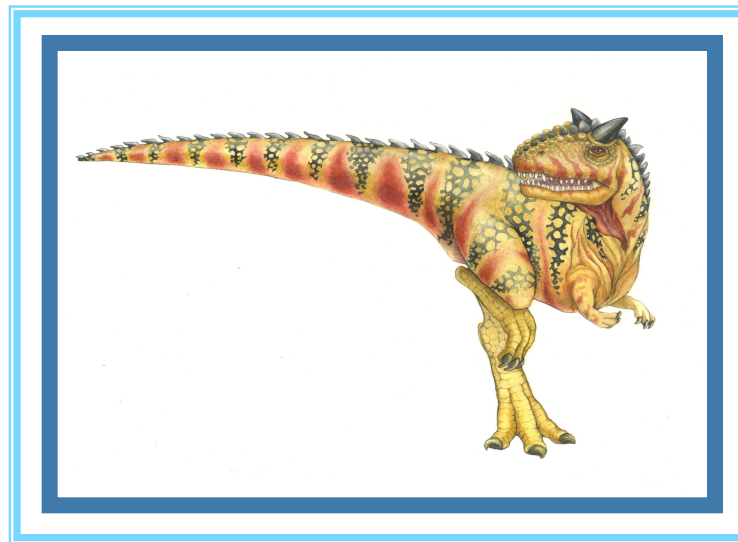


Chapter 11: Implementing File Systems





Chapter 11: Implementing File Systems

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





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





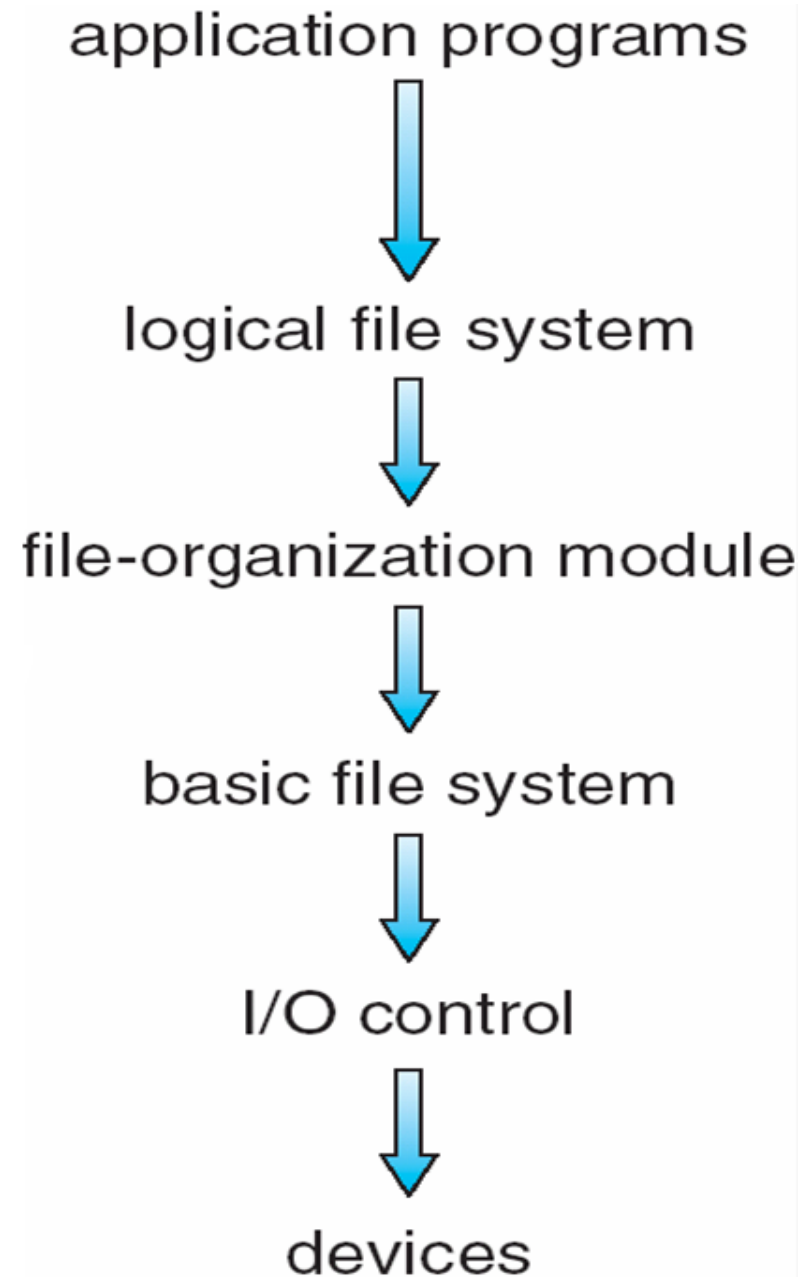
File-System Structure

- File structure
 - Logical storage unit
 - Collection of related information
- **File system** resides on secondary storage (disks)
 - Provided user interface to storage, mapping logical to physical
 - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
 - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers





Layered File System





File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
 - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
 - Buffers hold data in transit
 - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
- Manages free space, disk allocation





File System Layers (Cont.)

- **Logical file system** manages metadata information
 - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in Unix)
 - Directory management
 - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
 - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
 - Each with its own format (CD-ROM is ISO 9660; Unix has **UFS**, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc)
 - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE





File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
 - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
 - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
 - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
 - Names and inode numbers, master file table
- Per-file **File Control Block (FCB)** contains many details about the file
 - Inode number, permissions, size, dates
 - NFTS stores into in master file table using relational DB structures





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





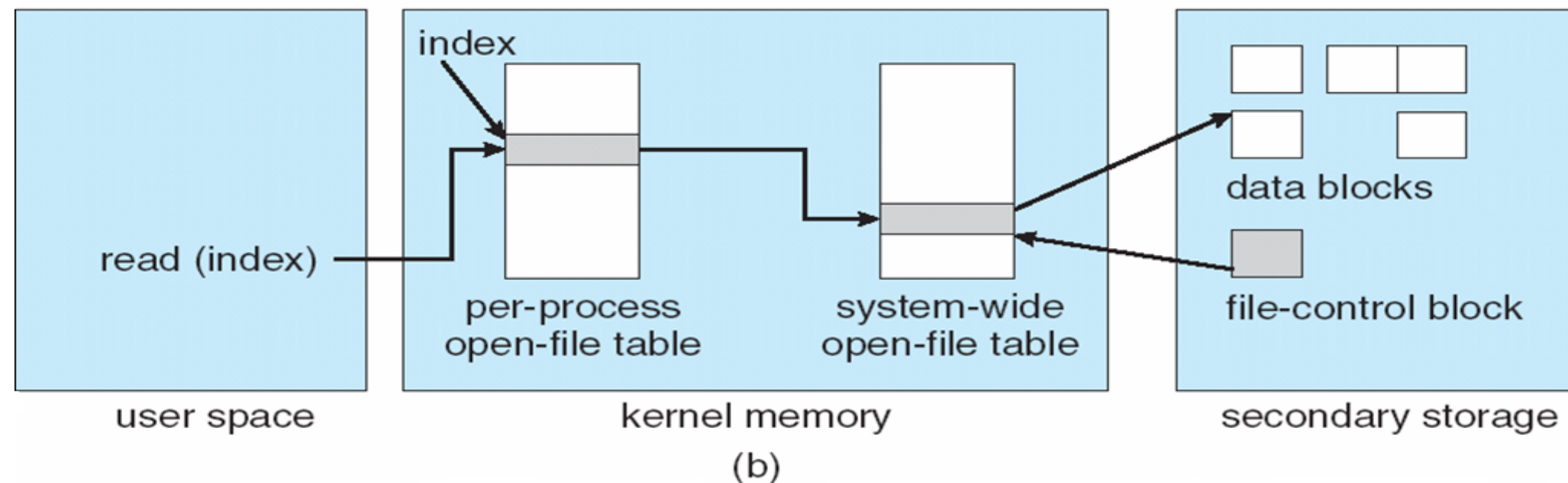
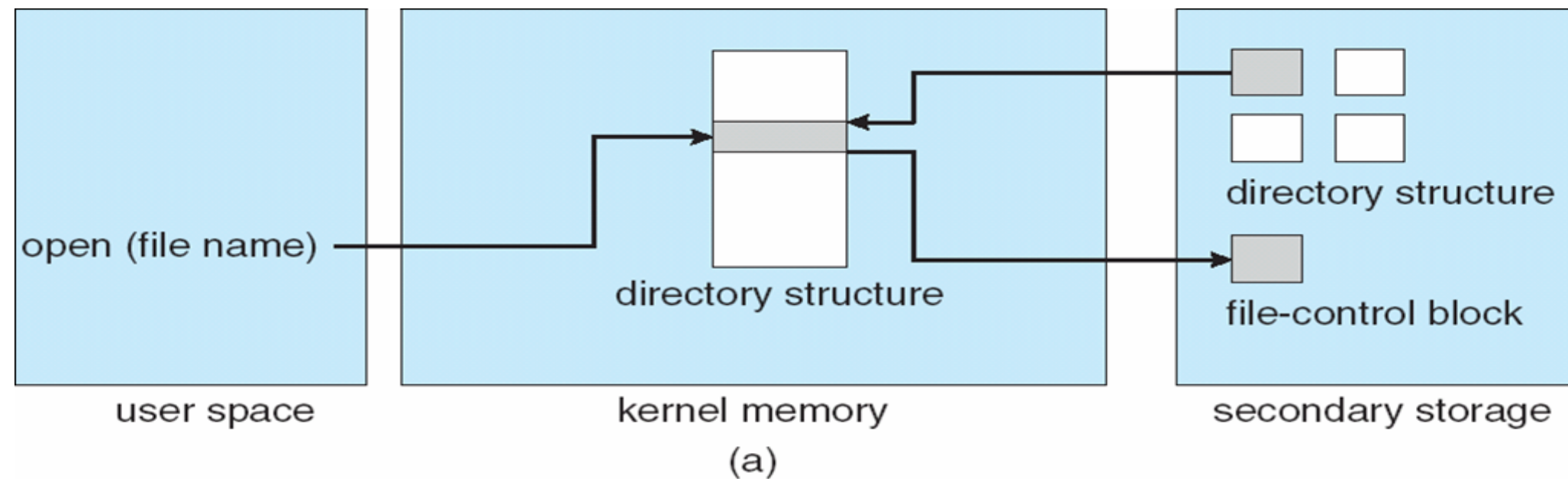
In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 11-3(a) refers to opening a file
- Figure 11-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address





In-Memory File System Structures





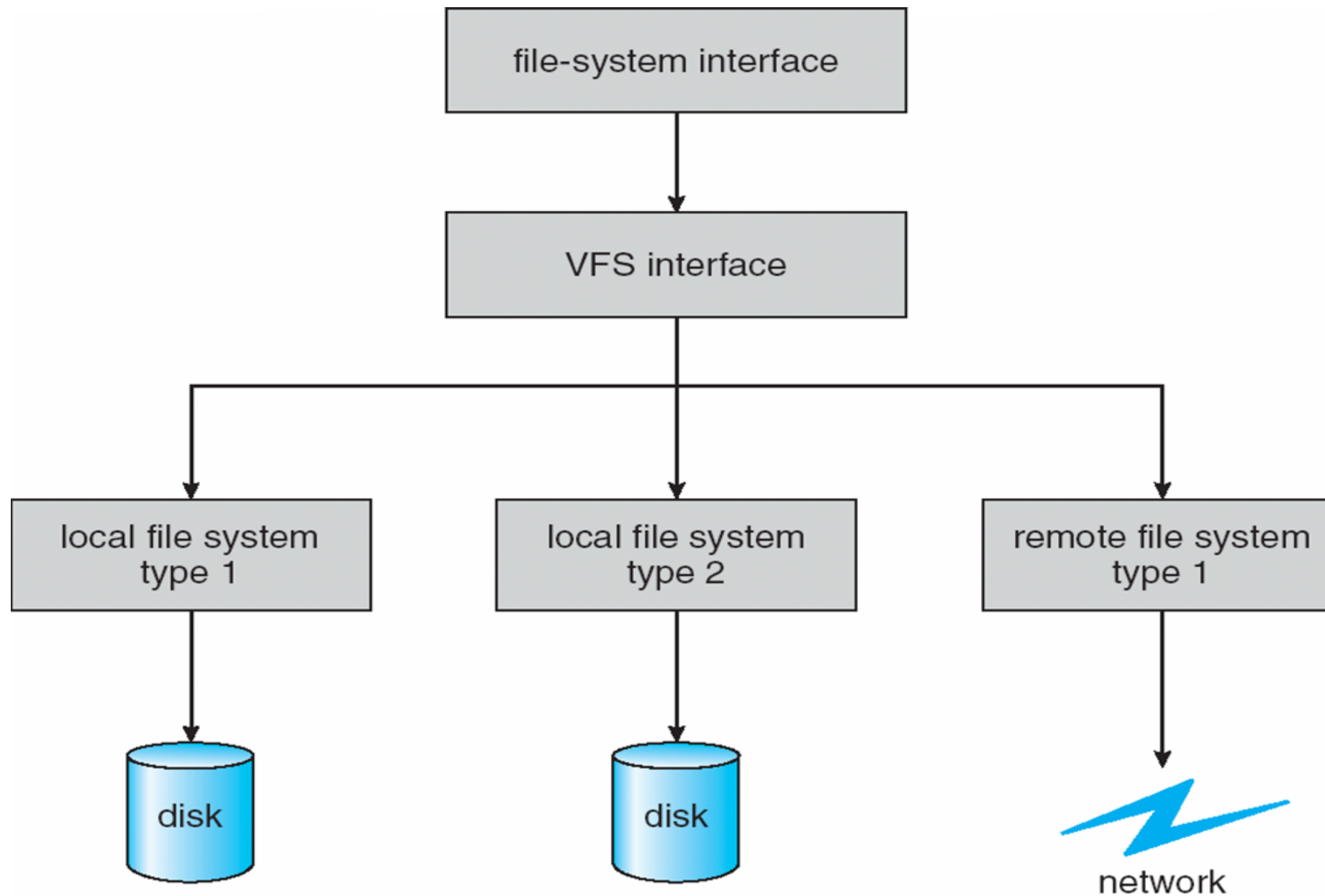
Virtual File Systems

- Virtual File Systems (VFS) on Unix 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
 - Separates file-system generic operations from implementation details
 - Implementation can be one of many file systems types, or network file system
 - ▶ Implements vnodes which hold inodes or network file details
 - Then dispatches operation to appropriate file system implementation routines
- 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 object types:
 - Inode: represents an individual file
 - File: represents an open file
 - Superblock: represents an entire file system
 - Dentry: represents an individual directory entry

- VFS defines set of operations on the objects that must be implemented
 - Every object has a pointer to a function table
 - ▶ Function table has addresses of routines to implement that function on that object





Directory Implementation

- **Linear list** of file names with pointer to the data blocks
 - Simple to program
 - Time-consuming to execute
 - ▶ Linear search time
 - ▶ Could keep ordered alphabetically via linked list or use B+ tree

- **Hash Table** – linear list with hash data structure
 - Decreases directory search time
 - **Collisions** – situations where two file names hash to the same location
 - Only good if entries are fixed size, or use chained-overflow method





Allocation Methods - Contiguous

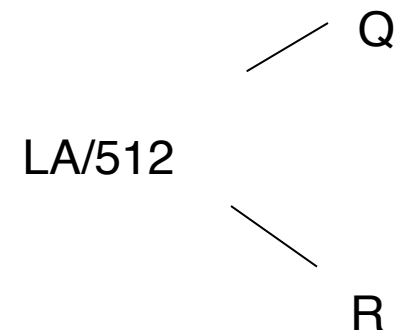
- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
 - Best performance in most cases
 - Simple – only starting location (block #) and length (number of blocks) are required
 - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line** (**downtime**) or **on-line**





Contiguous Allocation

- Mapping from logical to physical

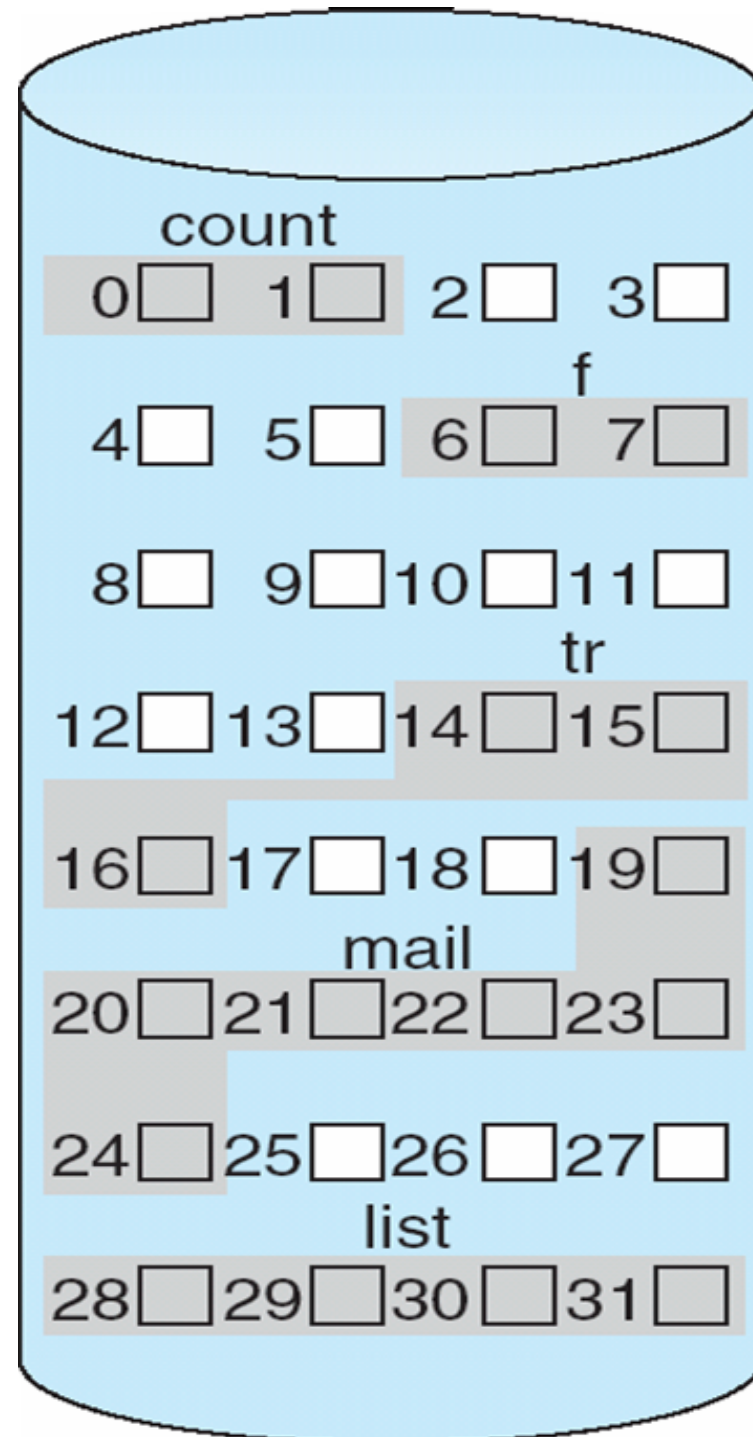


Block to be accessed = $Q + \text{starting address}$
Displacement into block = R





Contiguous Allocation of Disk Space



directory

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





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 block of disks
 - Extents are allocated for file allocation
 - A file consists of one or more extents

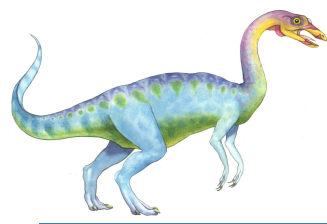




Allocation Methods - Linked

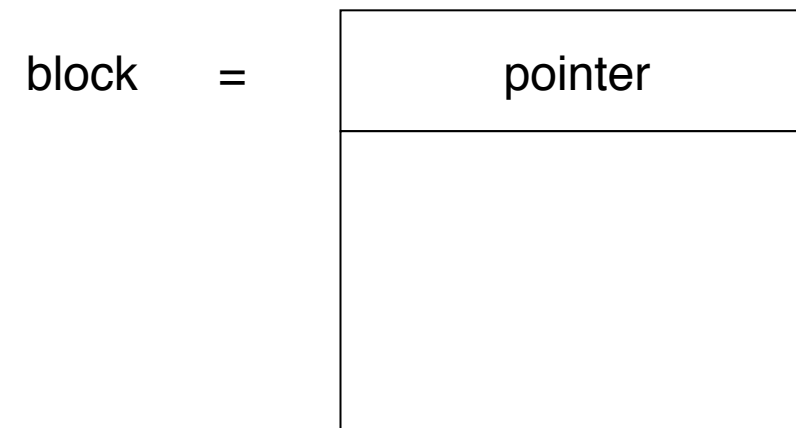
- **Linked allocation** – each file a linked list of blocks
 - File ends at nil pointer
 - No external fragmentation
 - Each block contains pointer to next block
 - No compaction, external fragmentation
 - Free space management system called when new block needed
 - Improve efficiency by clustering blocks into groups but increases internal fragmentation
 - Reliability can be a problem
 - Locating a block can take many I/Os and disk seeks
- **FAT (File Allocation Table) variation**
 - Beginning of volume has table, indexed by block number
 - Much like a linked list, but faster on disk and cacheable
 - New block allocation simple





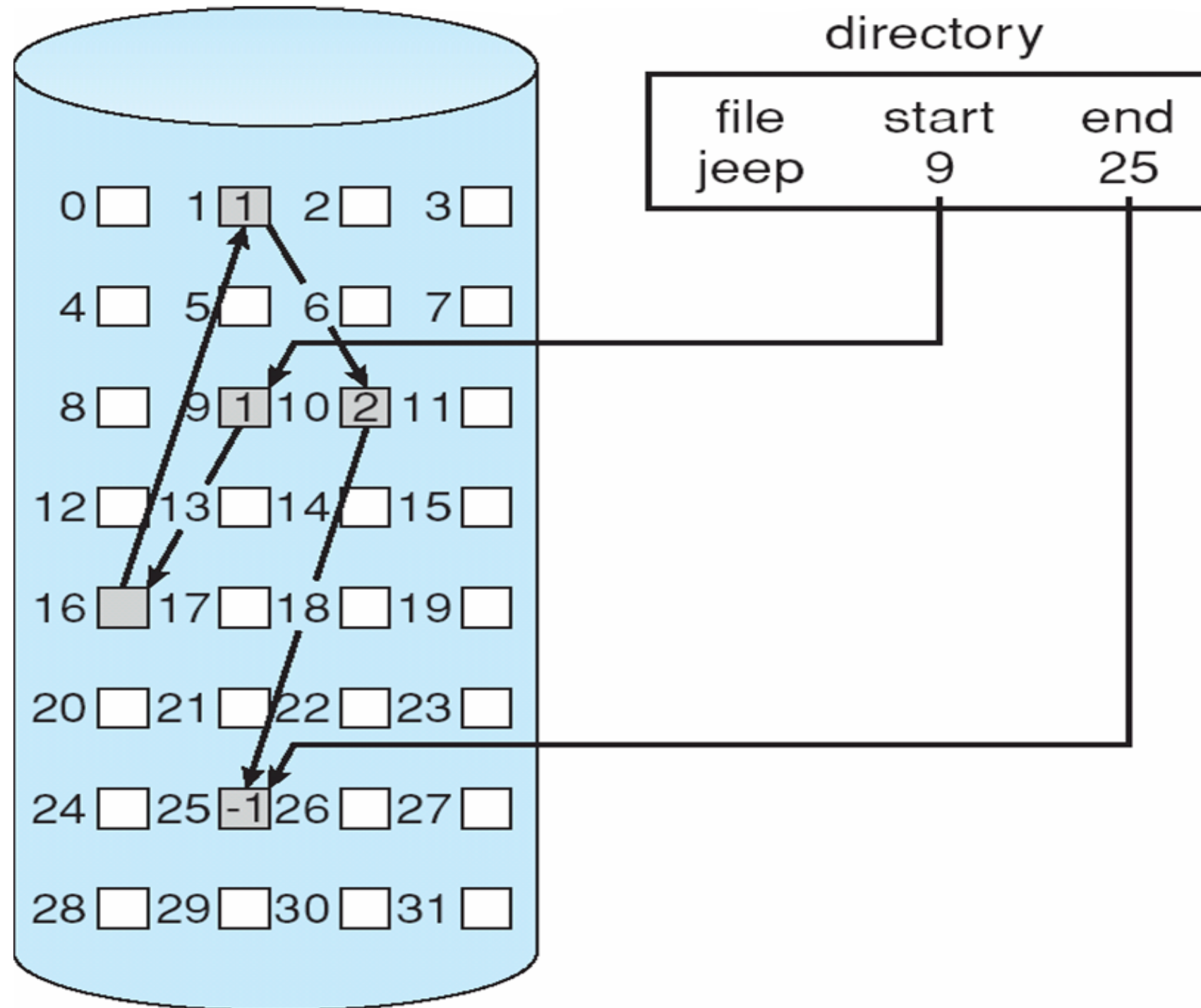
Linked Allocation

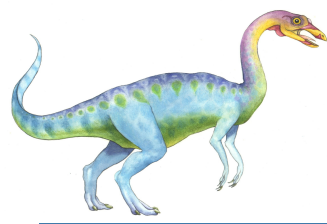
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



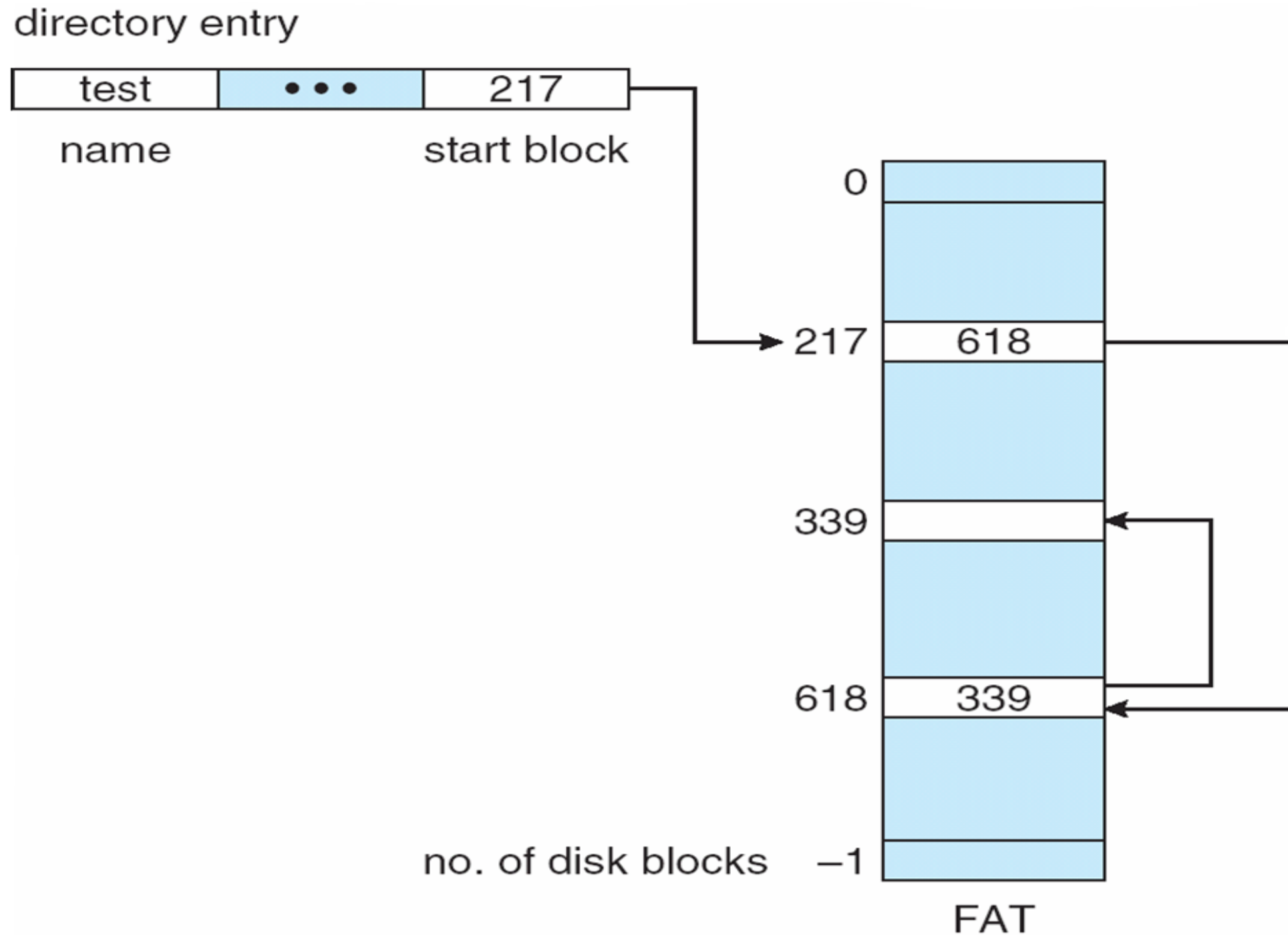


Linked Allocation





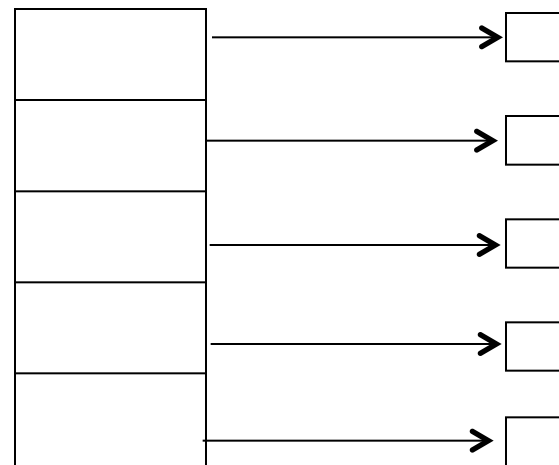
File-Allocation Table





Allocation Methods - Indexed

- **Indexed allocation**
 - Each file has its own **index block**(s) of pointers to its data blocks
- Logical view

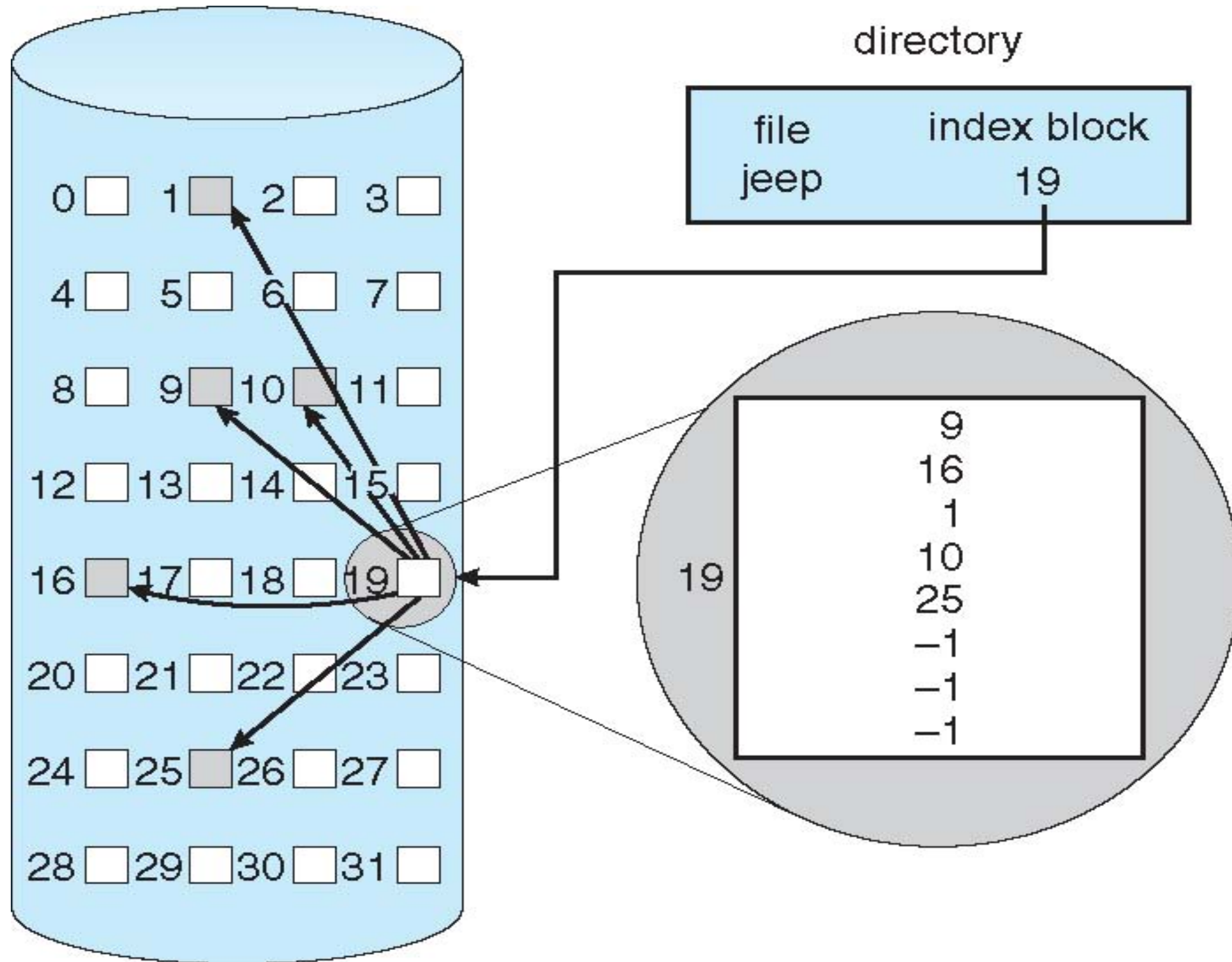


index table



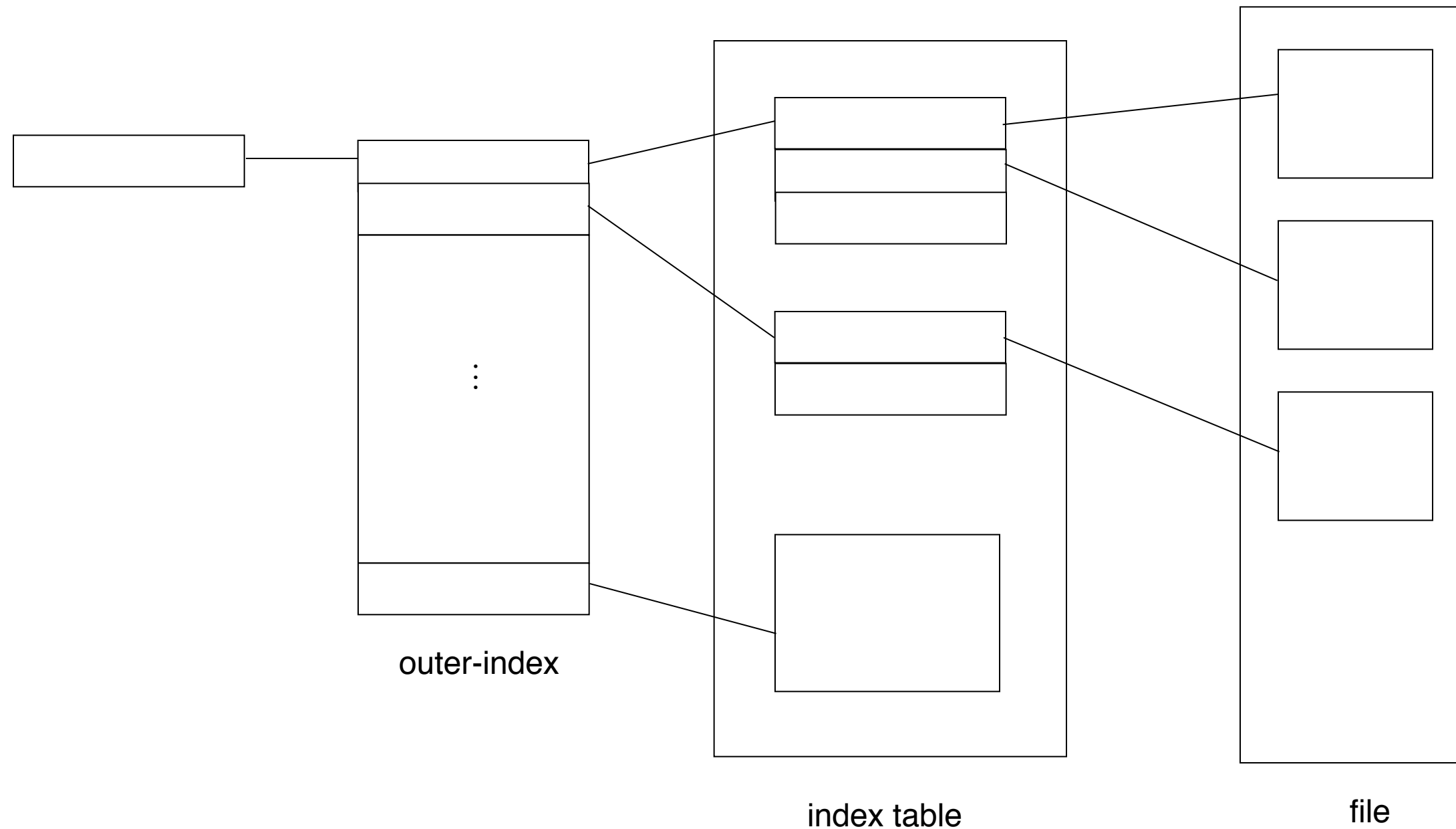


Example of Indexed Allocation



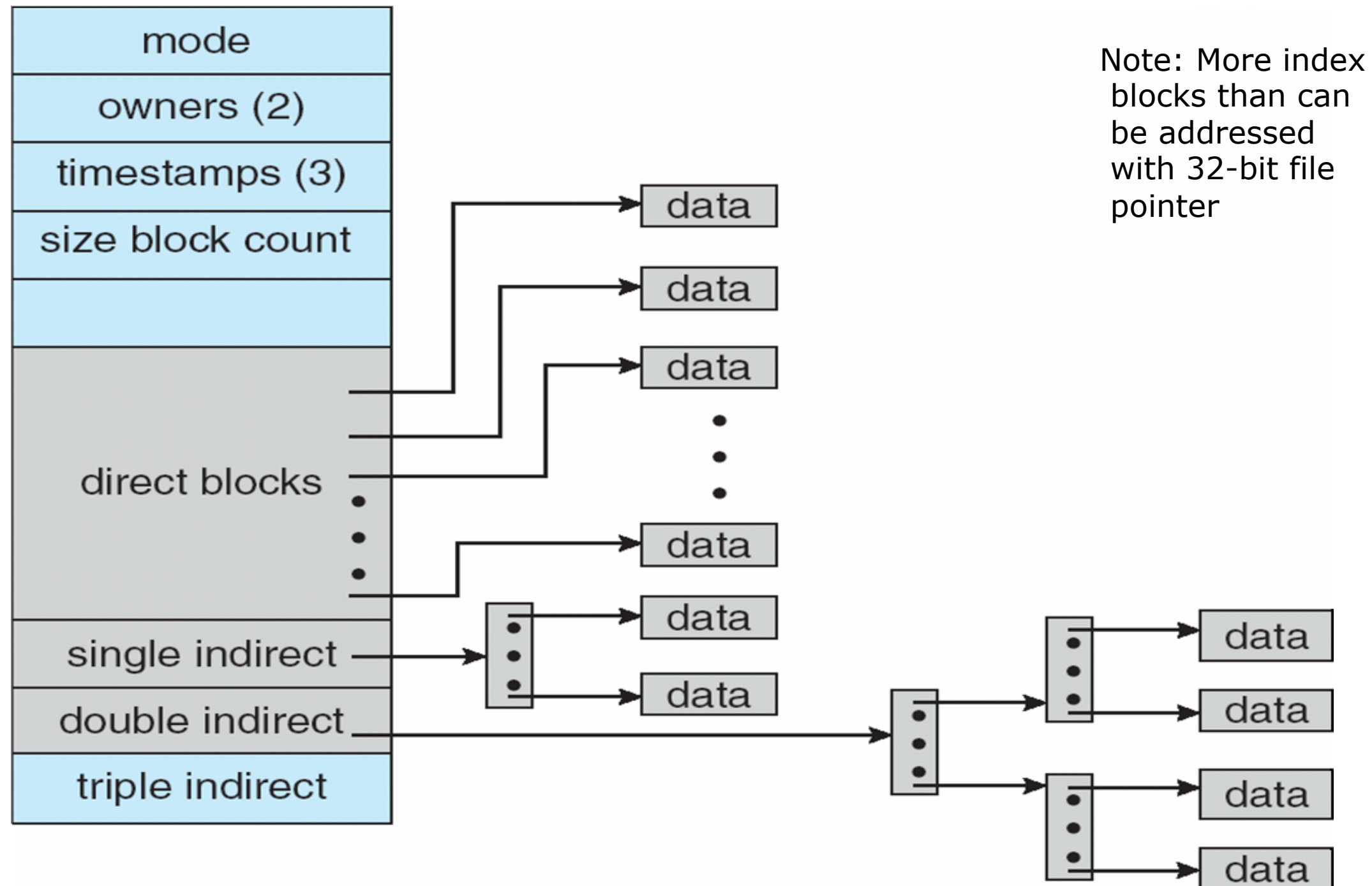


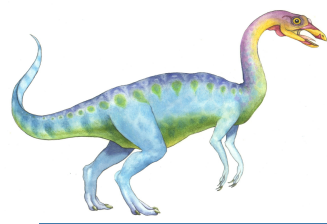
Indexed Allocation – Mapping (Cont.)





Combined Scheme: UNIX UFS (4K bytes per block, 32-bit addresses)





Performance

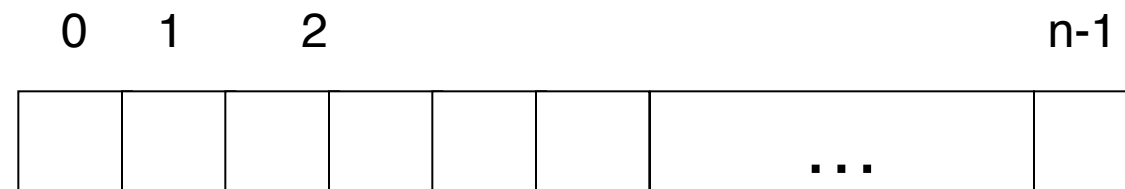
- Best method depends on file access type
 - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
 - Single block access could require 2 index block reads then data block read





Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
 - (Using term “block” for simplicity)
- **Bit vector** or **bit map** (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

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

CPUs have instructions to return offset within word of first “1” bit





Free-Space Management (Cont.)

- Bit map requires extra space

- Example:

- block size = 4KB = 2^{12} bytes

- disk size = 2^{40} bytes (1 terabyte)

- $n = 2^{40}/2^{12} = 2^{28}$ bits (or 256 MB)

- if clusters of 4 blocks -> 64MB of memory

- Easy to get contiguous files

- Linked list (free list)

- Cannot get contiguous space easily

- No waste of space

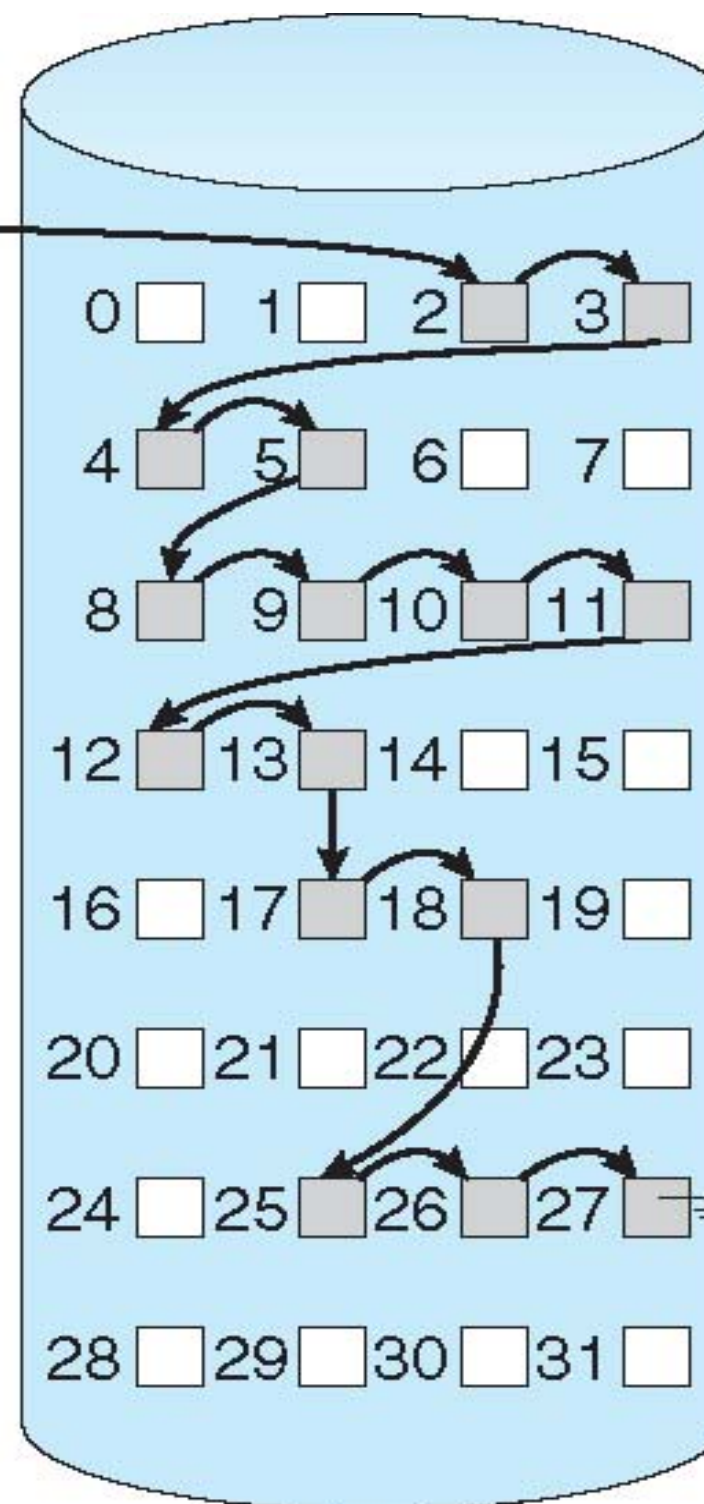
- No need to traverse the entire list (if # free blocks recorded)





Linked Free Space List on Disk

free-space list head





Free-Space Management (Cont.)

- Grouping
 - Modify linked list to store address of next $n-1$ free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
 - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
 - ▶ Keep address of first free block and count of following free blocks
 - ▶ Free space list then has entries containing addresses and counts





Efficiency and Performance

- Efficiency dependent on:
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures





Efficiency and Performance (Cont.)

- Performance
 - Keeping data and metadata close together
 - **Buffer cache** – separate section of main memory for frequently used blocks
 - **Synchronous** writes sometimes requested by apps or needed by OS
 - ▶ No buffering / caching – writes must hit disk before acknowledgement
 - ▶ **Asynchronous** writes more common, buffer-able, faster
 - **Free-behind** and **read-ahead** – techniques to optimize sequential access
 - Reads frequently slower than writes





Recovery

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





Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata 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 (sequentially)
 - Sometimes to a separate device or section of disk
 - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
 - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

