Chapter 11: File System Implementation

Chapter 11: File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS
- Example: WAFL File System

Objectives

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

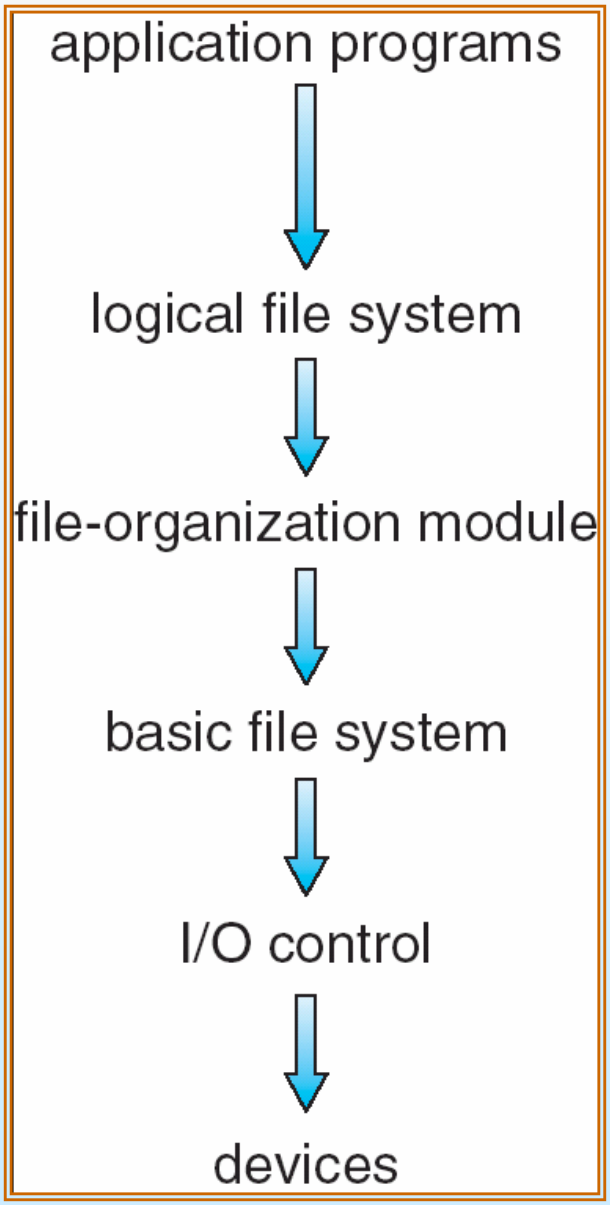
File Systems

- Most operating systems support more than one FS.
  - For example, most CD-ROMs are written in the **ISO 9660 format**, a standard format agreed on by CD-ROM manufacturers.
- In addition to removable-media file systems, each operating system has one disk-based file system (or more). UNIX uses the **UNIX file system** (UFS), which is based on the **Berkeley Fast File System** (FFS). Windows NT, 2000, and XP support disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM, DVD, and Floppy-disk file-system formats.
- Although Linux supports over forty different file systems, the standard Linux file system is known as the **extended file system**, with the most common version being **ext2** and **ext3**.
- There are also **distributed file systems** in which a file system on a server is mounted by one or more clients.

File-System Structure

- File system, design problems:
  - How the file system should look to the user
  - Create algorithms and data structures to map the logical FS to the physical FS.
- File system resides on secondary storage (disks)
- File system organized into layers
- **File control block** – storage structure consisting of information about a file

Layered File System



Device Drivers

- The lowest *level,* the *I/O control,* consists of **device drivers** and **interrupt handlers** to transfer information between the main memory and the disk system.

- A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123." Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system.

- The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.

Basic file system

- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector 10).

File Organization Module

- The **file-organization module** knows about files and their logical blocks, as well as physical blocks.
- By knowing the type of file allocation used and the location of the file, the file organization module can **translate** logical block addresses to physical block addresses for the basic file system to transfer.
- Each file's logical blocks are numbered from 0 (or 1) through N. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.
- The file-organization module also includes the **free-space manager**, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical File System

- Finally, the logical file system manages **metadata information**. Metadata includes all of the file-system structure except the actual *data* (or contents of the files).

- The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name.

- It maintains file structure via **file-control blocks**. A file-control block (FCB) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection and security.

A Typical File Control Block

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

FS Implementation

- Several **on-disk** and **in-memory** structures are used to implement a file system.
- These structures vary depending on the operating system and the file system but some general principles apply.
- A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the **boot block**; in NTFS, it is called the **partition boot sector.**
- A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, free-block count and free-block pointers, and free FCB count and FCB pointers. In UFS, this is called a **superblock**; in NTFS, it is stored in the **master file table**.

FS Implementation

- **A directory structure** per file system is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS it is stored in the **master file table**.
- A **per-file FCB** contains many details about the file, including file permissions, ownership, size, and location of the data blocks. In UFS, this is called the **inode**.
- In NTFS, this information is actually stored within the **master file table**, which uses a relational database structure, with a row per file.

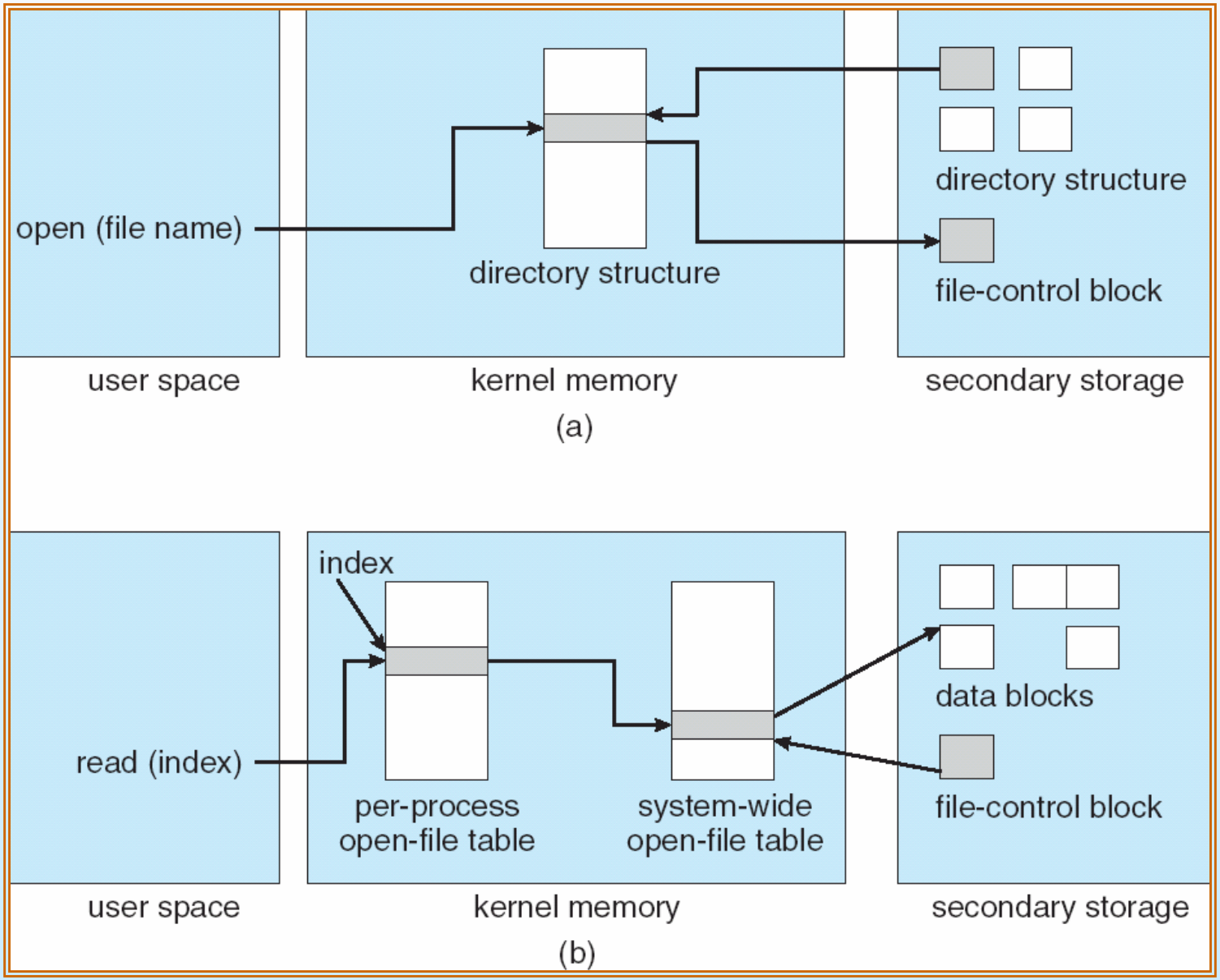
In-memory structures

- An **in-memory mount table** contains information about each mounted volume.
- An **in-memory directory-structure** cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

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



Partitions and mounting

- Raw Disk or Cooked Disk
- Boot information can be stored in a separate partition.
  - Again, it has its own format, because at boot time the system does not have file-system device drivers loaded and therefore cannot interpret the file-system format.
  - Boot information is usually a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte. This boot image can contain more than the instructions for how to boot a specific operating system.
- PCs and other systems can be **dual-booted**.
  - **Multiple operating systems** can be installed on such a system. How does the system know which one to boot?
  - A **boot loader** that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the disk.
  - The disk can have multiple partitions, each containing a different type of file system and a different operating system.

Partitions and mounting

- The root partition, which contains the operating-system kernel and sometimes other system files, is **mounted at boot time**.
- Other volumes can be **automatically** mounted at boot or **manually** mounted later, depending on the operating system.
- As part of a successful mount operation, the operating system verifies that the device contains a **valid file system**. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.
- If the format is invalid, the partition must have its **consistency checked** and possibly corrected, either with or without user intervention.

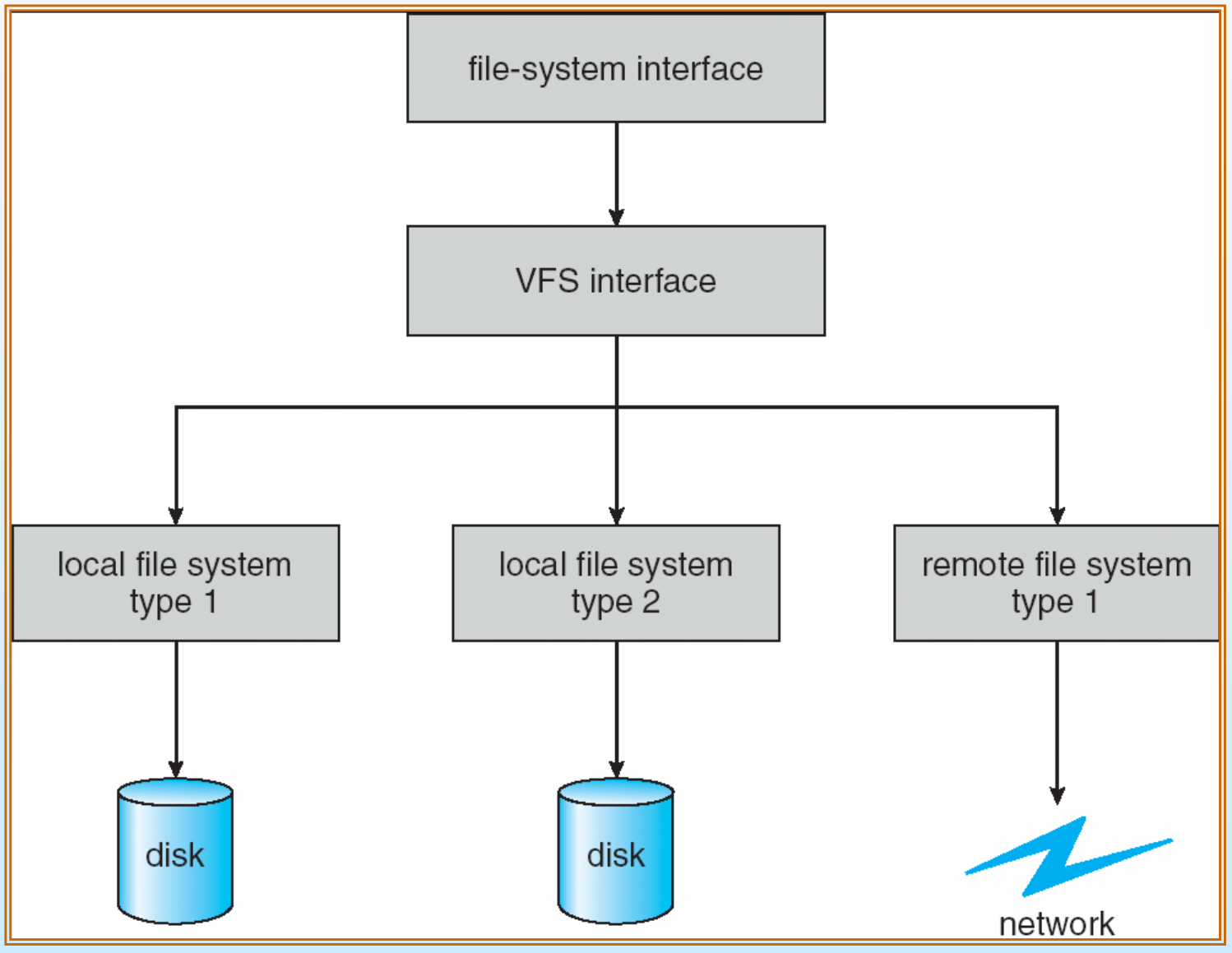
Mount Table

- Finally, the operating system notes in its **in-memory mount table** structure that a file system is mounted, along with the type of the file system.
- The details of this function depend on the operating system. Microsoft Windows-based systems mount each volume in a **separate name space**, denoted by a letter and a colon.
- To record that a file system is mounted at F:, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:.
- When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory. Later versions of Windows can mount a file system at any point within *the* existing directory structure.

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



Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
- **Hash Table** – linear list with hash data structure.
  - The hash takes the name of the file and returns a pointer to the file
  - **decreases** directory search time
  - **collisions** – situations where two file names hash to the same location
  - fixed size

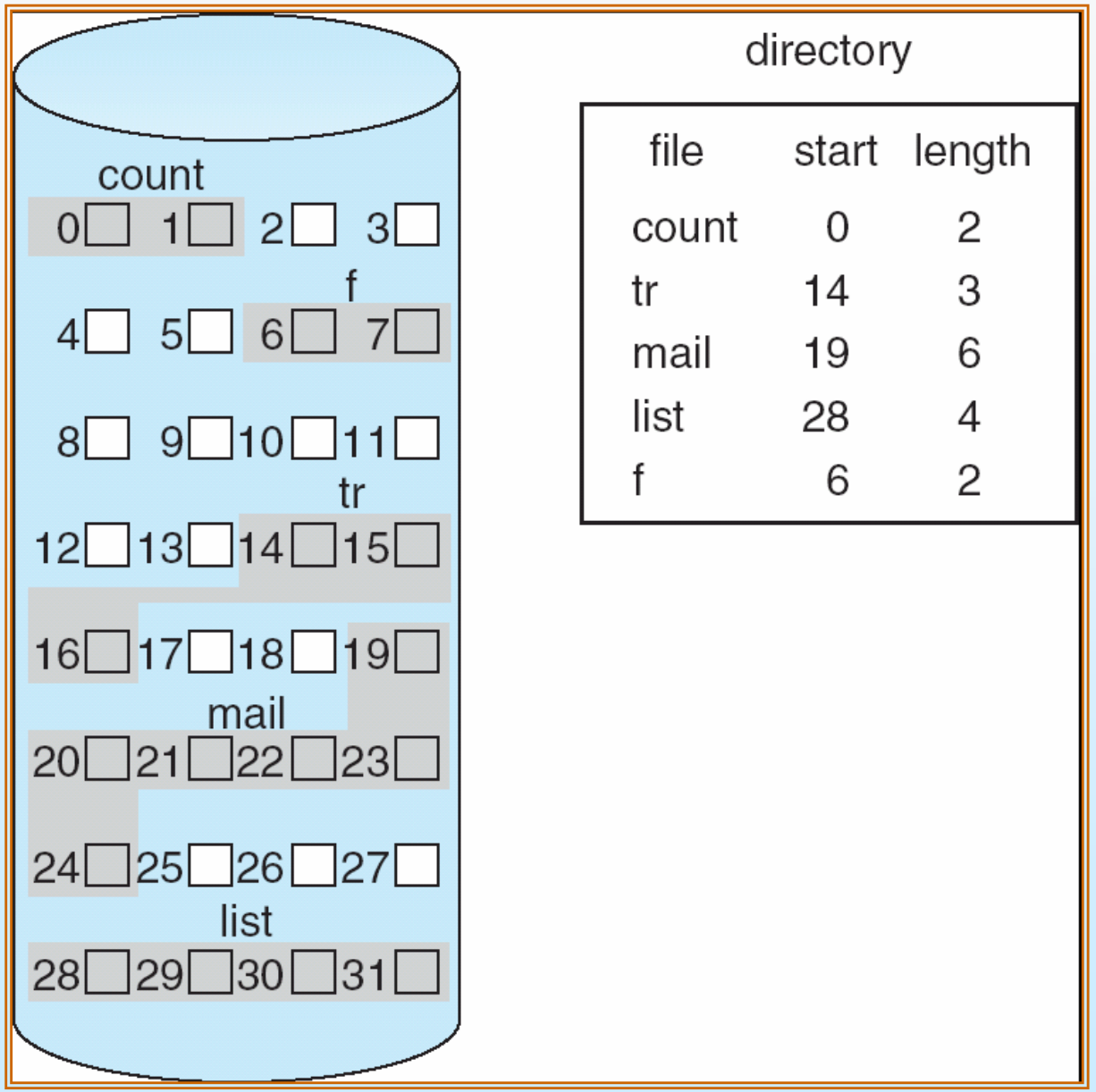
Allocation Methods

- 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
- Wasteful of space (dynamic storage-allocation problem)
  - Satisfy a request of size n from a list of free holes
  - First-fit and best-fit are better than worst-fit solutions in terms of time and storage utilization
- External fragmentation

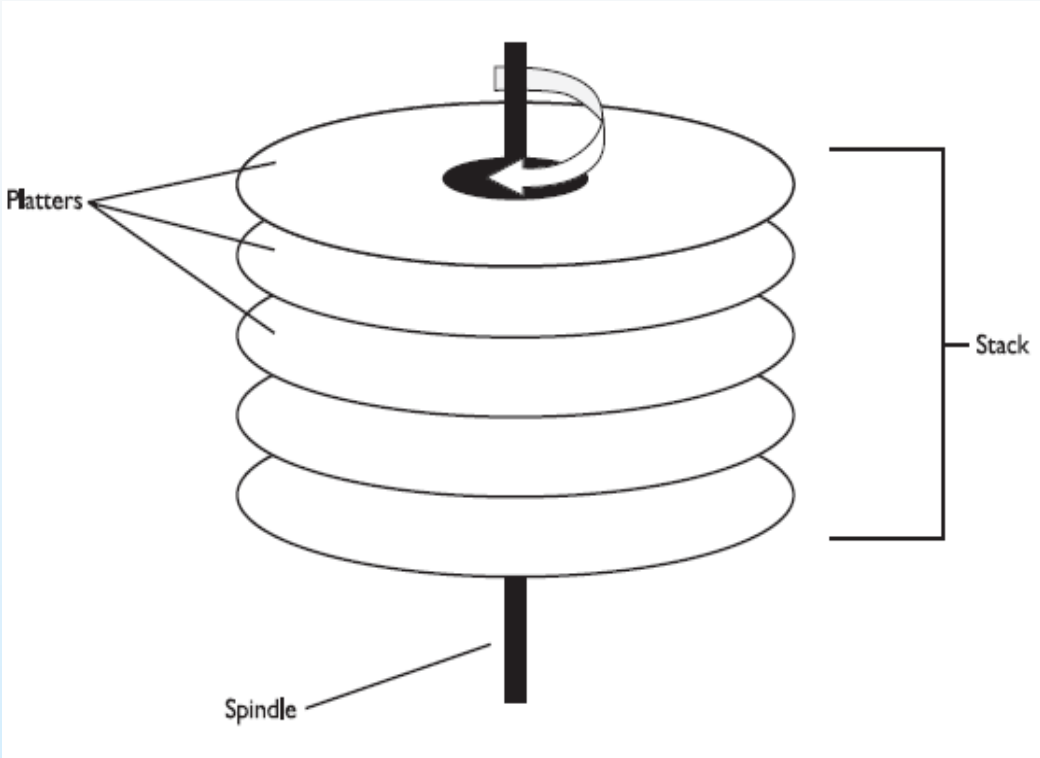
Contiguous Allocation of Disk Space

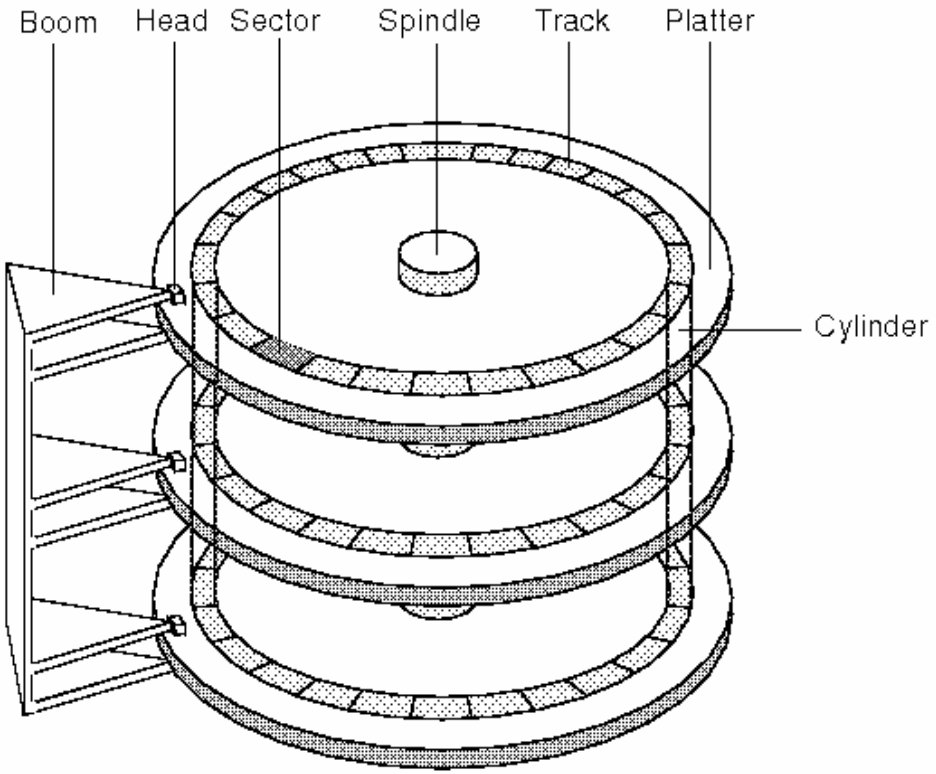


directory

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

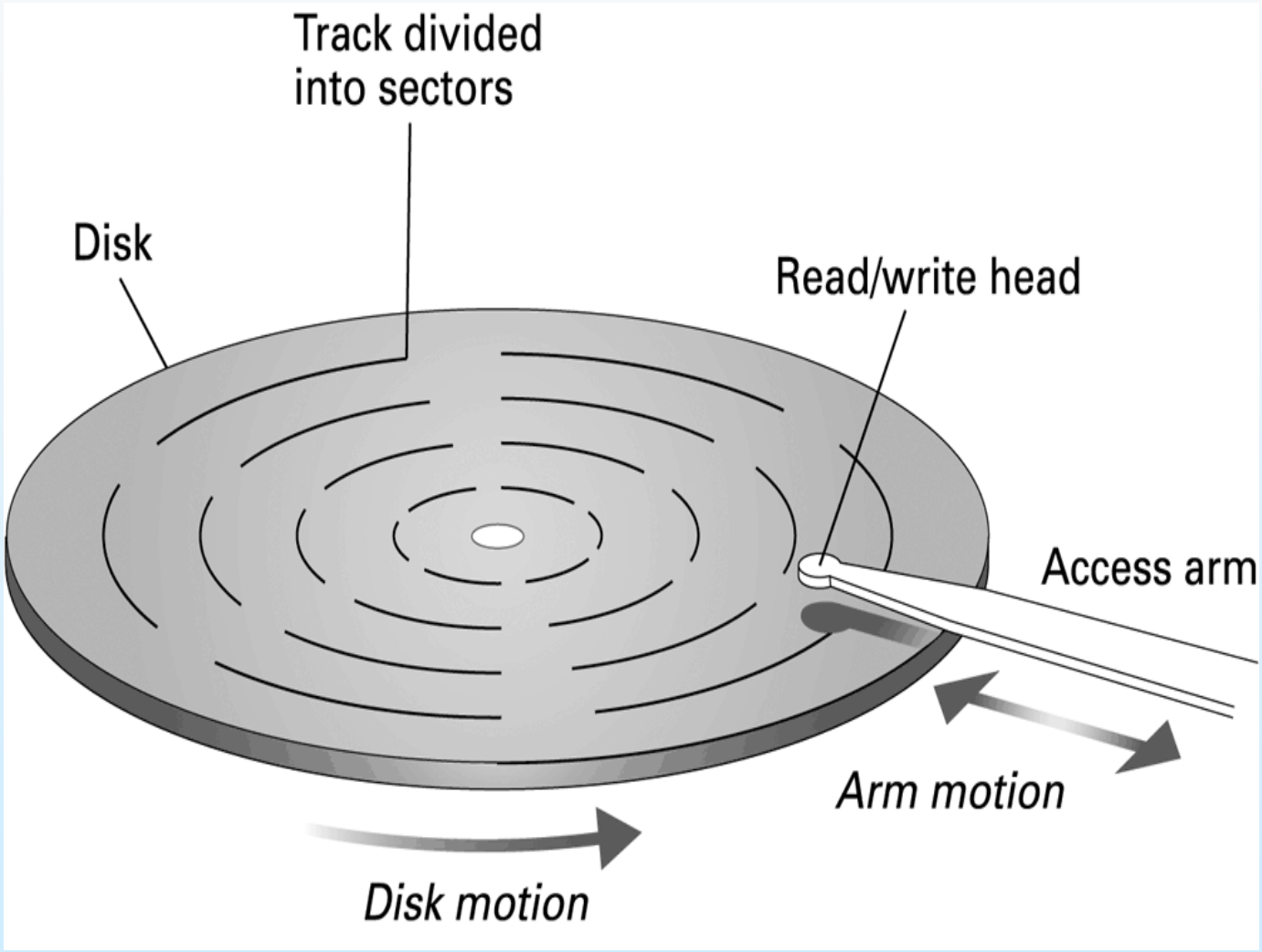
Hard Disk





- The stack of platters is attached through its center to a rotating pole, called a *spindle*.
- Each side of each platter can hold data and has its own read/write head.
- The read/write heads all move as a single unit back and forth along the stack.

Figure 1.9 A magnetic disk storage system



File grow

- How much space is needed for a file? When the file is created, the total amount of space it will need must be found and allocated.
  - How does the creator (program or person) know the size of the file to be created?
  - If we allocate too little space to a file, we may find that the file **cannot be extended.**
- Especially with a best-fit allocation strategy, the space on both sides of the file may be in use.
- Two possibilities:
  - First, the user program can be **terminated**.
    - The user must then **allocate more space** and run the program again.
    - To prevent these restarts, the user will normally **overestimate** the amount of space needed, resulting in considerable wasted space.
  - **Find a larger hole**, copy the contents of the file to the new space and release the previous space.

Extent

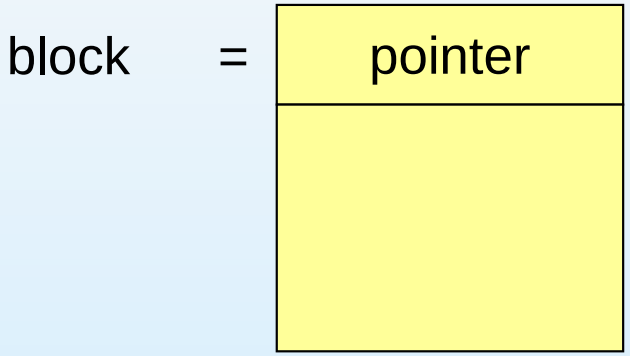
- Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient.
- A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time.
- The file therefore has a large amount of **internal fragmentation**.
- To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme.
  - **Extent**

Extent-Based Systems

- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in **extents**
- An **extent** is a contiguous chunk of space
  - A contiguous chunk of space is allocated initially; and then, if that amount proves not to be large enough, another chunk of contiguous space, known as an extent, is added.
  - A file consists of one or more extents.

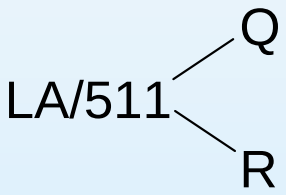
Linked Allocation

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



Linked Allocation (Cont.)

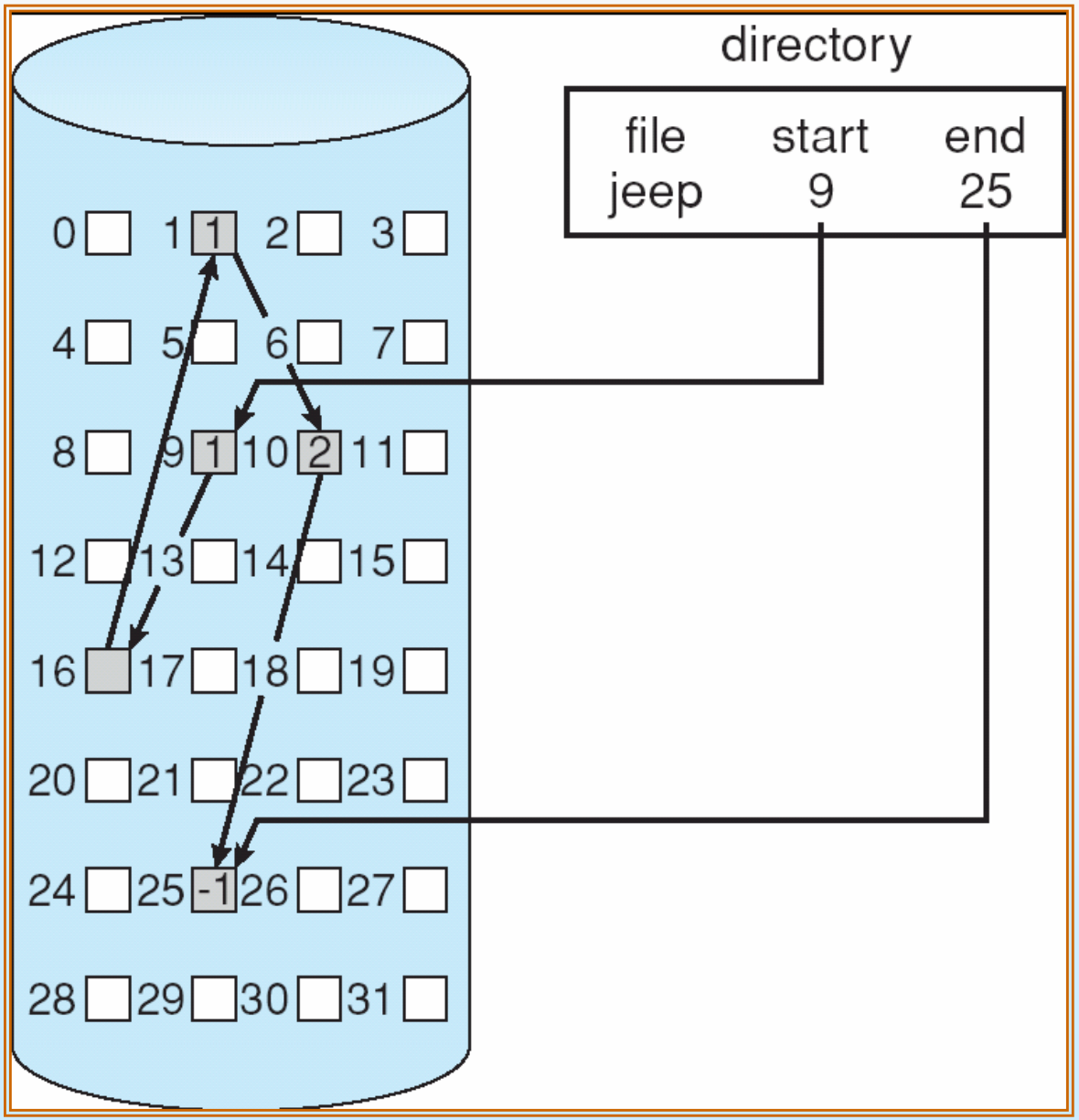
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping



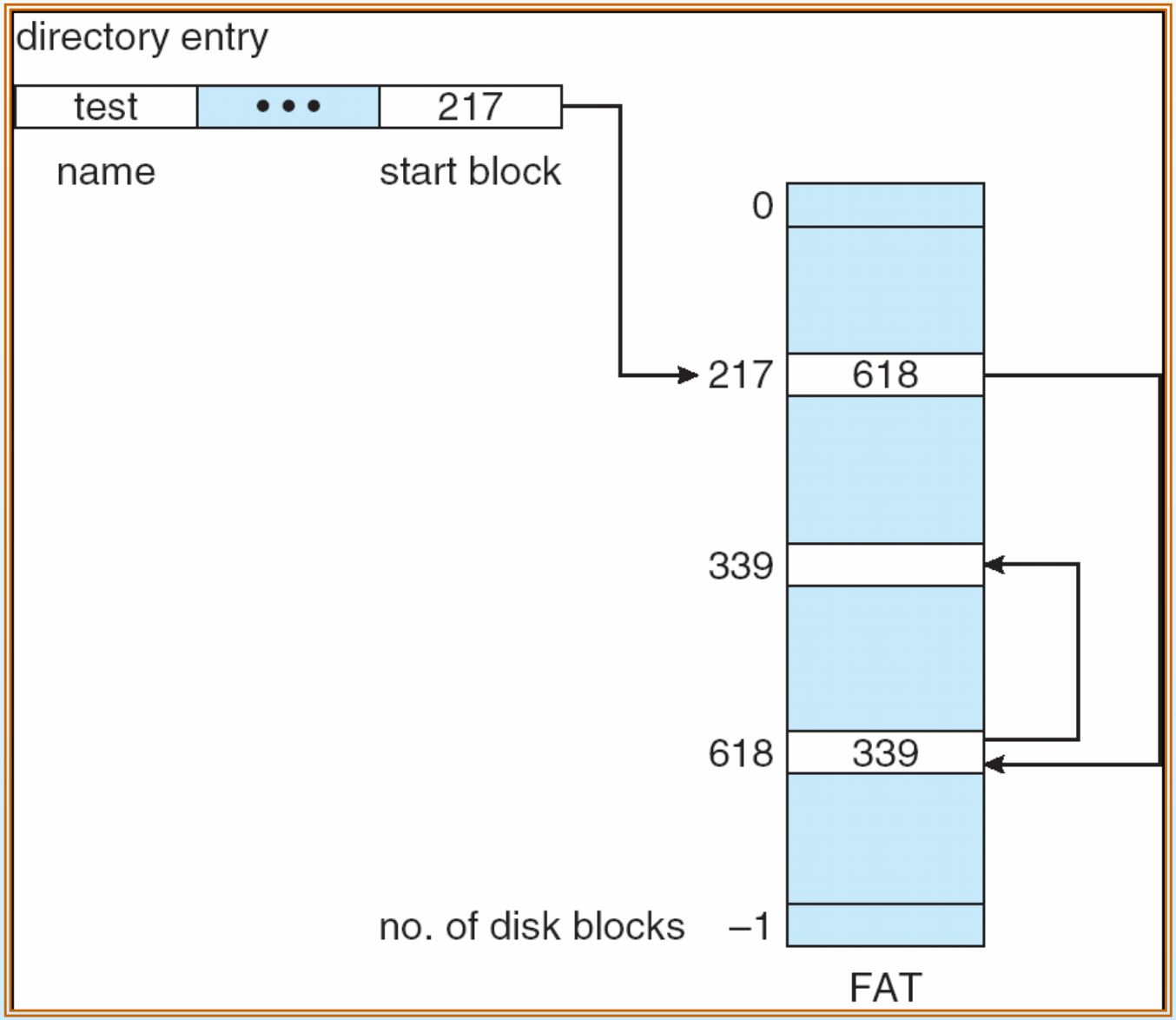
Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1

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

Linked Allocation



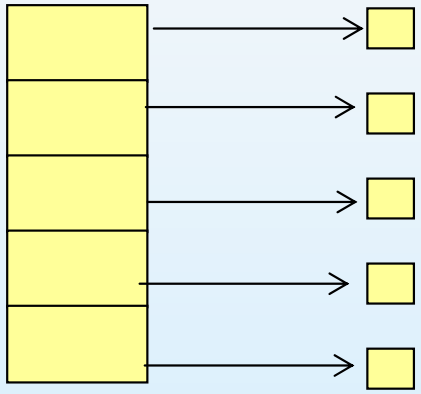
File-Allocation Table



FAT should be cached to gain in performance

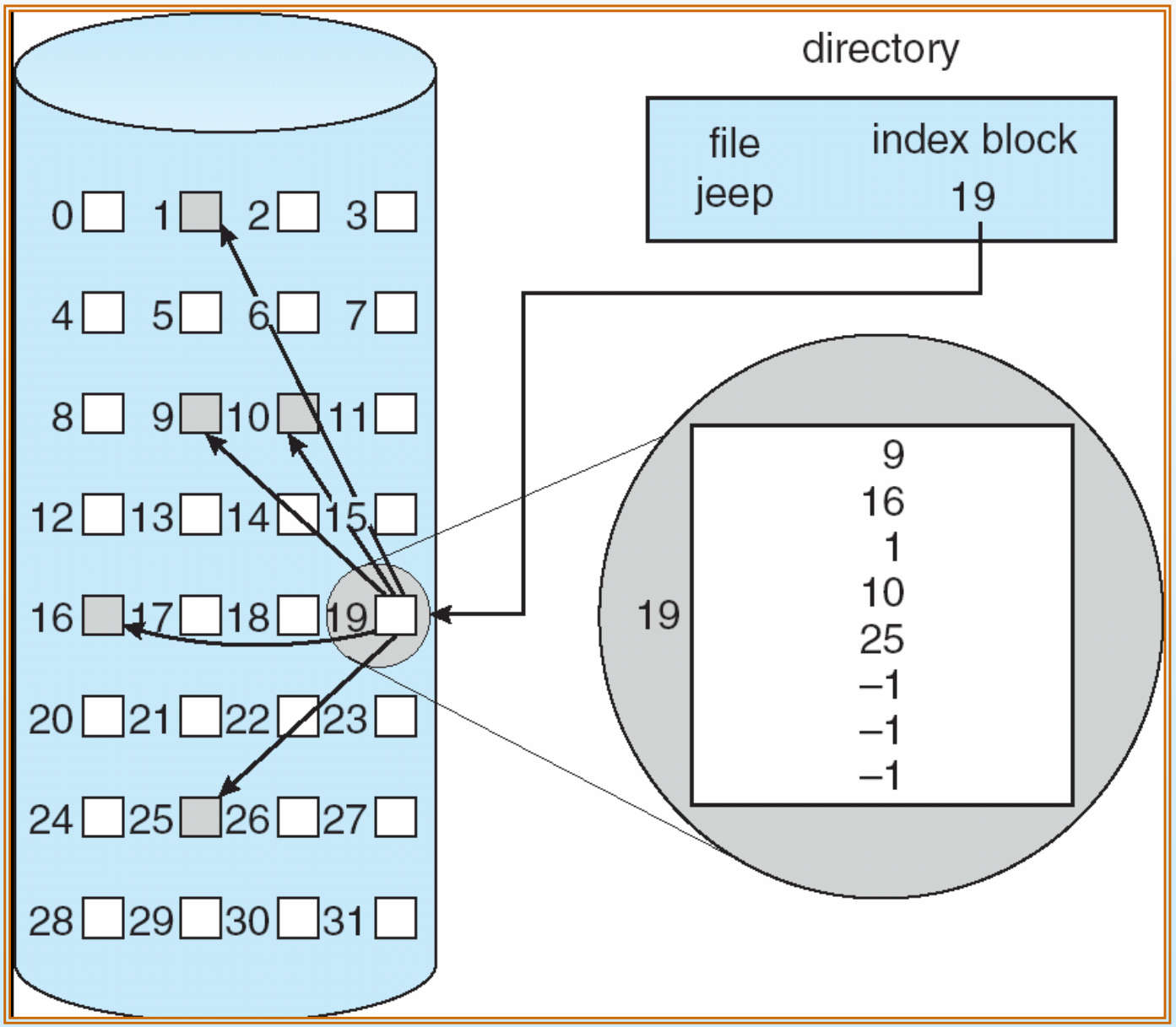
Indexed Allocation

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



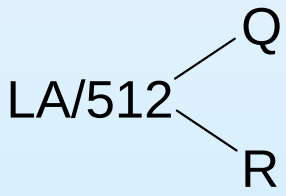
index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

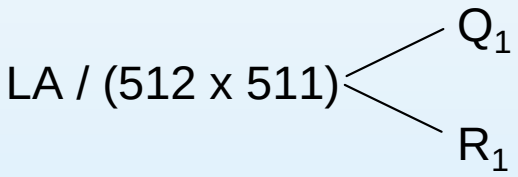
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



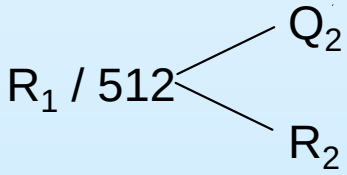
Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

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



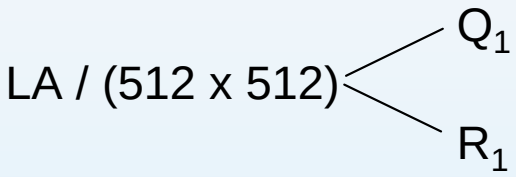
Q_1 = block of index table
R_1 is used as follows:



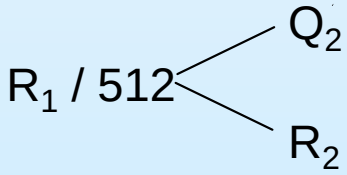
Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

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

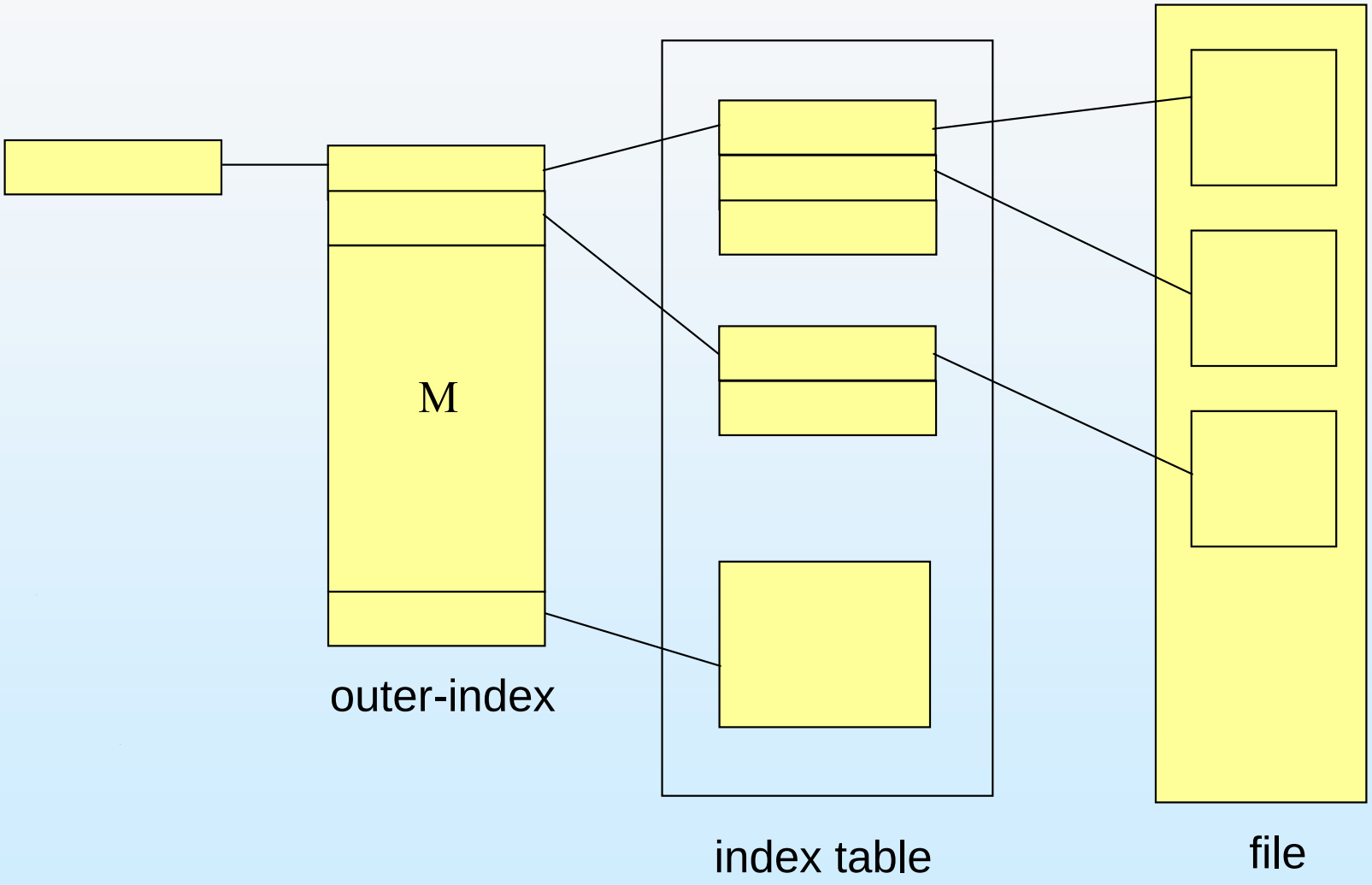


Q_1 = displacement into outer-index
R_1 is used as follows:

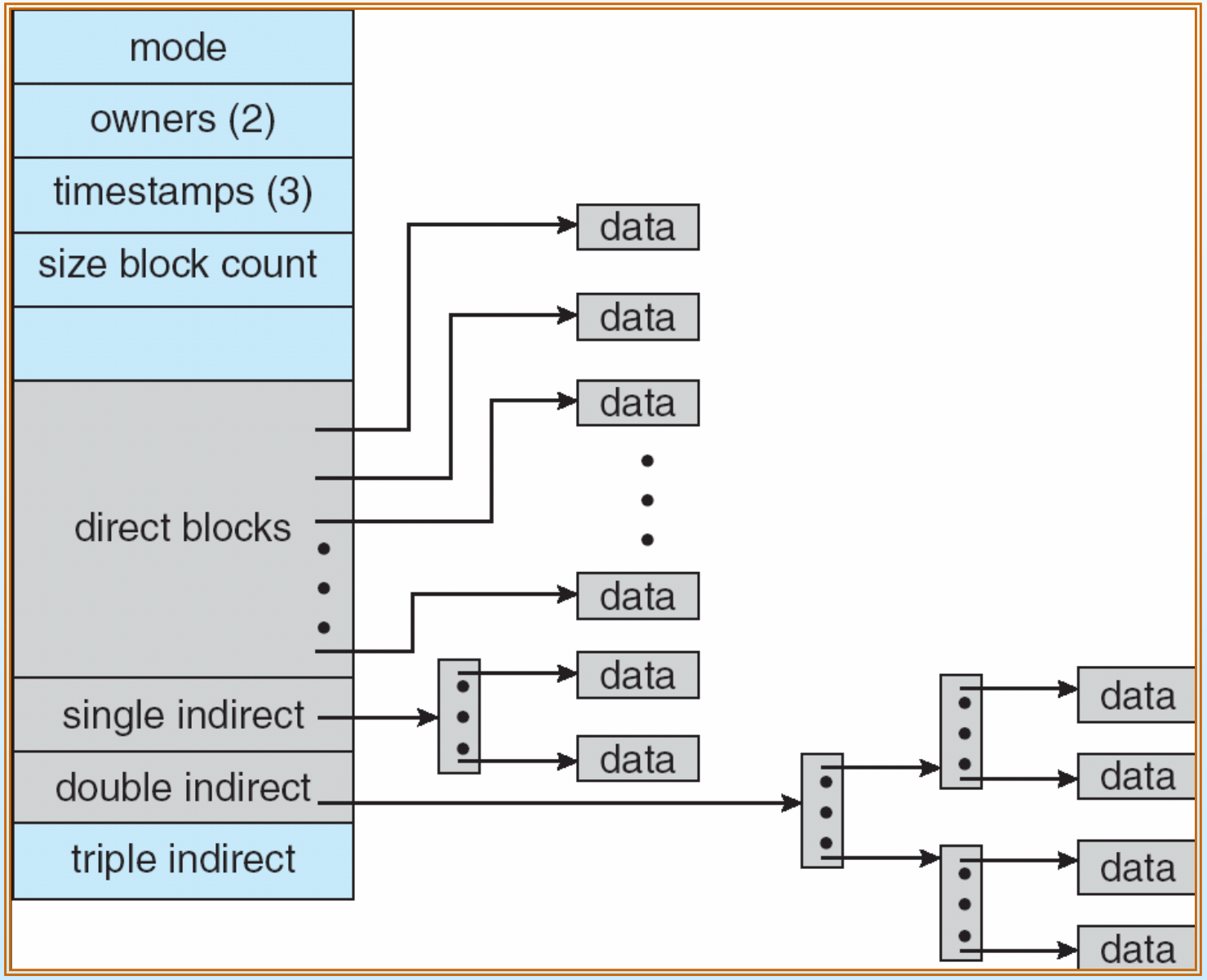


Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

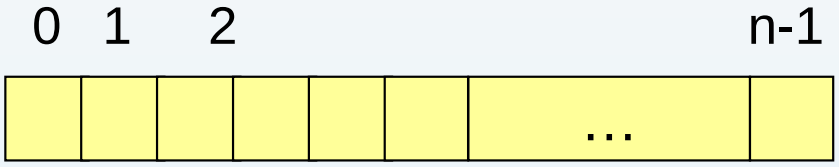


Combined Scheme: UNIX (4K bytes per block)



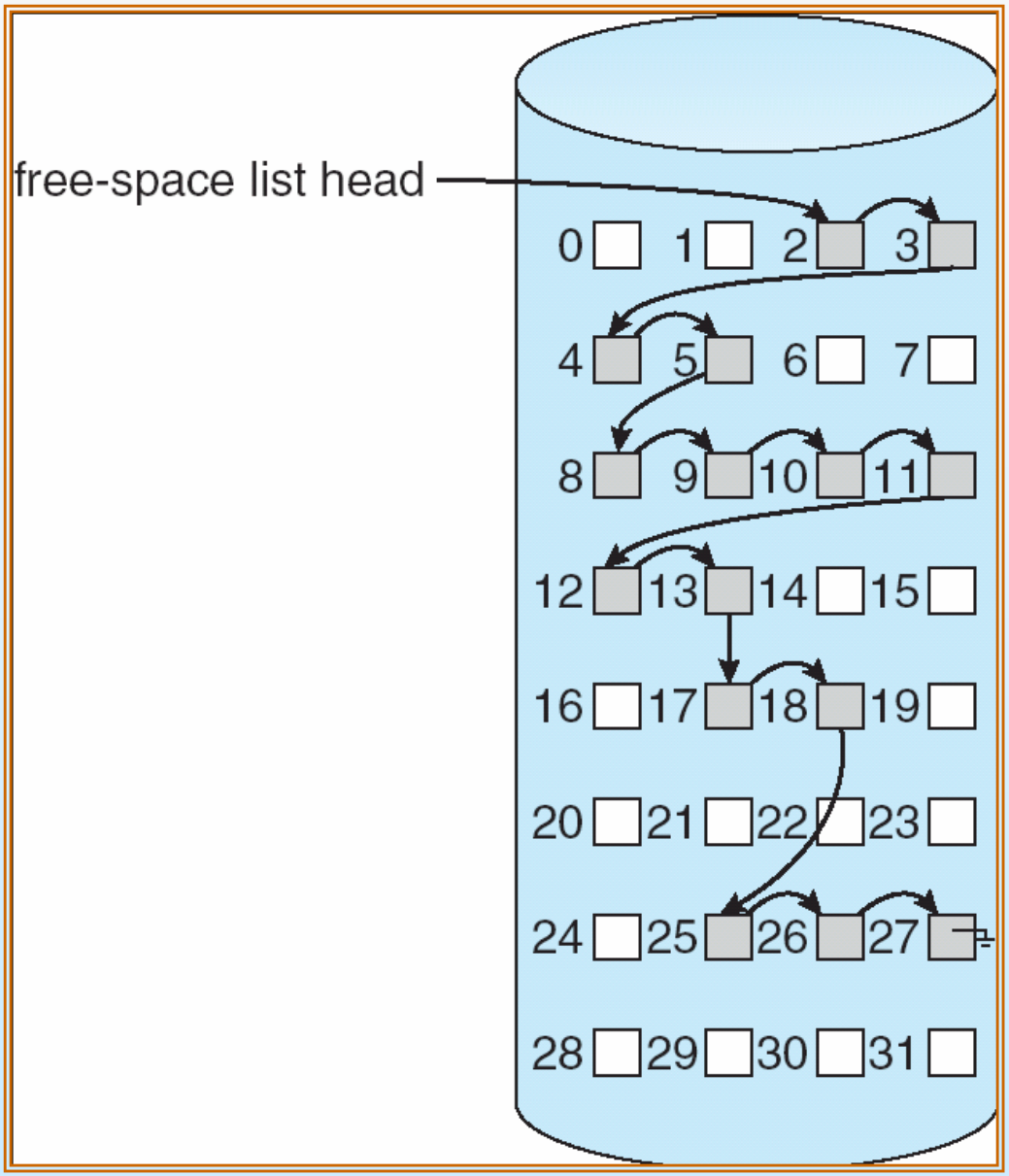
Free-Space Management

- Bit vector (*n* blocks)



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

Linked Free Space List on Disk



Grouping

- A modification of the free-list approach is to **store the addresses of *n* free blocks** in the first free block. The first n-1 of these blocks are actually free.
- The last block contains the addresses of another *n* free blocks, and so on.
- The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Counting

- Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm.

- Thus, rather than keeping a list of n free disk addresses, we can keep the **address of the first free block and the number *n* of free contiguous blocks that follow the first block**.

- Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

Free-Space Management (Cont.)

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - 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
    - Allocate block[*i*]
    - Set bit[*i*] = 1 in memory

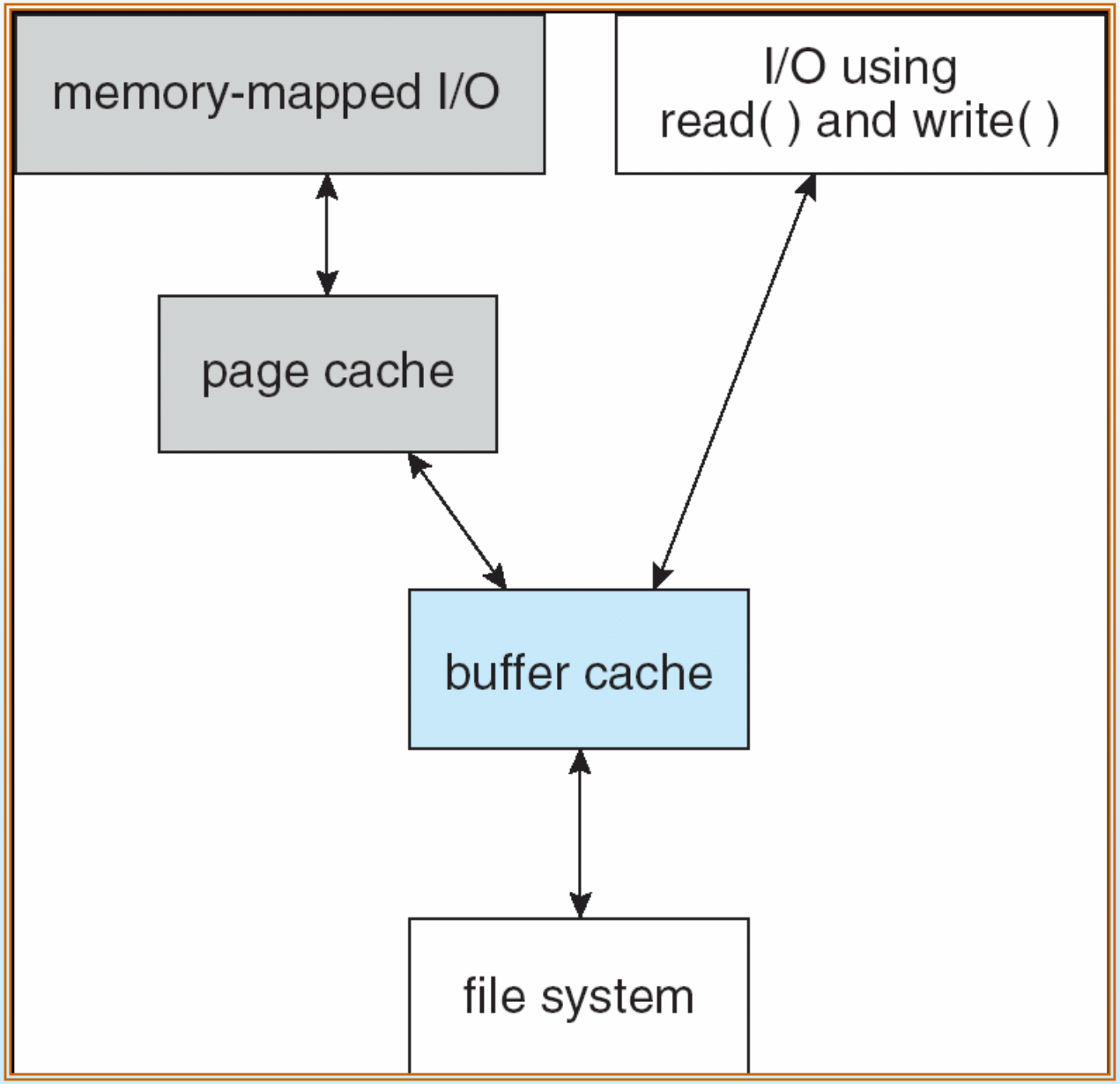
Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance
  - **disk cache** – separate section of main memory for frequently used blocks
  - **free-behind and read-ahead** – techniques to optimize sequential access
    - FB removes a page as soon as next page is requested
    - RA, a requested page and several subsequent pages are read and cached.
  - 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

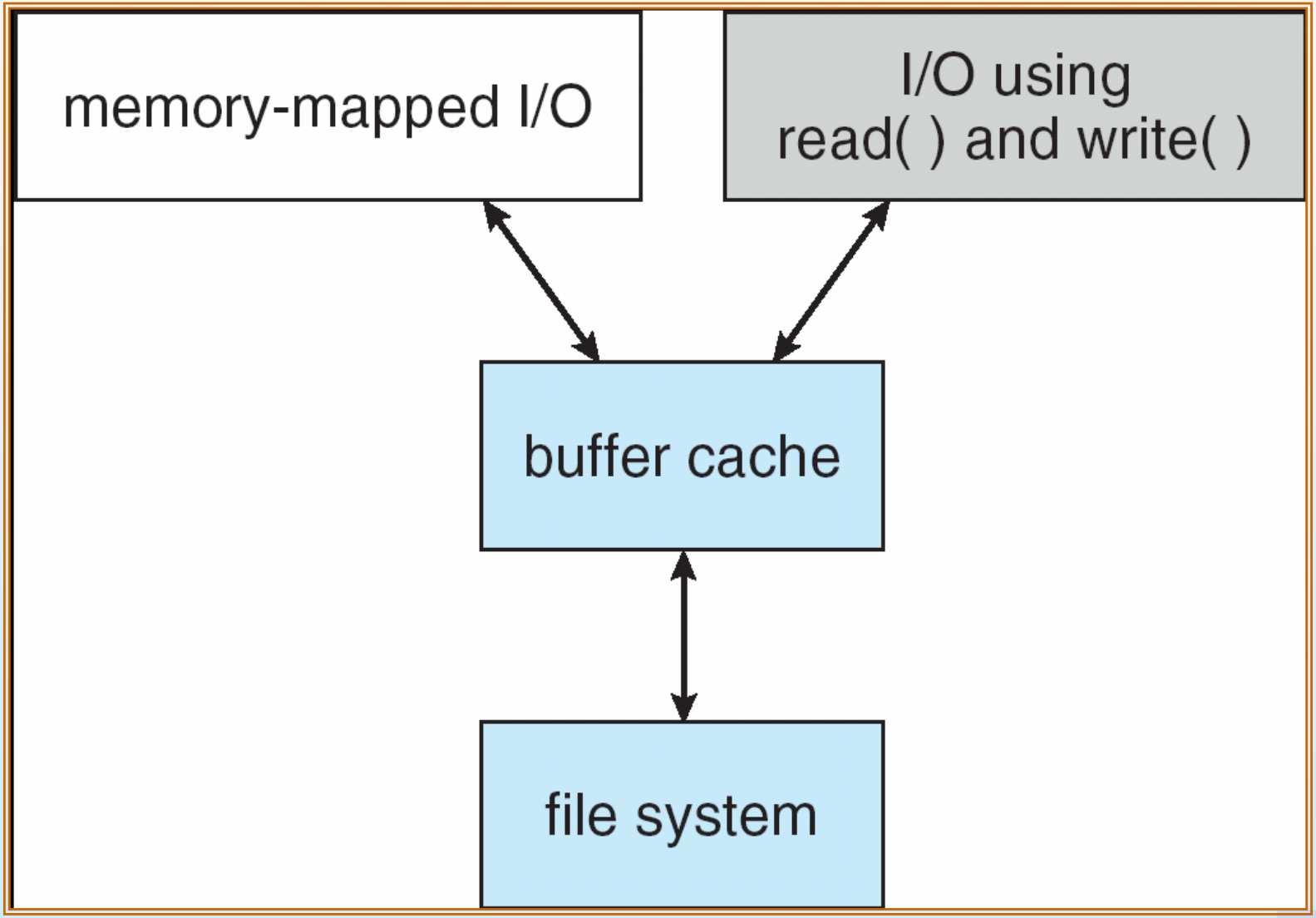
I/O Without a Unified Buffer Cache



Unified Buffer Cache

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

I/O Using a Unified Buffer Cache



Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Use system programs to **back up** data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup
- **Full** or **incremental** back-up

Log Structured File Systems

- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**
- All transactions are written to a **log**
  - A transaction is considered **committed** once it is written to the log
  - **However, the file system may not yet be updated**
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet

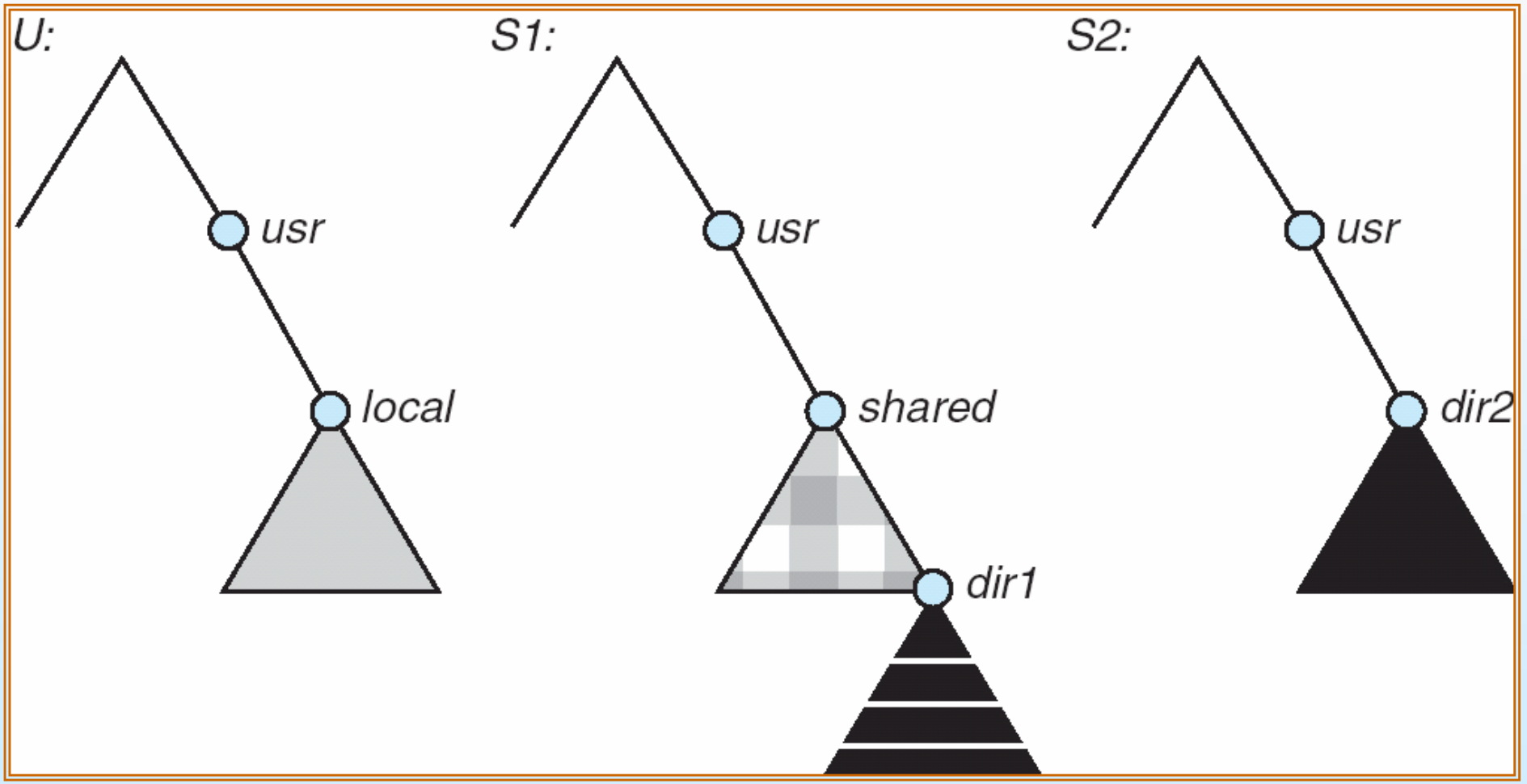
NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a **transparent manner**
  - A remote directory is mounted over a local file system directory
    - The mounted directory looks like an integral **subtree** of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is **nontransparent**; the host name of the remote directory has to be provided
    - Files in the remote directory can then be accessed in a transparent manner
  - Subject to **access-rights accreditation**, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

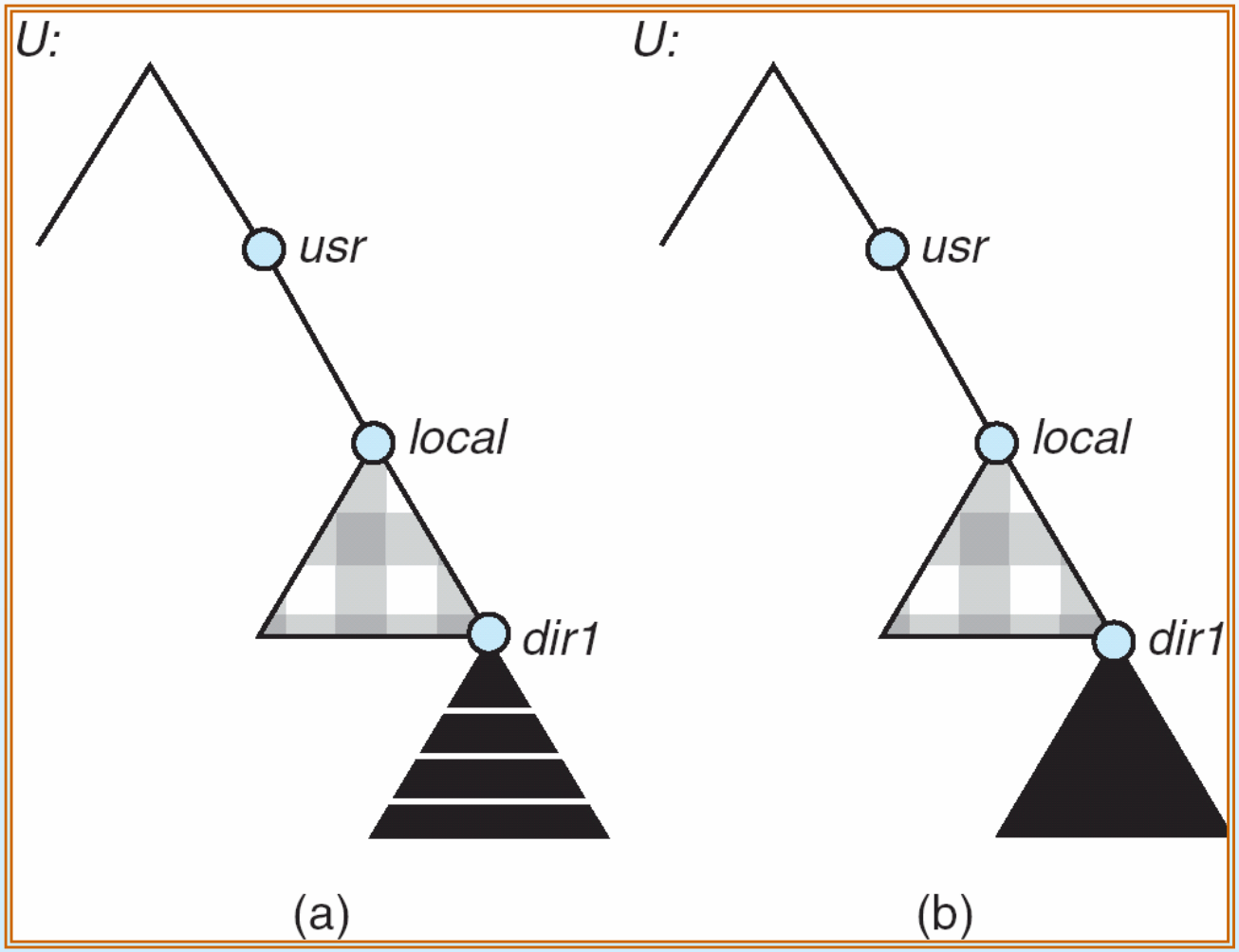
NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications are independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

Three Independent File Systems



Mounting in NFS



Mounts

Cascading mounts

NFS Mount Protocol

- Establishes initial logical connection between **server** and **client**
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - **Mount request** is mapped to corresponding RPC and forwarded to mount server running on server machine
  - **Export list** – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a **file handle** — a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

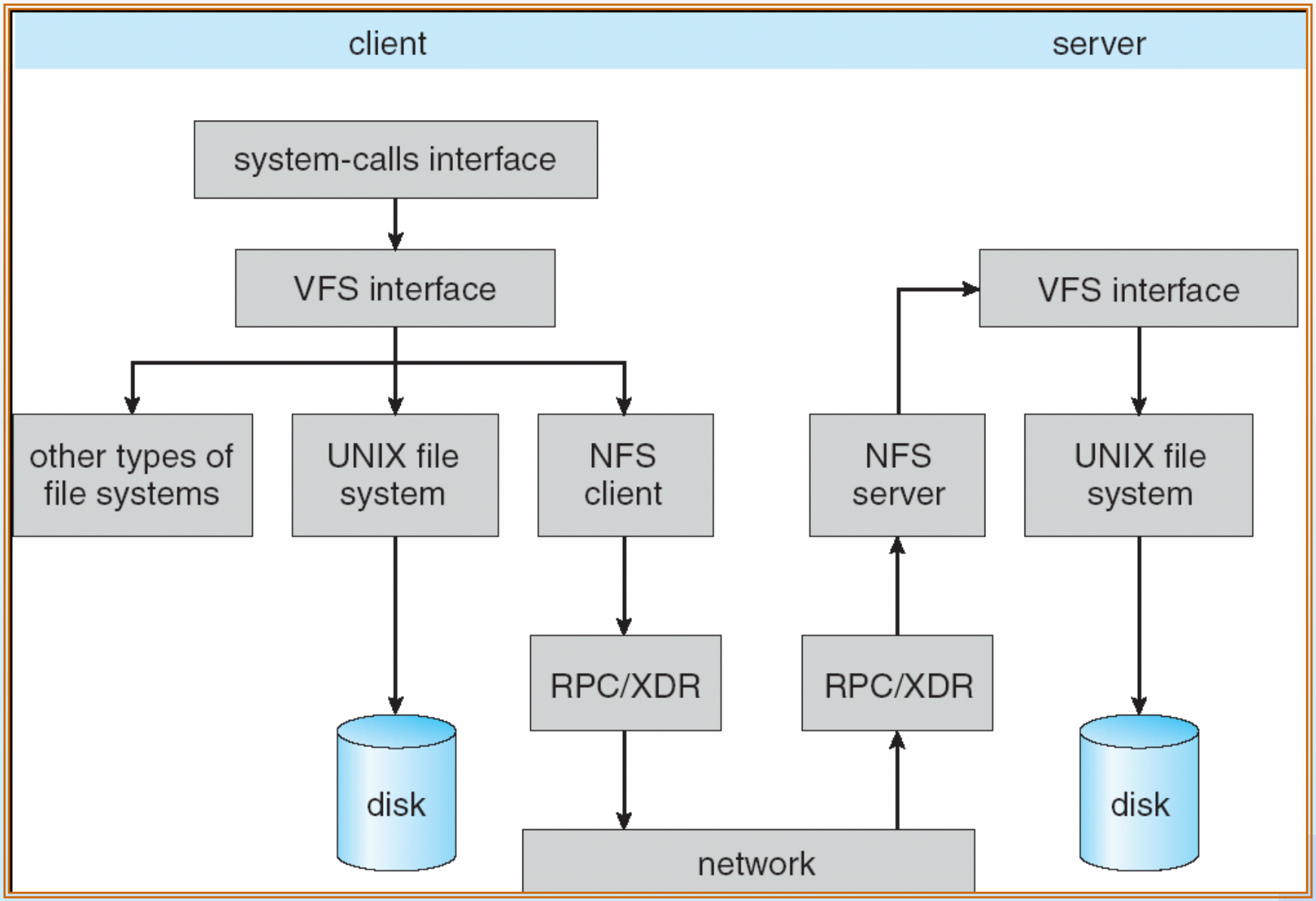
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments
  (NFS V4 has just come available – very different, stateful)
- 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 NFS 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



NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a **directory name lookup cache** on the client's side holds the vnodes for remote directory names

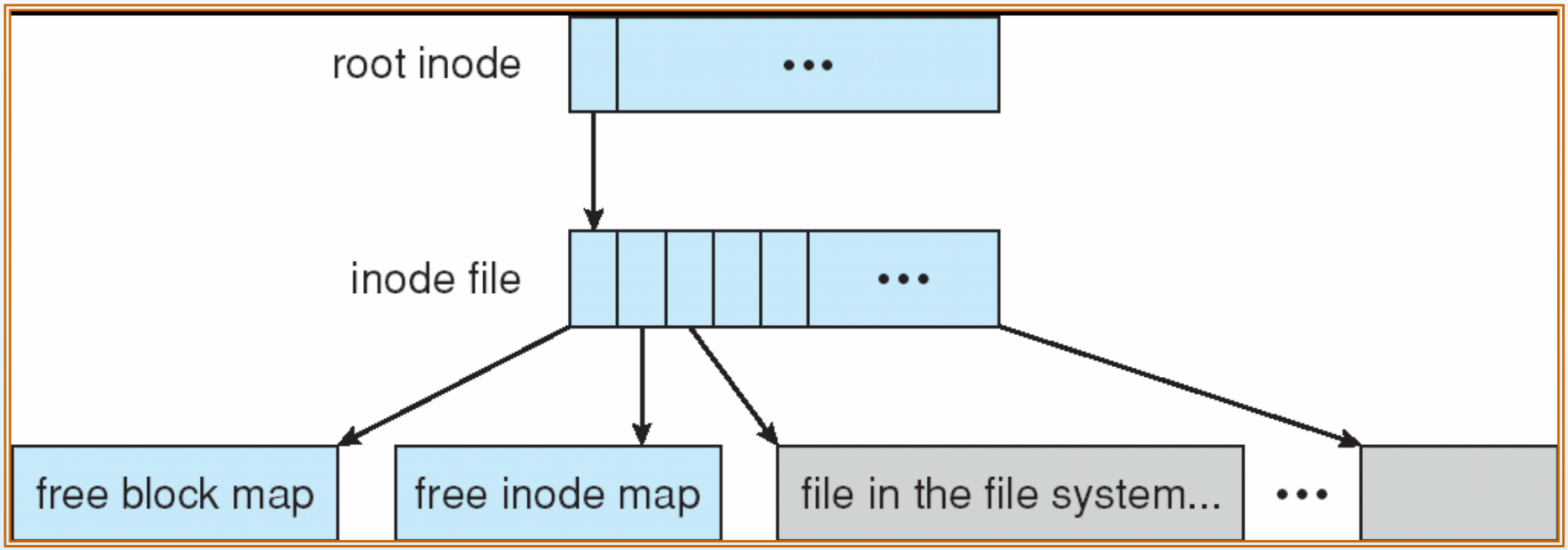
NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the **remote-service paradigm**, but employs buffering and caching techniques for the sake of performance
- **File-blocks cache** – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- **File-attribute cache** – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free **delayed-write blocks** until the server confirms that the data have been written to disk

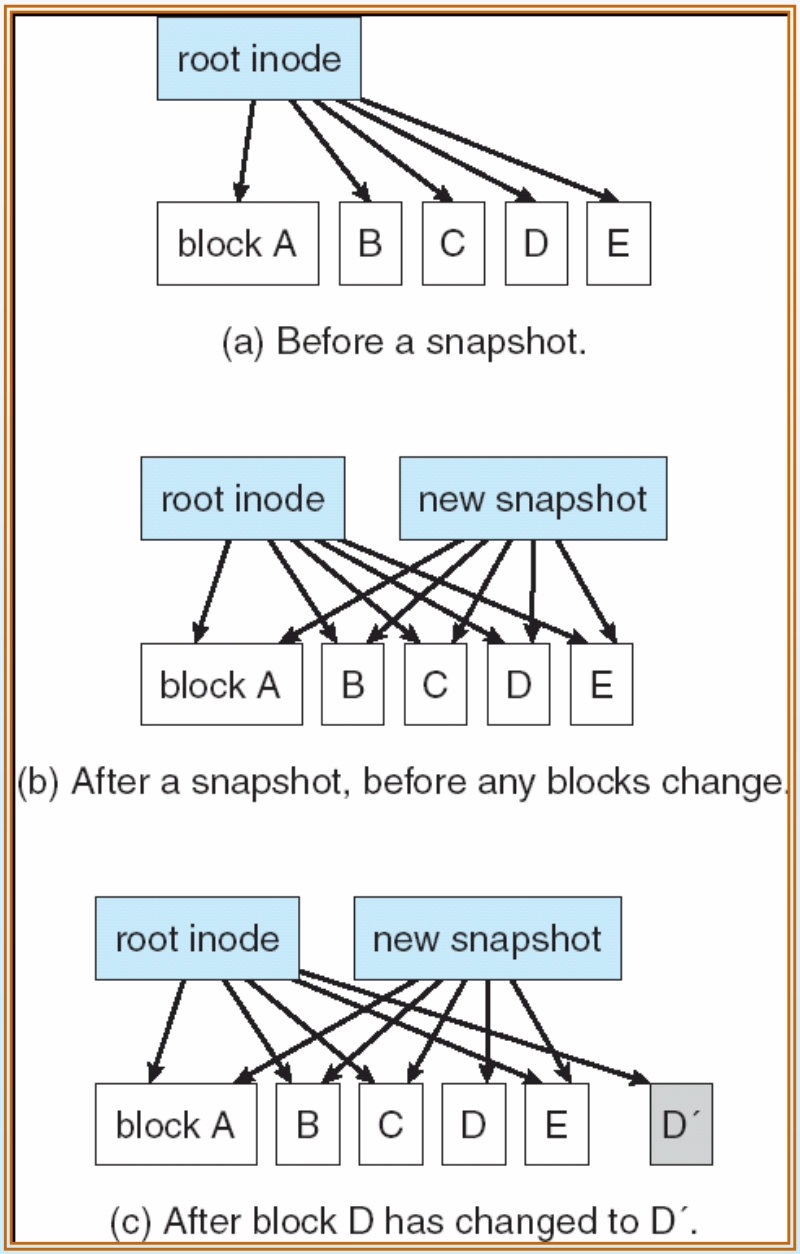
Example: WAFL File System

- 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 cache for write caching
- Similar to Berkeley Fast File System, with extensive modifications

The WAFL File Layout



Snapshots in WAFL



End of Chapter 11