Wasteful of space: dynamic storage-allocation problem; external fragmentation.
- Files cannot grow unless more space than necessary is allocated when file is created (clearly this strategy can lead to **internal fragmentation**).

Contiguous Allocation of Disk Space



directory

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

To deal with the dynamic allocation problem (external fragmentation), the system should periodically **compact** the disk.

Compaction may take a long time, during which the system is effectively **down**.

To deal with possibly growing files, one needs to pre-allocate space larger than required at the initial time => this leads to **internal fragmentation**.

Linked Allocation

Each file is a linked list of disk blocks.

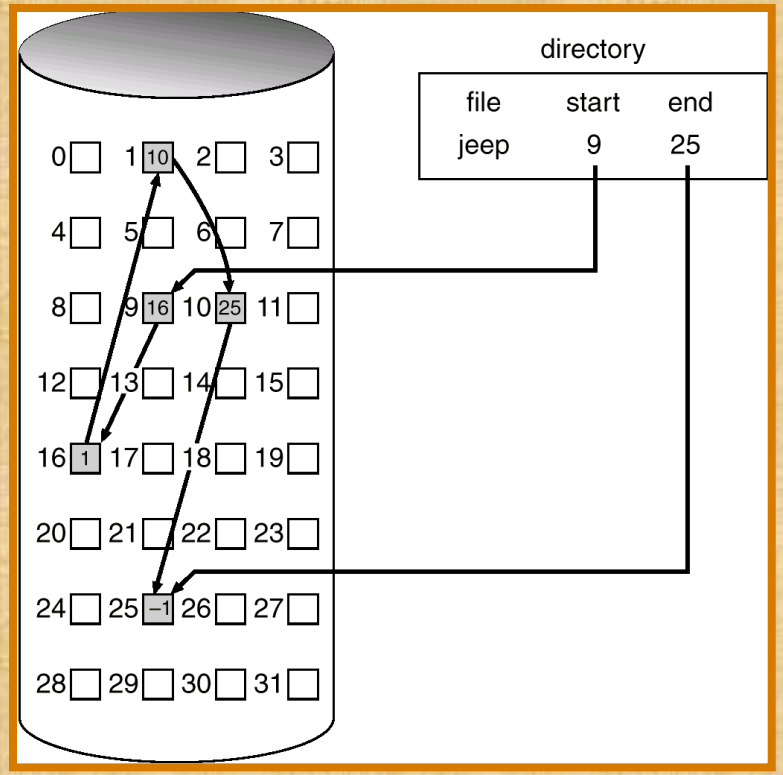
Simple: need only starting address.

Overhead: each block links to the next.

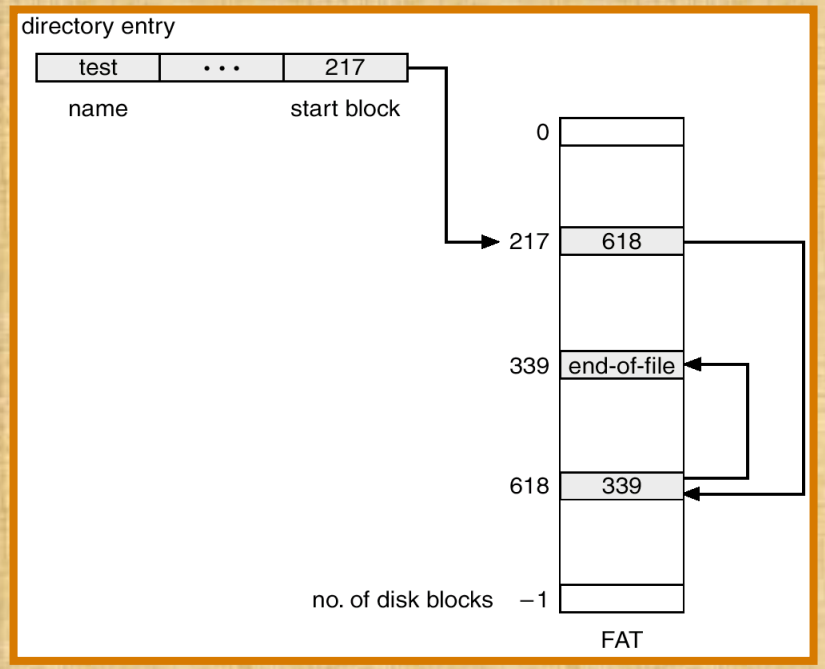
Space cost to store pointer.

Time cost to read one block to find the next.

Internal fragmentation, but not external.
Sequential access comes naturally, random does not.



Example: File-Allocation Table (FAT)



Simple and efficient: One entry for each block; indexed by block number. The table implements the list linking the blocks in a file.

Growing a file is easy: find a free block and link it in.

Random access is easy.

If the FAT is not cached in memory, a considerable number of disk seeks happens.

Used by MS-DOS and OS/2.

Indexed Allocation

Brings all pointers together into an *index block.*

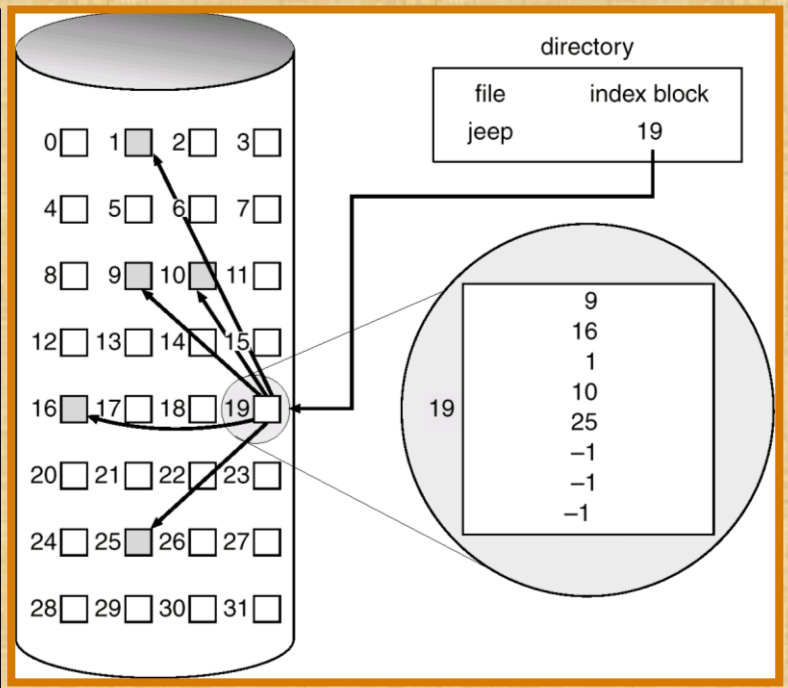
One index block ***per file***.

Random access comes easy.

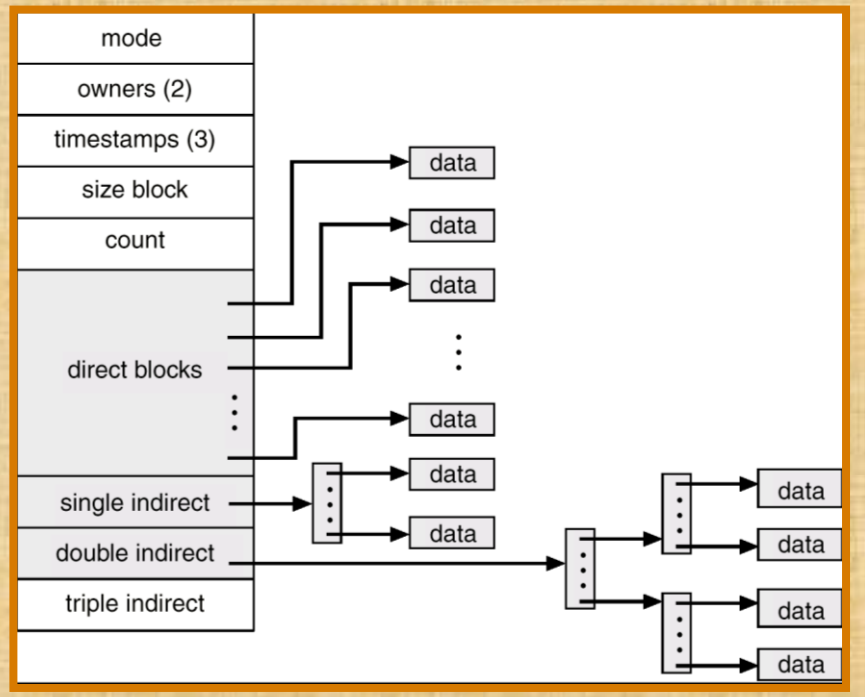
Dynamic access without external fragmentation, but have overhead of index block.

Wasted space: how large should an index block be to minimize the overhead?

- linked index blocks
- multilevel index
- combined scheme



Combined Scheme: UNIX



If file is small enough, use only **direct blocks** pointers.

If number of blocks in file is greater than the number of direct block pointers, use **single**, **double**, or **triple indirect**.

Additional levels of indirection increase the number of blocks that can be associated with a file.

Index blocks can be cached in memory, like FAT. **Access to data blocks, however, may require many disk seeks**.