Chapter 11: File System Implementation

File Systems

- Tmpfs, a temporary file system in memory, fast but not persistent
- Squashfs, a read-only compressed file system supporting fast access
- Ext4, the main Linux journaling file system, reliable
- Ceph, an open source distributed and scalable file system
- FAT, simple and robust file system widely used for compatibility

- Why so many file systems?
- File system is still one of the most active areas of OS research!

Chapter 11: File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance

Objectives

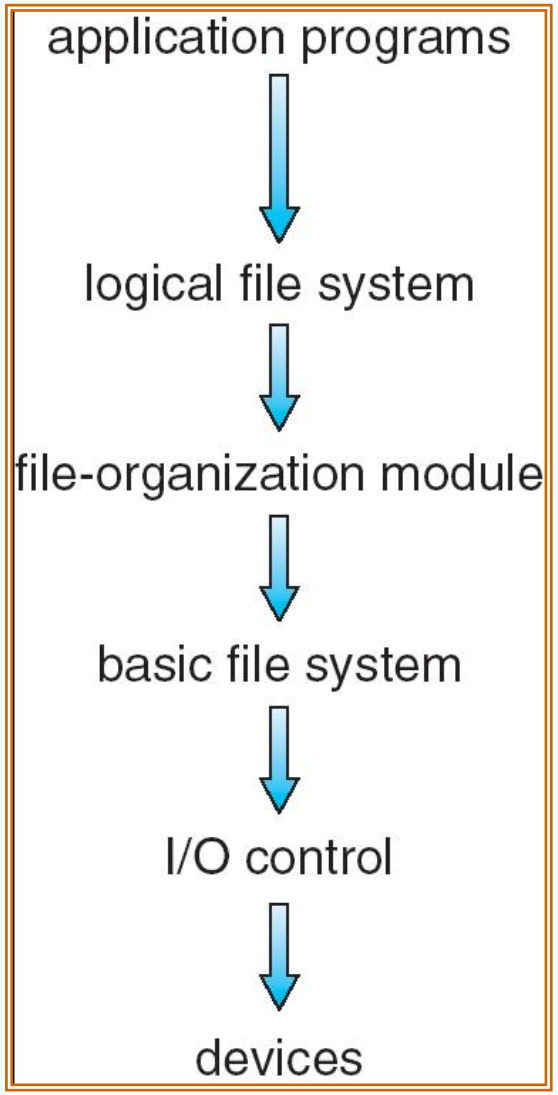
- To describe the details of implementing local file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs

File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
- File system is composed of many layers

Layered File System

Question: Why layers?



Manages metadata of files, Protection and security

Translates logical block addr To physical addr. Free space mgmt

aka. block I/O subsystem

Commands to r/w physical blocks, I/O schedule, buffer

Device drivers/ interrupt handlers

Translates ‘r/w block x’ to low-level hw instructions

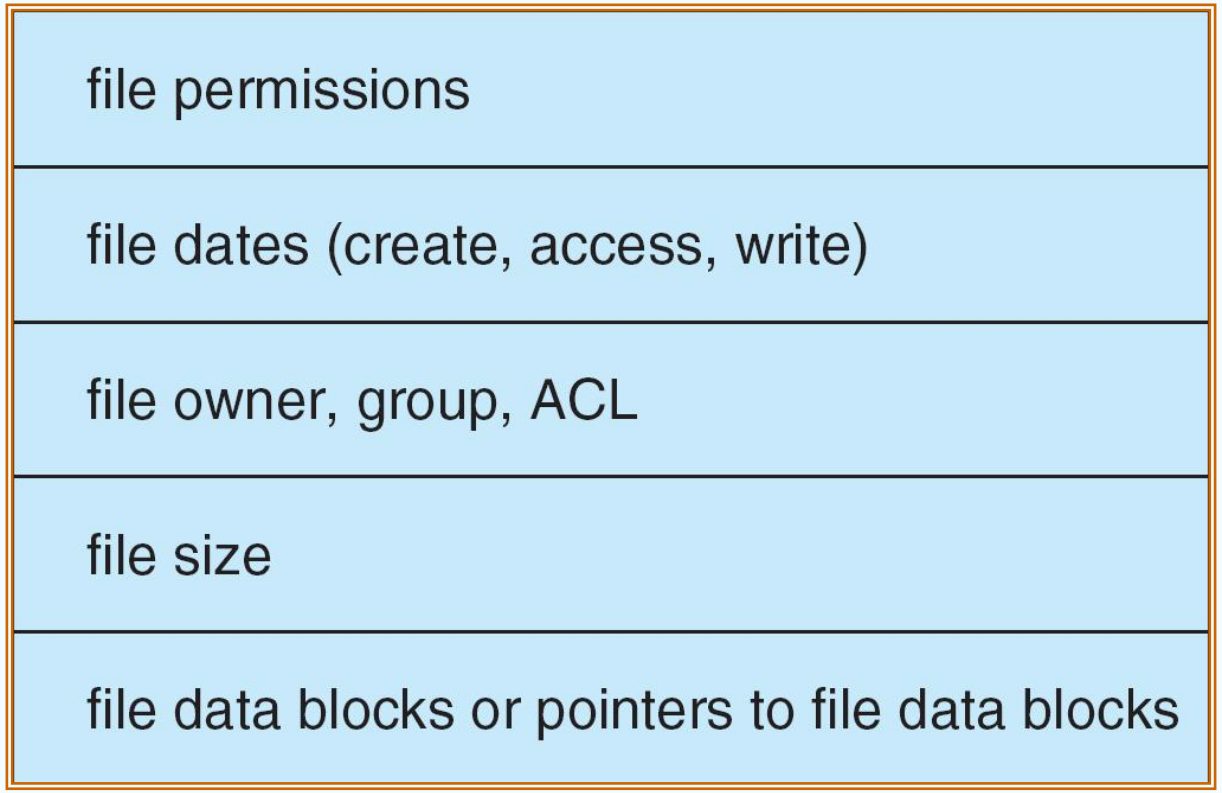
Data Structures Used to Implement FS

- Disk structures
  - Boot control block (per volume)
  - Volume control block per volume (superblock in Unix)
  - Directory structure per file system
  - Per-file FCB (inode in Unix)
- In-memory structures (see fig)
  - *In-memory mount table* about each mounted volume
  - *Directory cache* for recently accessed directories
  - System-wide open-file table
  - Per-process open-file table

Question: Why directory cache works?

A Typical File Control Block

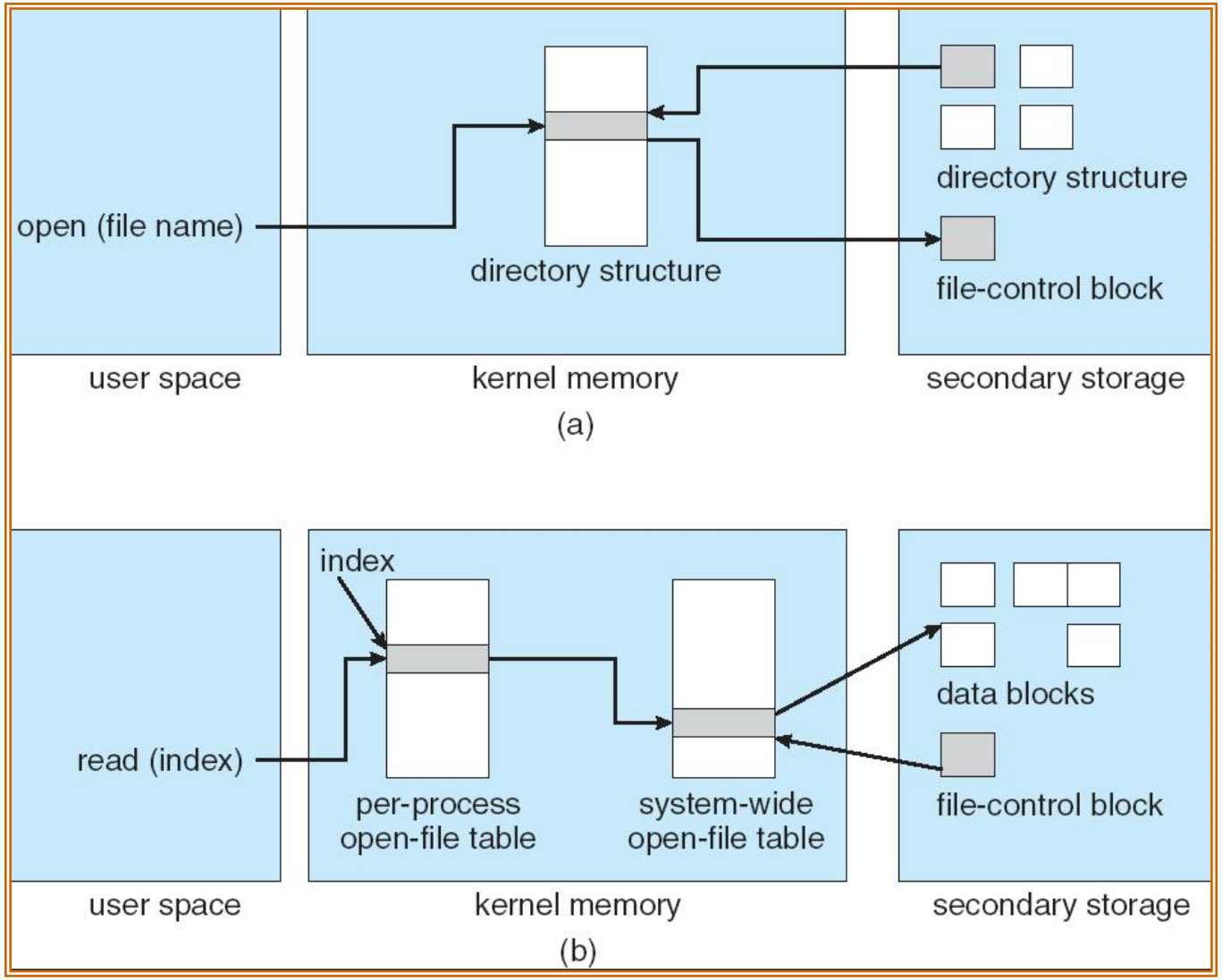
File control block – storage structure consisting of information about a file



In-Memory File System Structures

- The following figure illustrates the necessary file system structures provided by the operating systems.
- Figure 12-3(a) refers to opening a file.
- Figure 12-3(b) refers to reading a file.

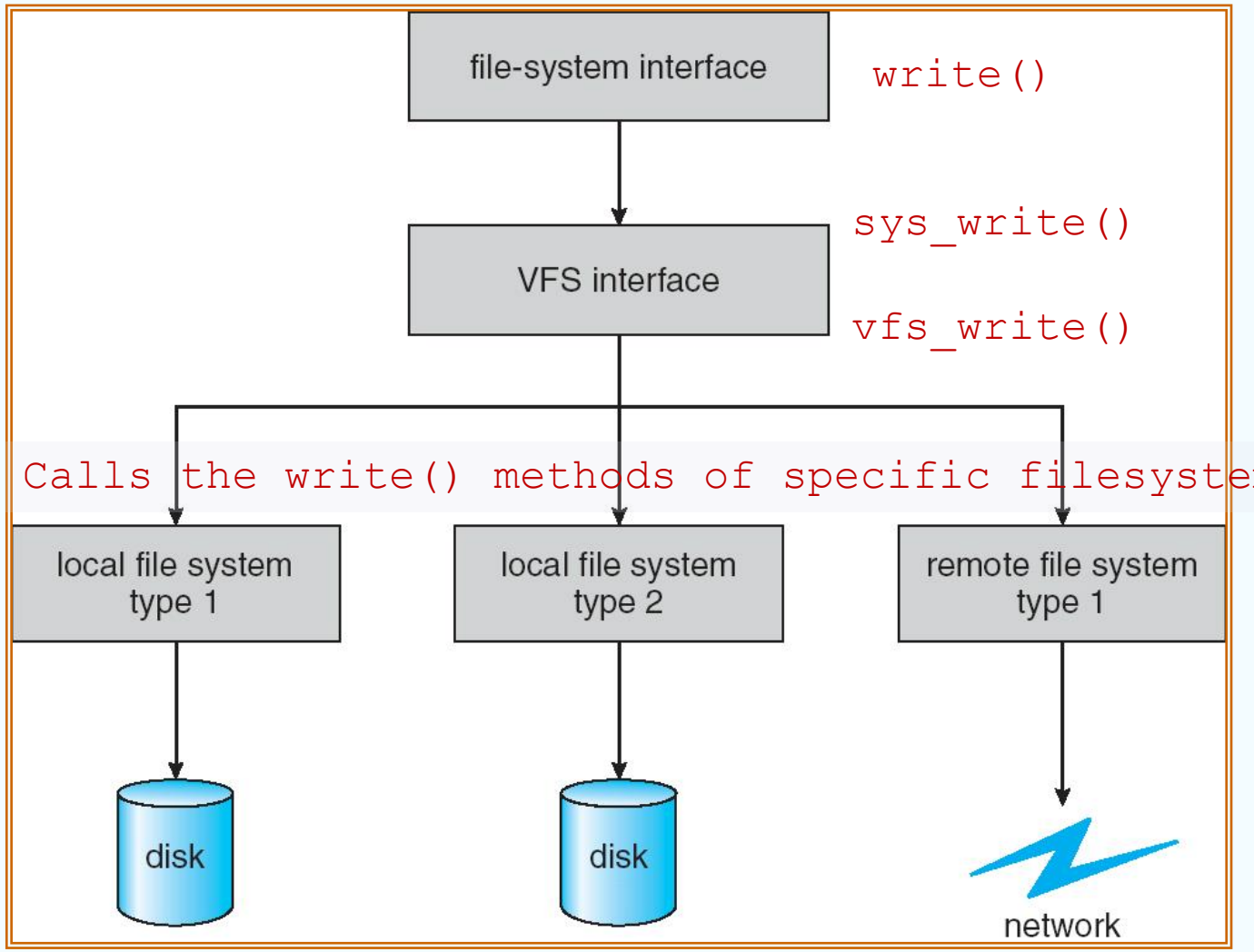
In-Memory File System Structures



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems. VFS is NOT a disk file system!
- 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.
- Defines a network-wide unique structure called vnode.

Schematic View of Virtual File System

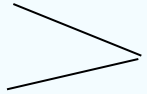


In-Memory VFS Objects

- The four primary object types of VFS:
  - superblock object: a specific mounted filesystem，对应(但不是)磁盘文件系统的文件系统超级块或控制块。
  - inode object: a specific file, 对应(但不是)磁盘文件系统的文件控制块
  - dentry object: an individual directory entry
  - file object: an *open file* as associated with a process, 只要文件一直打开，这个对象就一直存在于内存。

Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute



tradeoff

- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - **collisions** – situations where two file names hash to the same location
  - fixed size – can use chained-overflow hash table
  - Or rehashing to another larger hash table

Allocation Methods

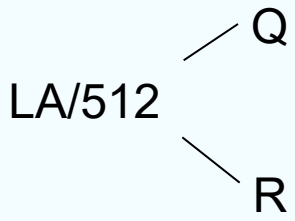
- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

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
- Random access supported
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow

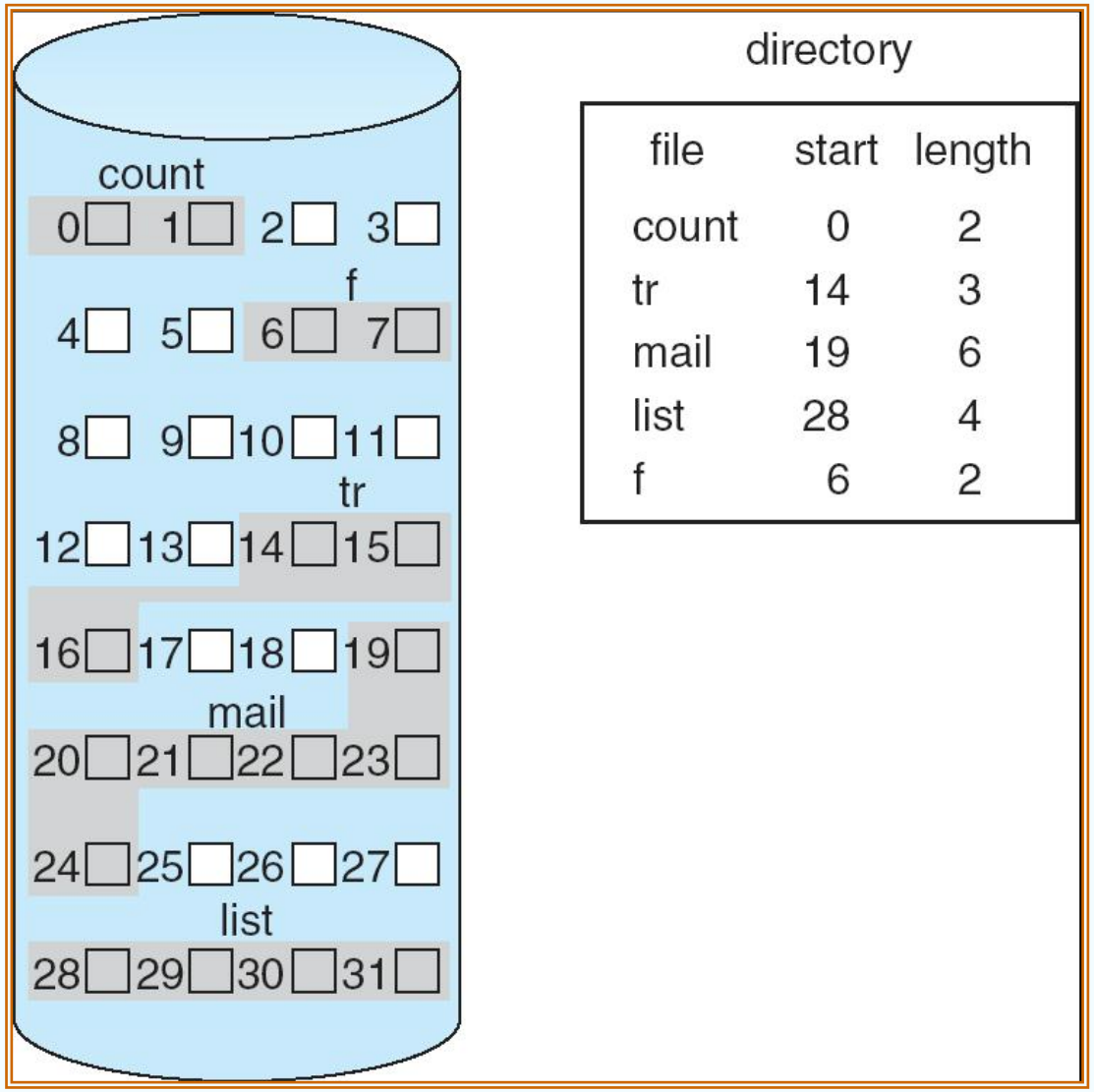
Contiguous Allocation

- Mapping from logical to physical



Block to be accessed = Q + start_address
Displacement into block = R

Contiguous Allocation of Disk Space

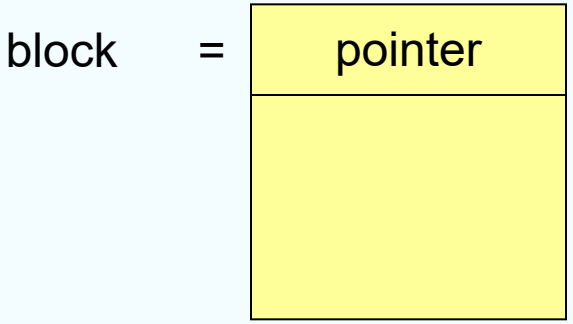


Extent-Based Systems

- Many newer file systems use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in **extents**
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents.

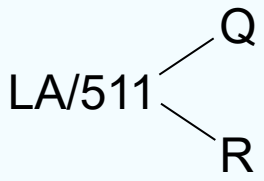
Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.



Linked Allocation (Cont.)

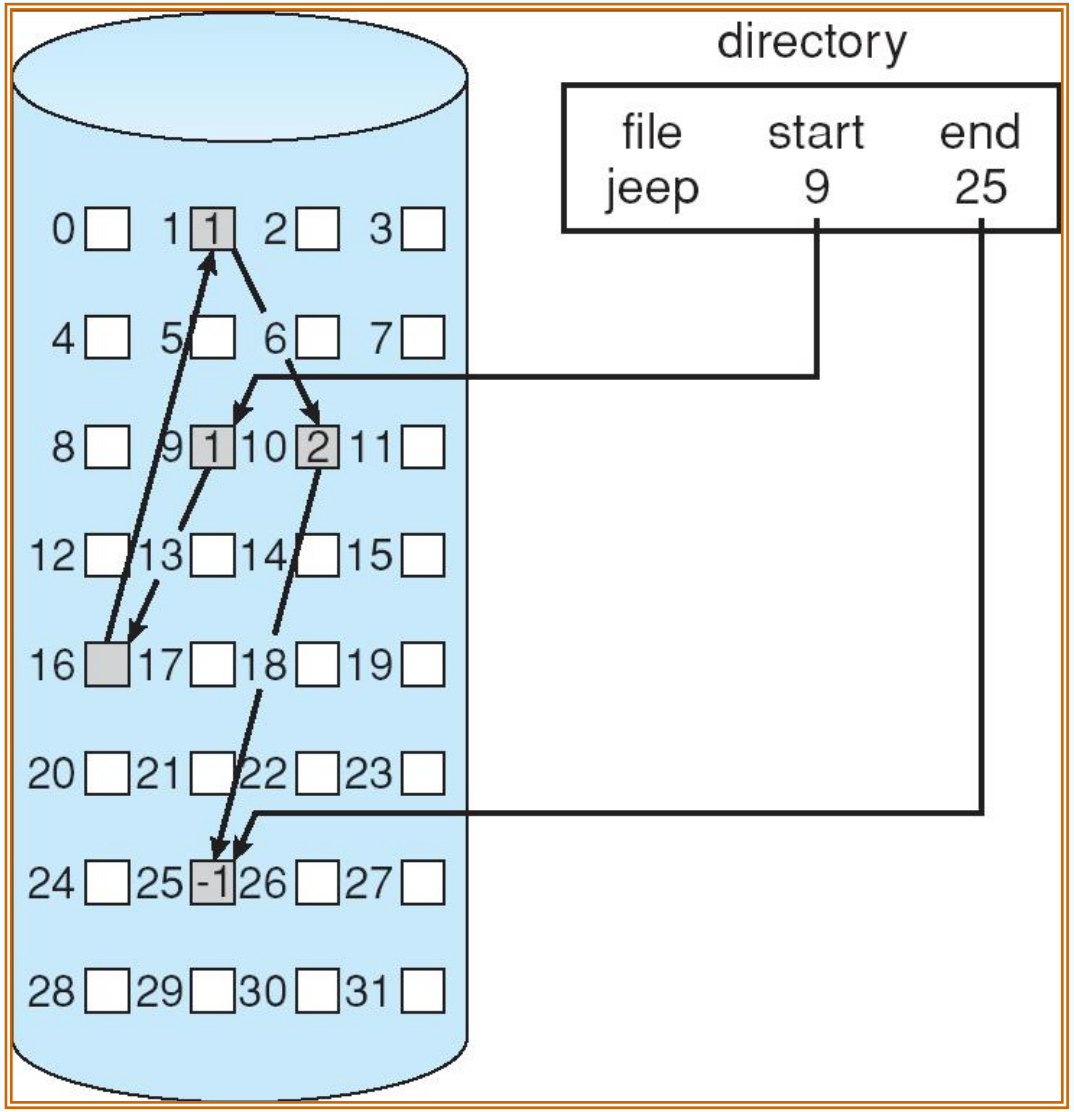
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access, poor reliability
- Mapping



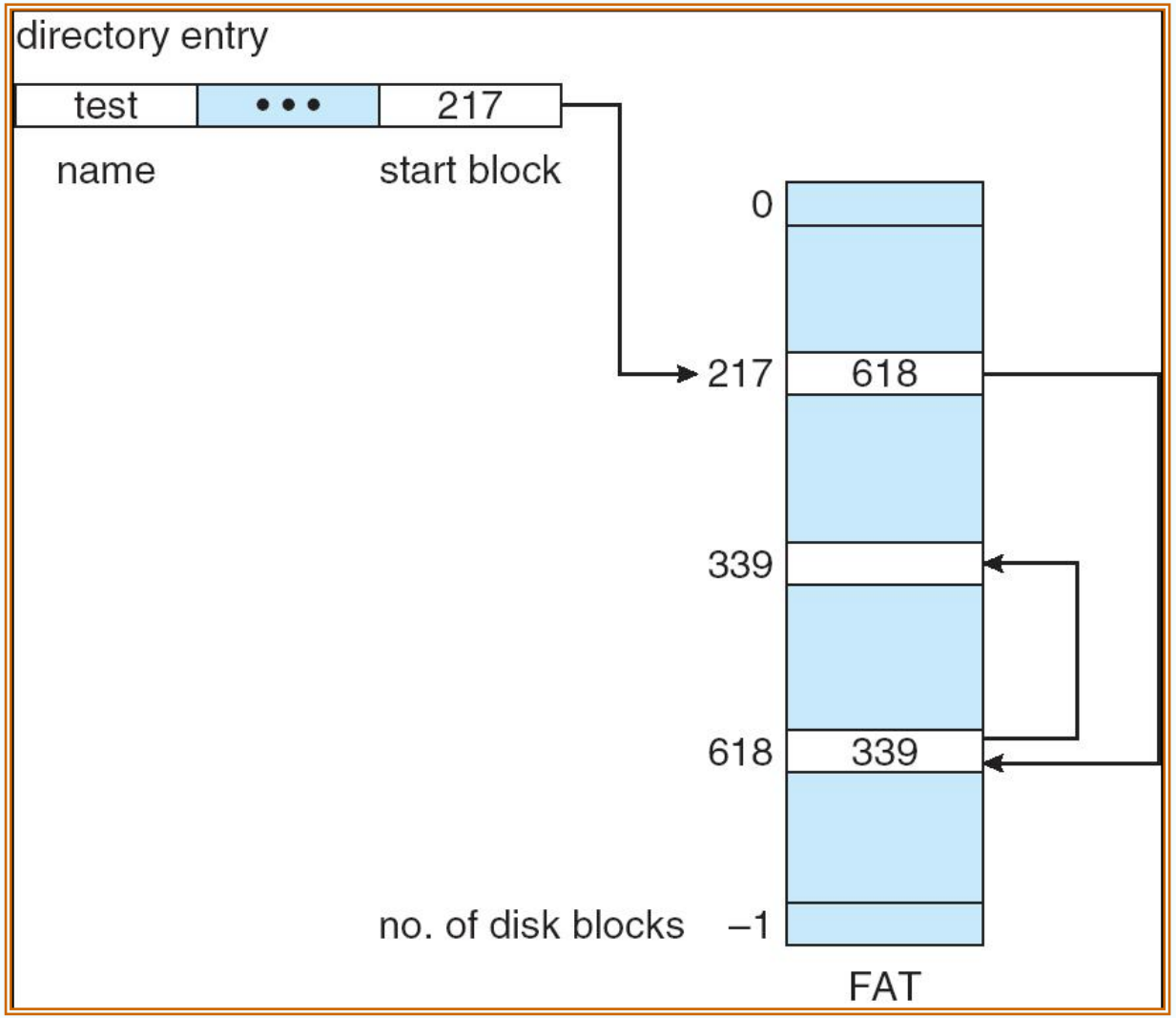
Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.

Linked Allocation

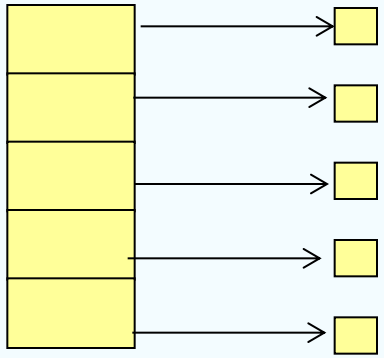


File-Allocation Table



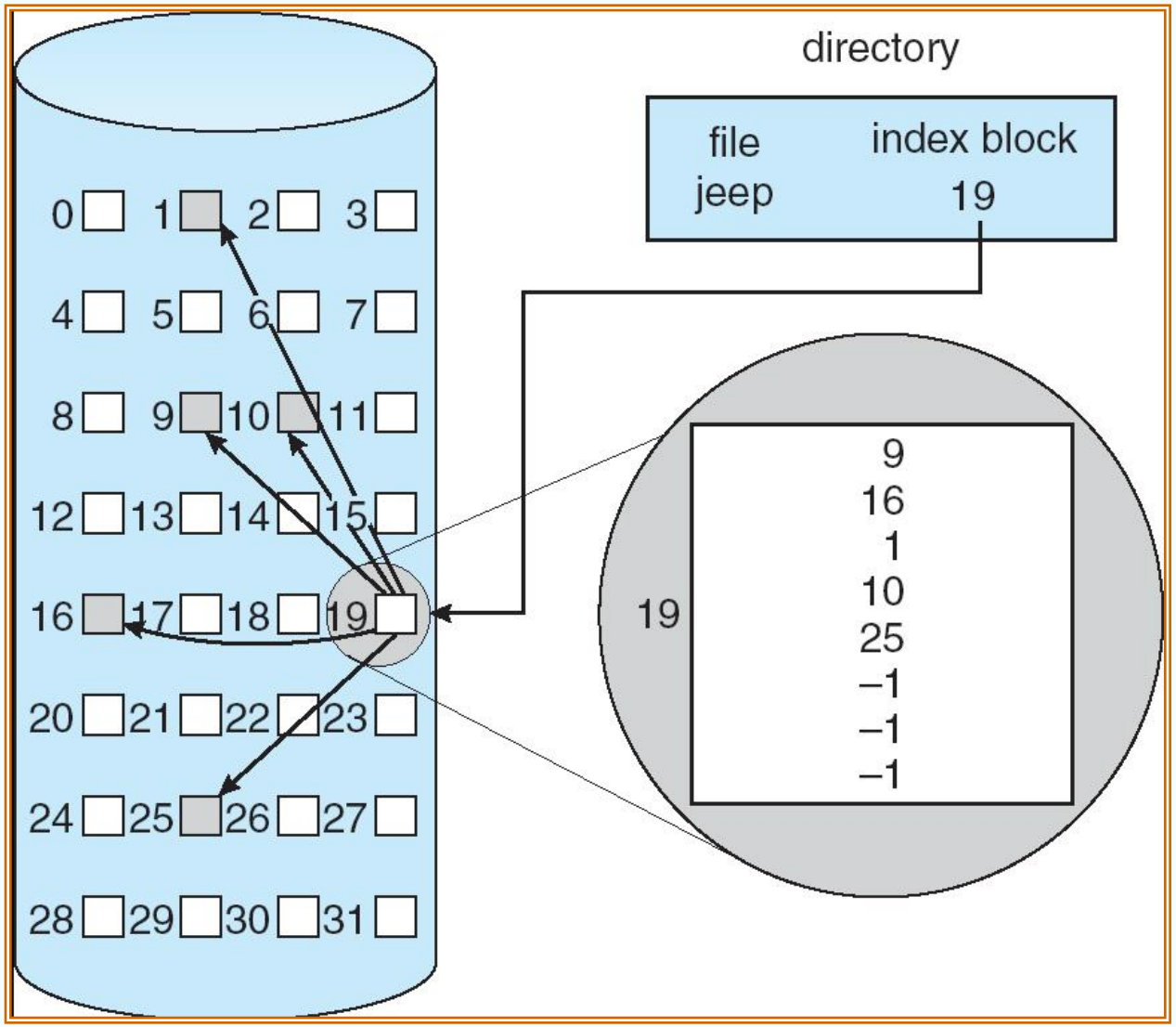
Indexed Allocation

- Brings all pointers together into the *index block.*
- Logical view.



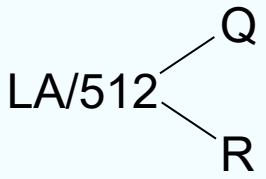
index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

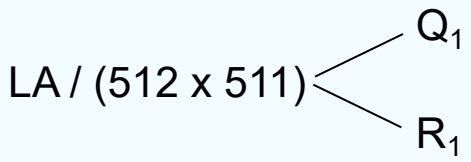
- Need index table (analogous to page table)
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- When mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



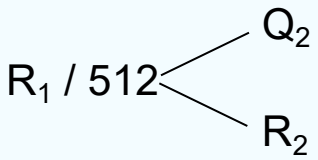
Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

- When mapping from logical to physical in a file of unbounded length (block size of 512 words). – more pointers are needed
- **Linked scheme** – Link blocks of index table (no limit on size).



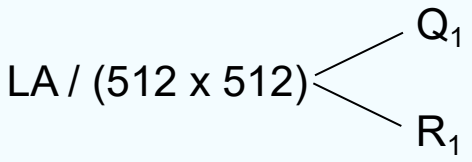
Q_1 = block of index table
R_1 is used as follows:



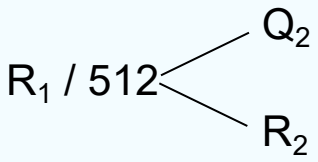
Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

- **Two-level index** (maximum file size is 512^3)

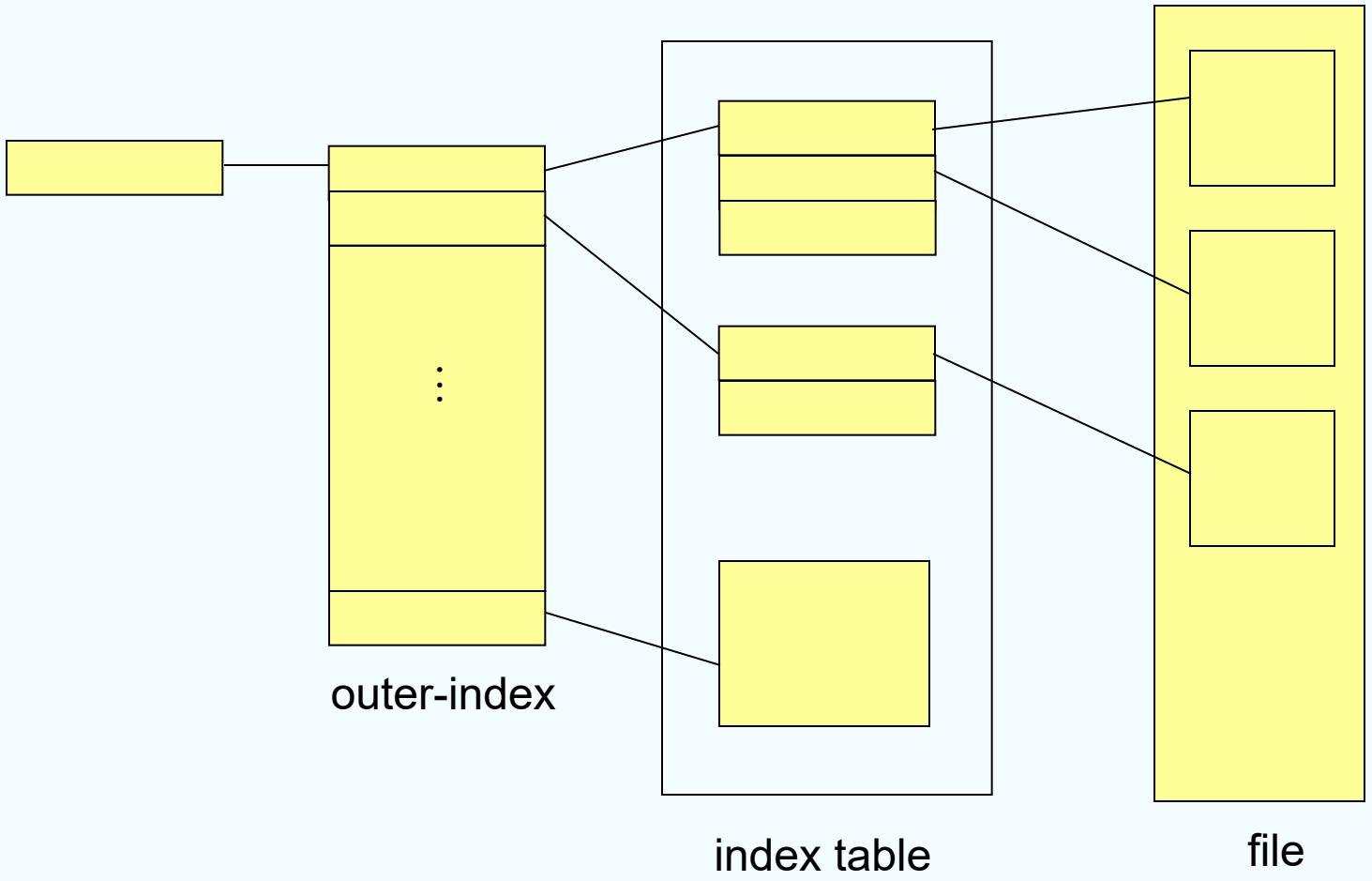


Q_1 = displacement into outer-index
R_1 is used as follows:

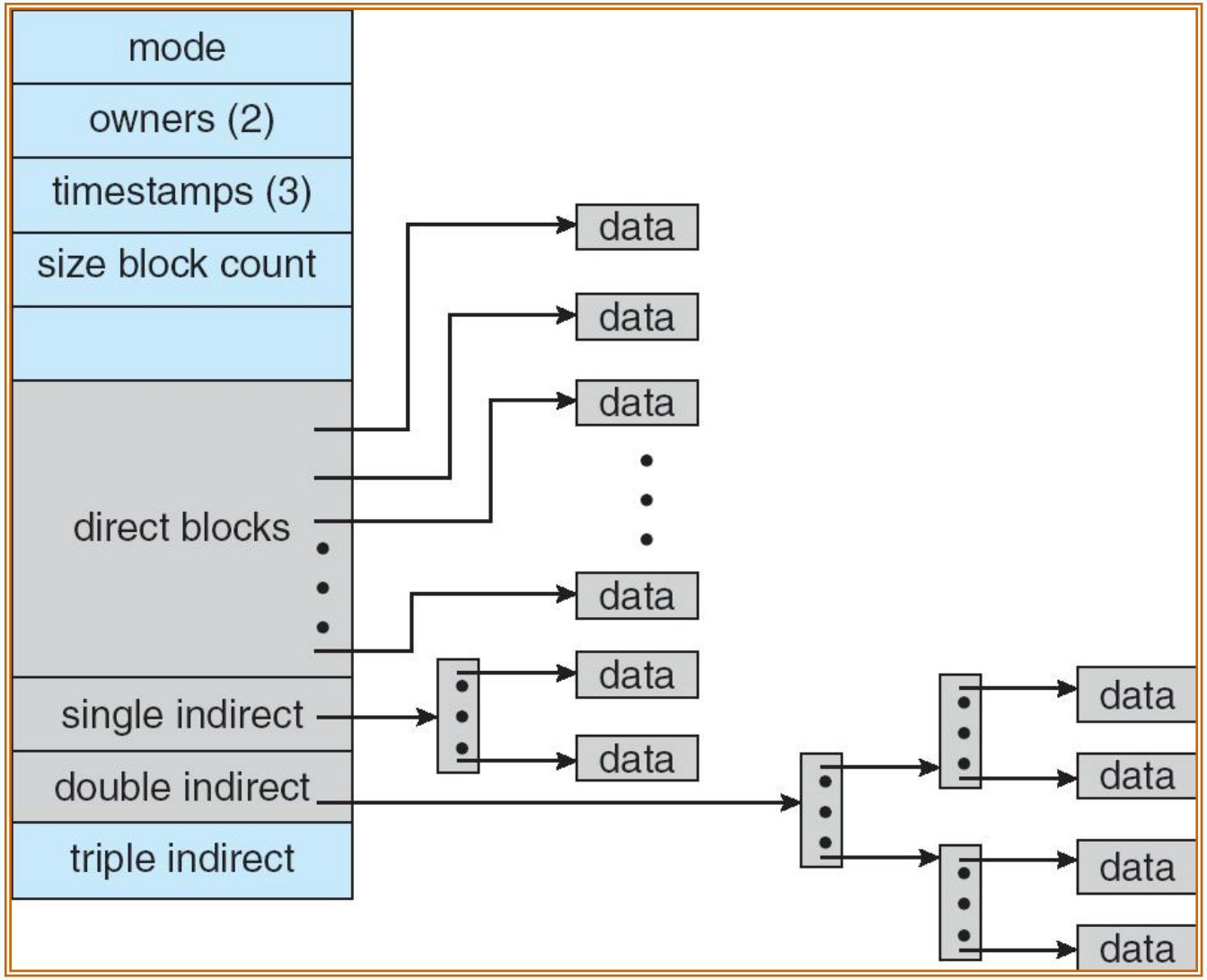


Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

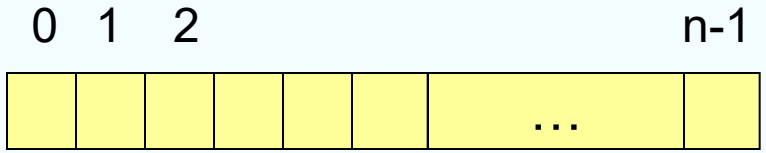


Combined Scheme: UNIX (4K bytes per block)



Free-Space Management

- Bit vector (*n* blocks)



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

Block number calculation (finding the first free block)

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

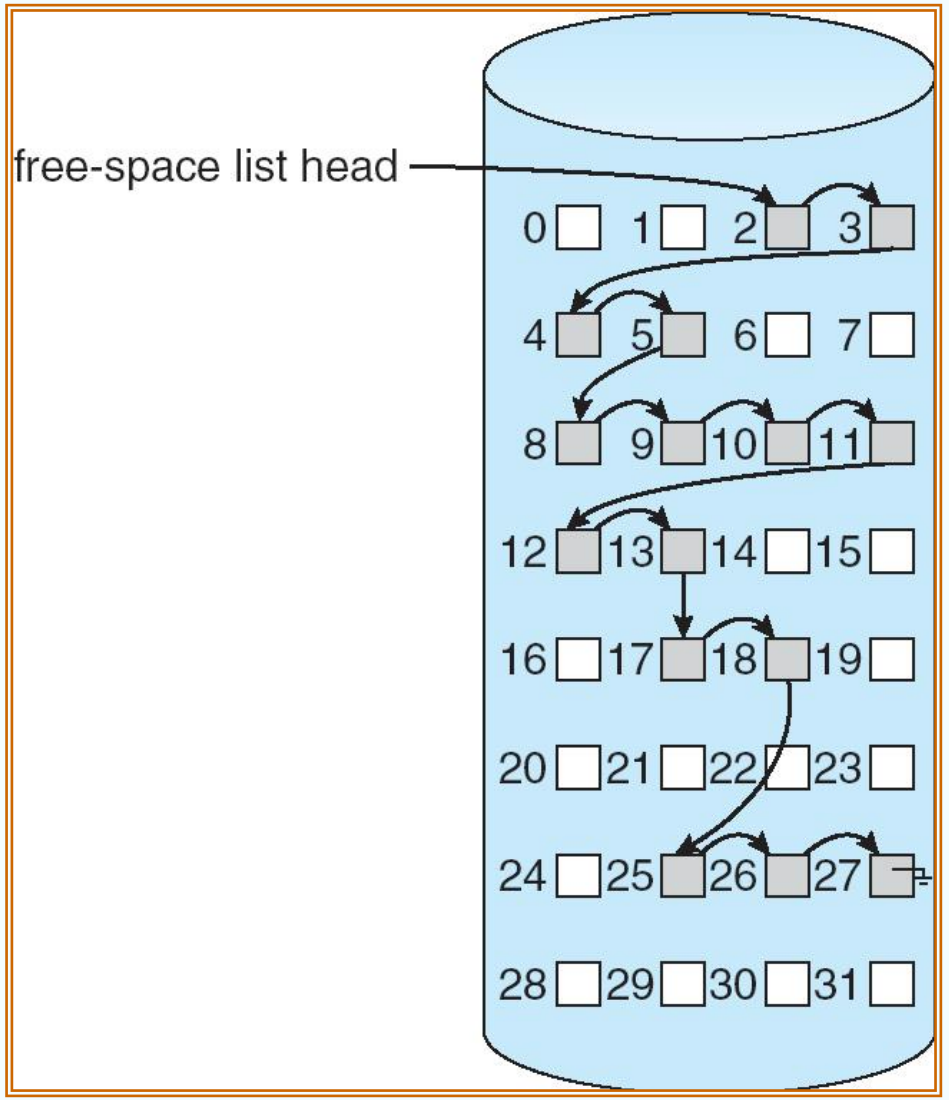
    block size = 2^{12} bytes

    disk size = 2^{30} bytes (1 gigabyte)

    $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files

- Linked list (free list) – see figure
  - Cannot get contiguous space easily
  - But basically can work (FAT)
  - No waste of space
- Grouping – a modification of the Linked List
  - Addresses of the n free blocks are stored in the first block.
  - The first n-1 blocks are actually free. The last block contains addresses of another n free blocks
- Counting
  - Address of the first free block and number n contiguous blocks

Linked Free Space List on Disk



Free-Space Management (Cont.)

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - The copy in memory and disk may differ
    - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk
  - Solution:
    - Set bit[*i*] = 1 in disk
    - deallocate block[*i*]
    - Set bit[*i*] = 1 in memory

Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry (for example "last write date" is recorded in directory)

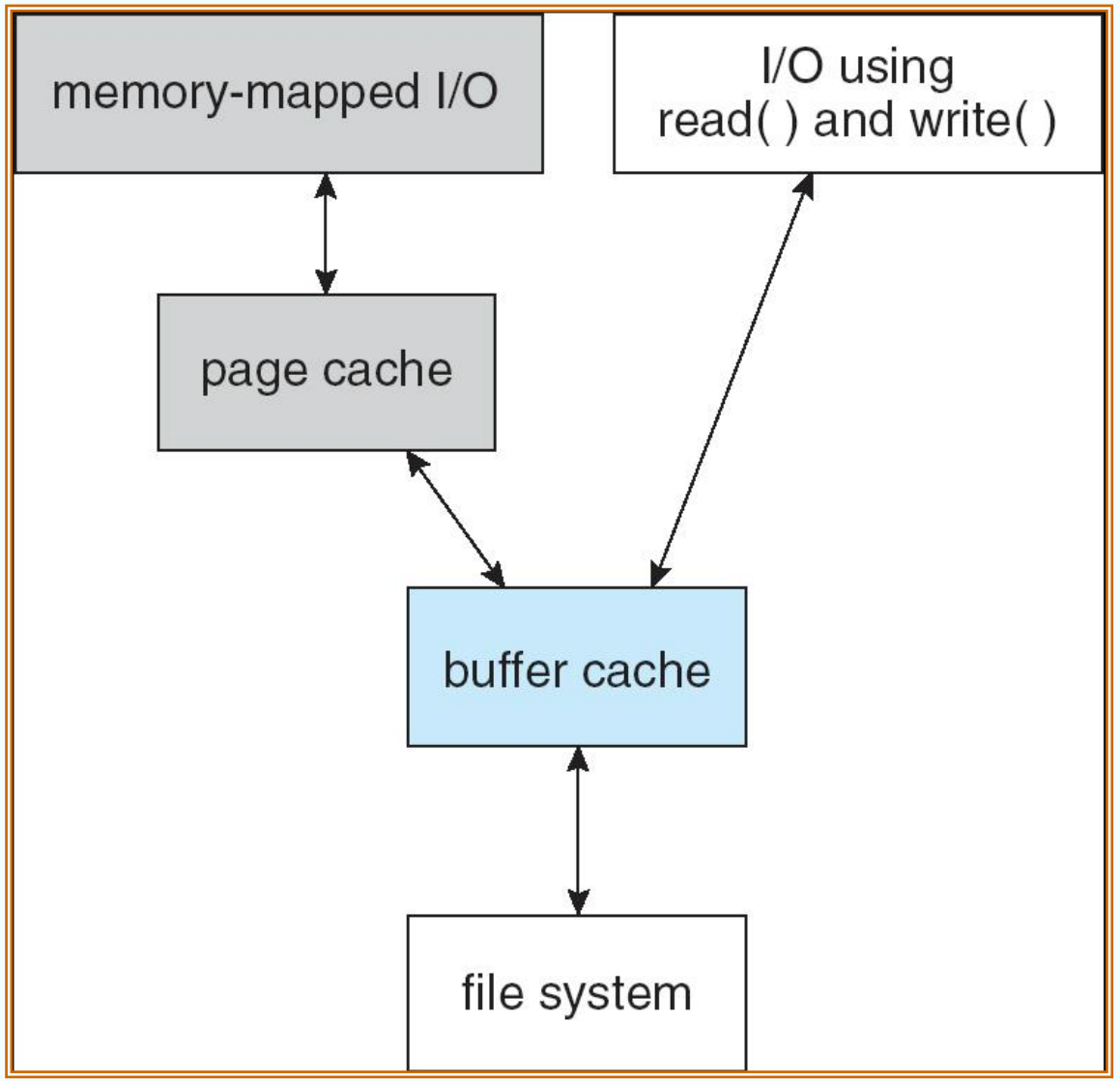
Generally, every data item has to be considered for its effect.

- Performance
  - **disk cache** – separate section of main memory for frequently used blocks
  - **free-behind and read-ahead** – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or **RAM disk**

Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

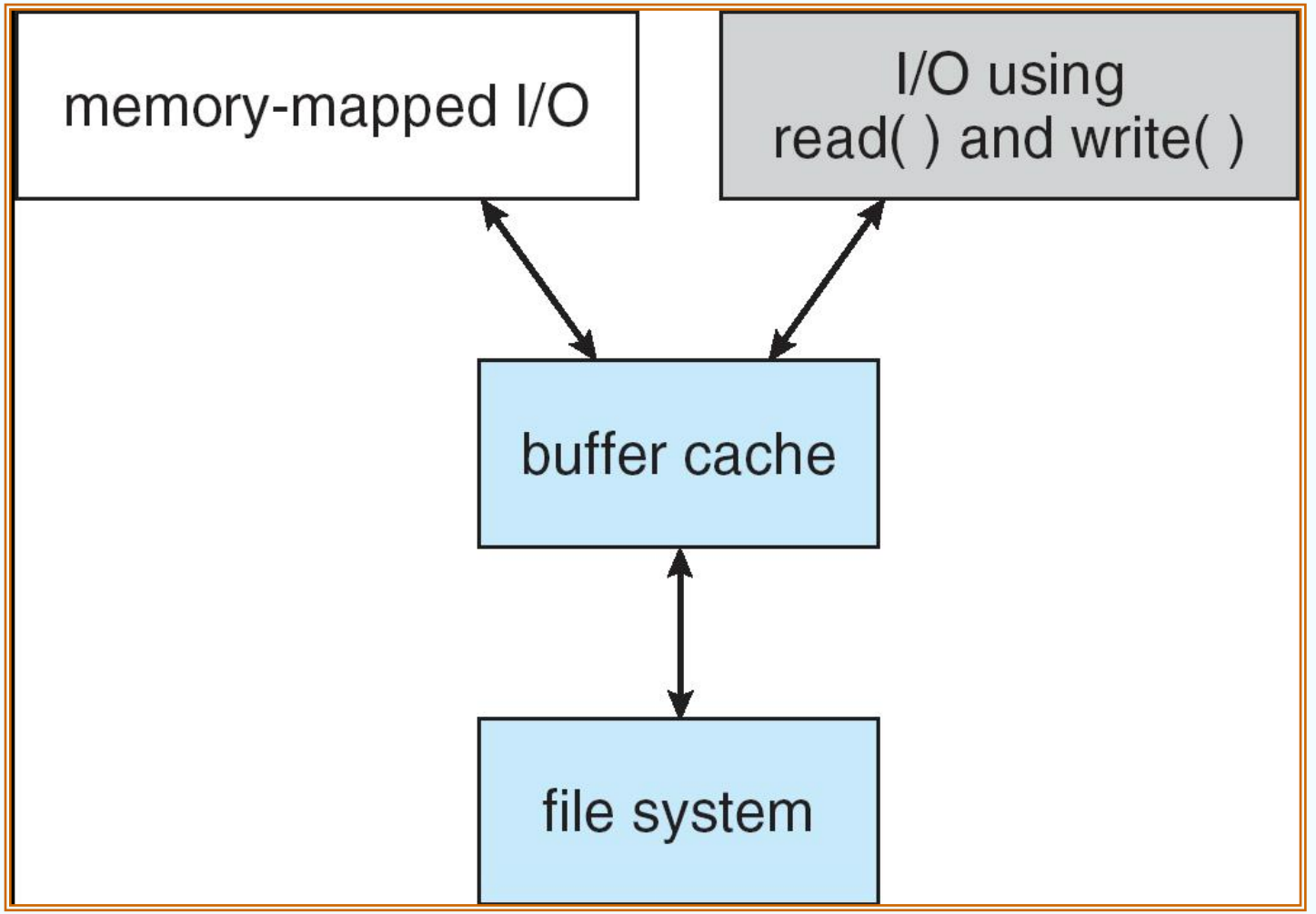
I/O Without a Unified Buffer Cache



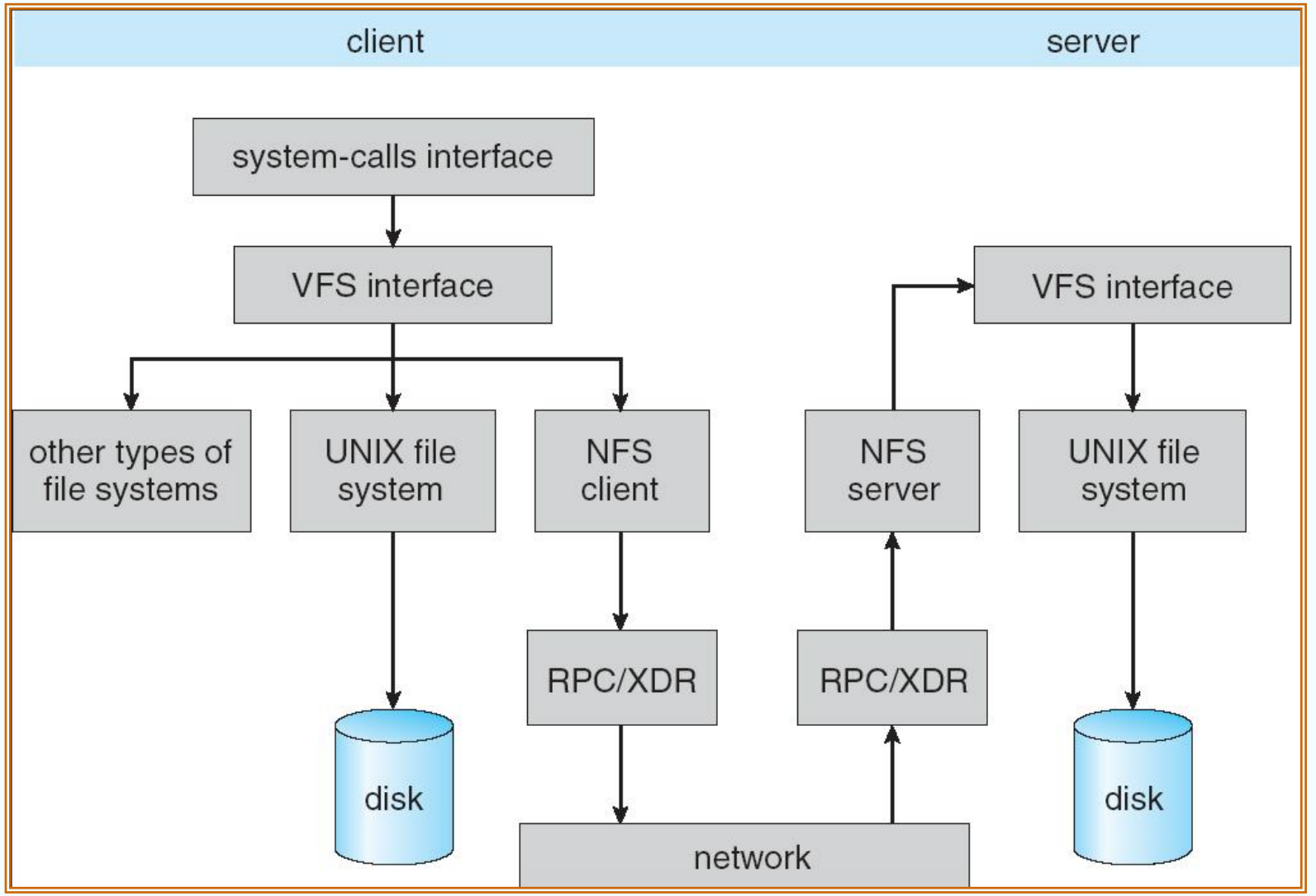
Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O
- Avoids double caching

I/O Using a Unified Buffer Cache



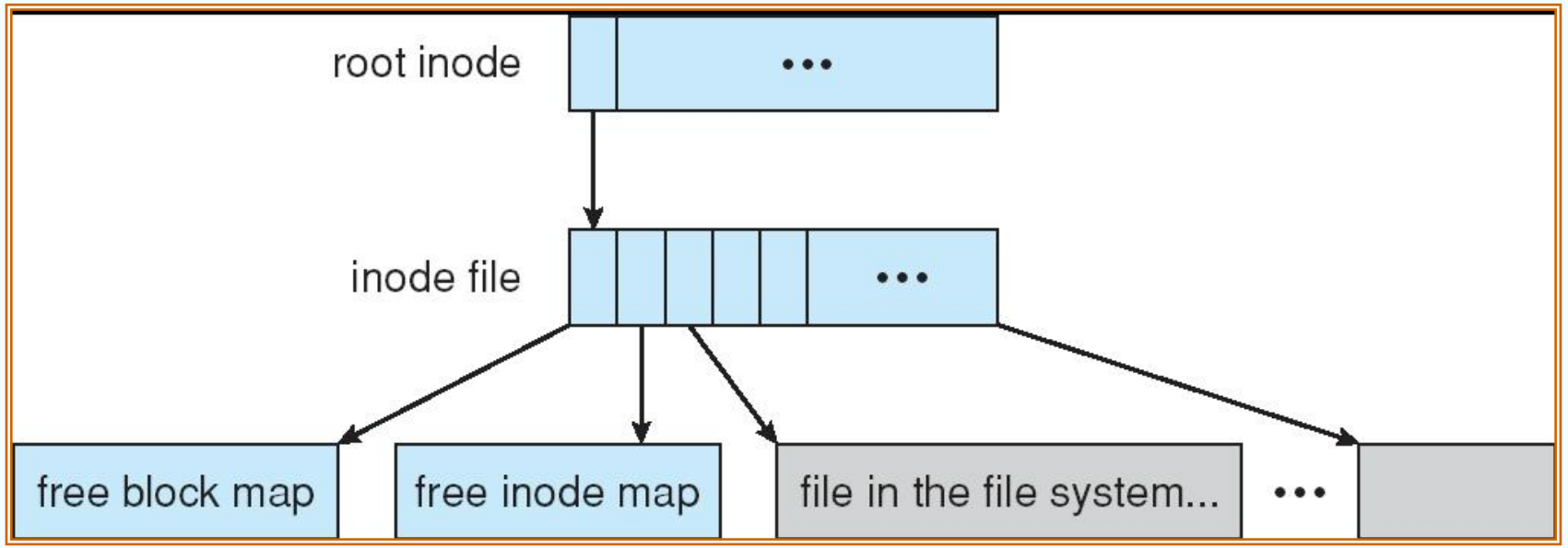
Schematic View of NFS Architecture



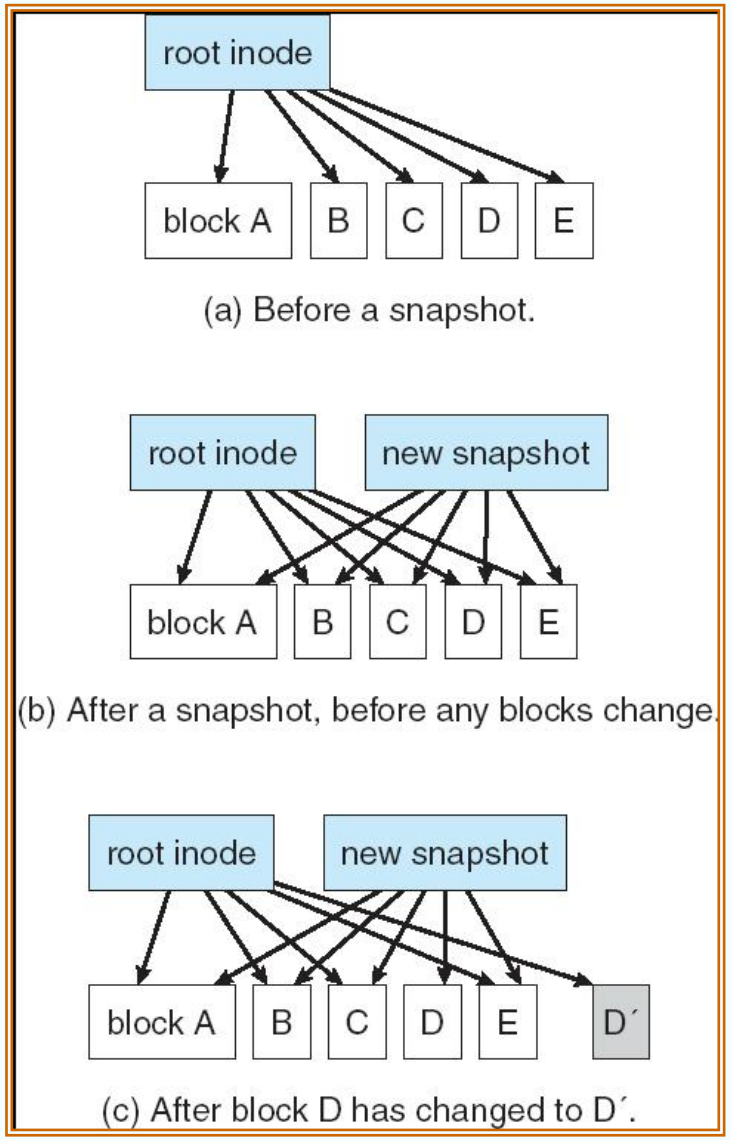
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM (flash memory) for write caching
- Similar to Berkeley Fast File System, with extensive modifications

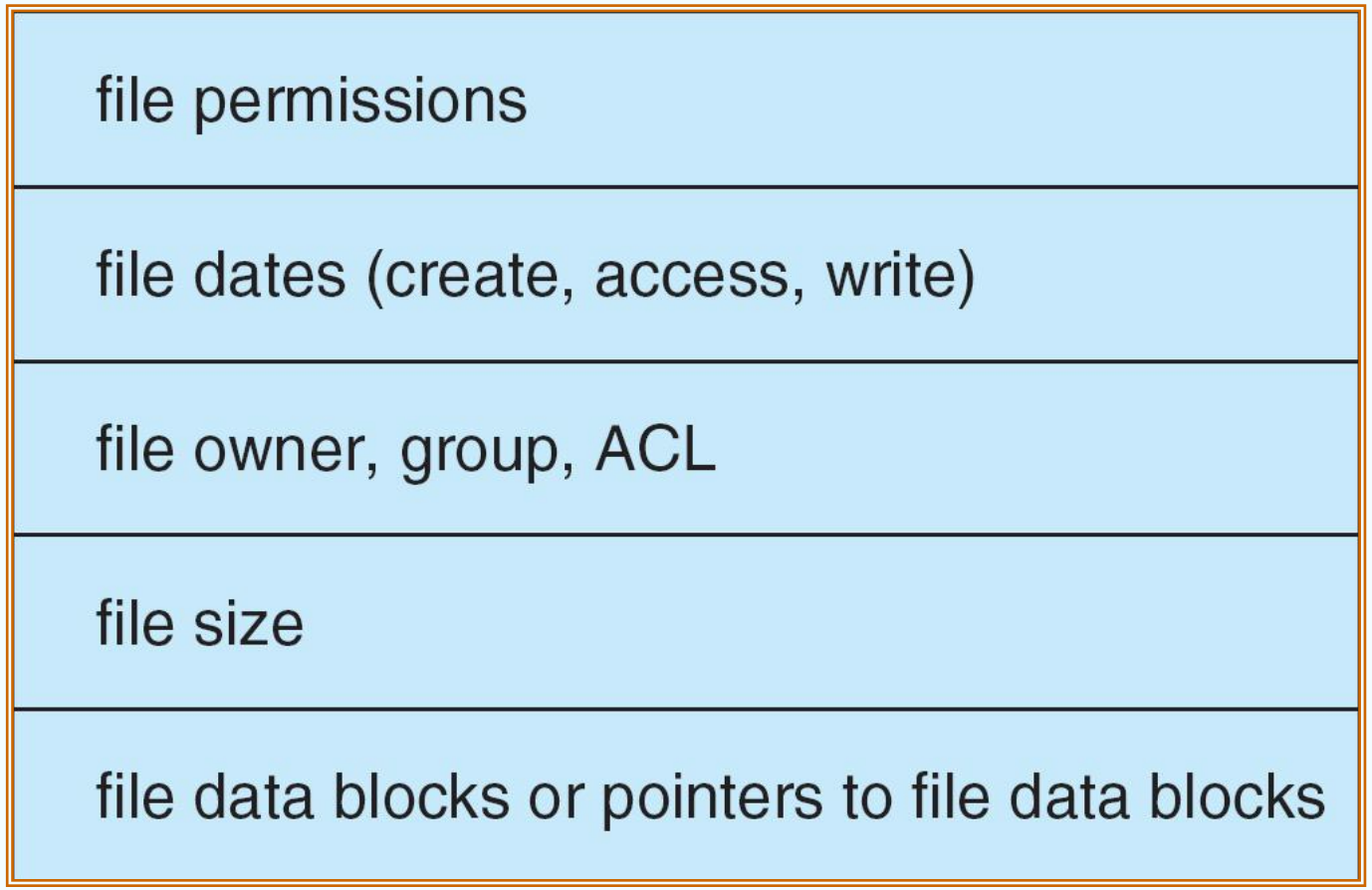
The WAFL File Layout



Snapshots in WAFL



11.02


file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

End of Chapter 11

Write/Read Amplification

- *Write amplification* is the ratio of the amount of data written to the storage device versus the amount of data written to the database.
- If you are writing 10 MB to the file, and you observe 30 MB disk write, your write amplification is 3.
- Write amplification is bad for *flash-based storage* lifetime.