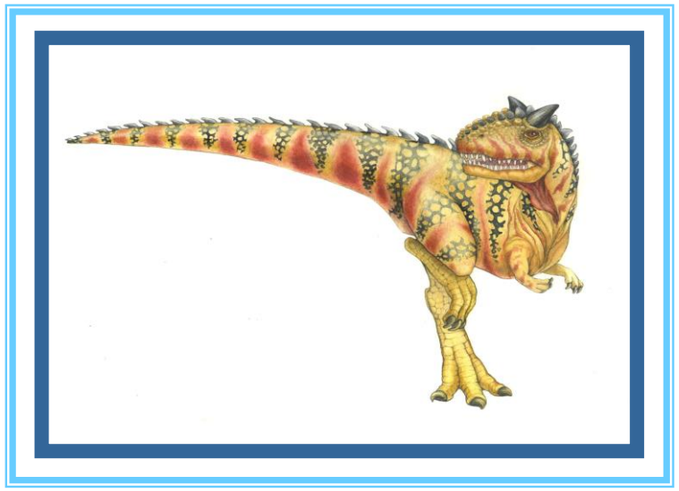
Chapter 11: Implementing File Systems



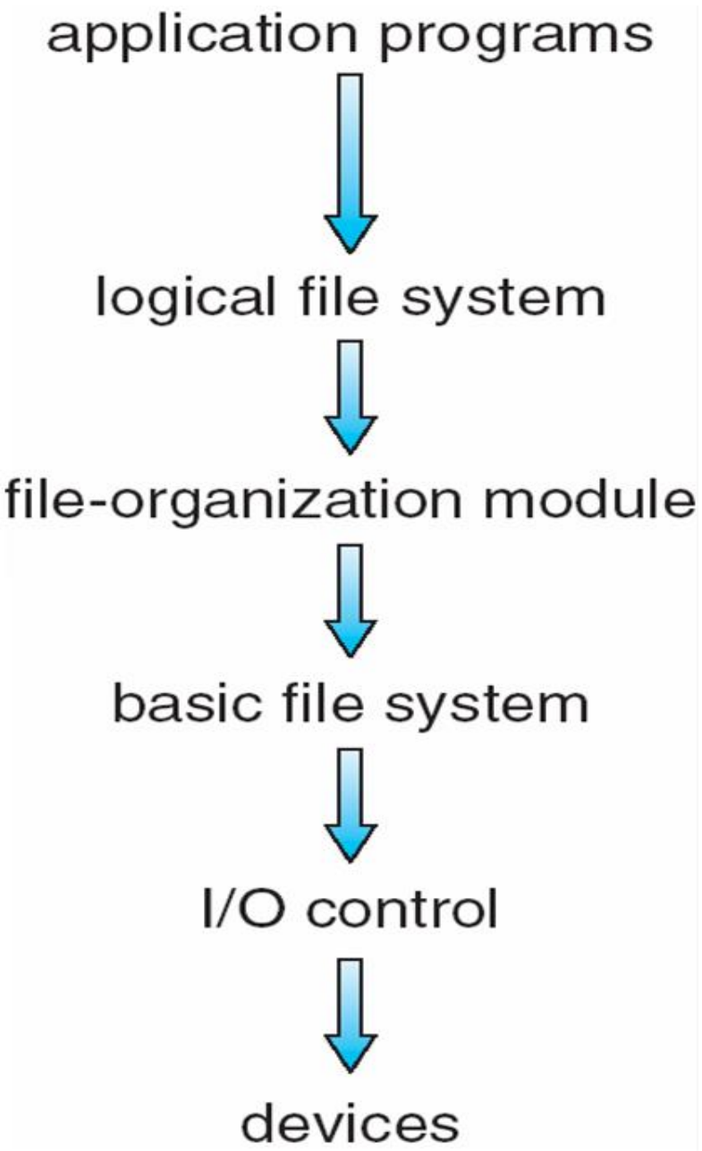
Chapter 11: Implementing File Systems

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

File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
  - **File control block** – consisting of information about a file
- **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)
- **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 hardware specific commands to I/O controller
- **Basic file system**
  - translate physical block# to driver command like “drive1, cylinder 72, …”
  - 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, physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- **Logical file system** manages metadata information
  - Translates file name into file id, file handle, location by maintaining FCB (e.g.,inode)
  - Directory management
  - Protection (e.g., ownership, permissions)

File System Layers (Cont.)

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - ***Device driver*** and ***basic file-system*** code can be used by multiple file systems
  - Each file system can have its own ***logical file system*** and ***file organization*** modules.
- Many file systems, sometimes many within an operating system
  - Each with its own format
    - CD-ROM is ISO 9660;
    - Unix has UFS (Unix File System), FFS (Fast File System);
    - 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

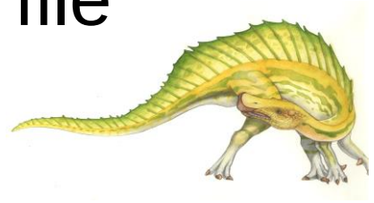
- We have system calls at the API level, but how do we implement their functions?
  - Several on-disk and in-memory structures are used
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (per volume, also called superblock) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure (per file system) organizes the files
  - File names and associated inode
- Per-file **File Control Block (FCB)** contains many details about the file
  - Inode number, permissions, size, dates

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

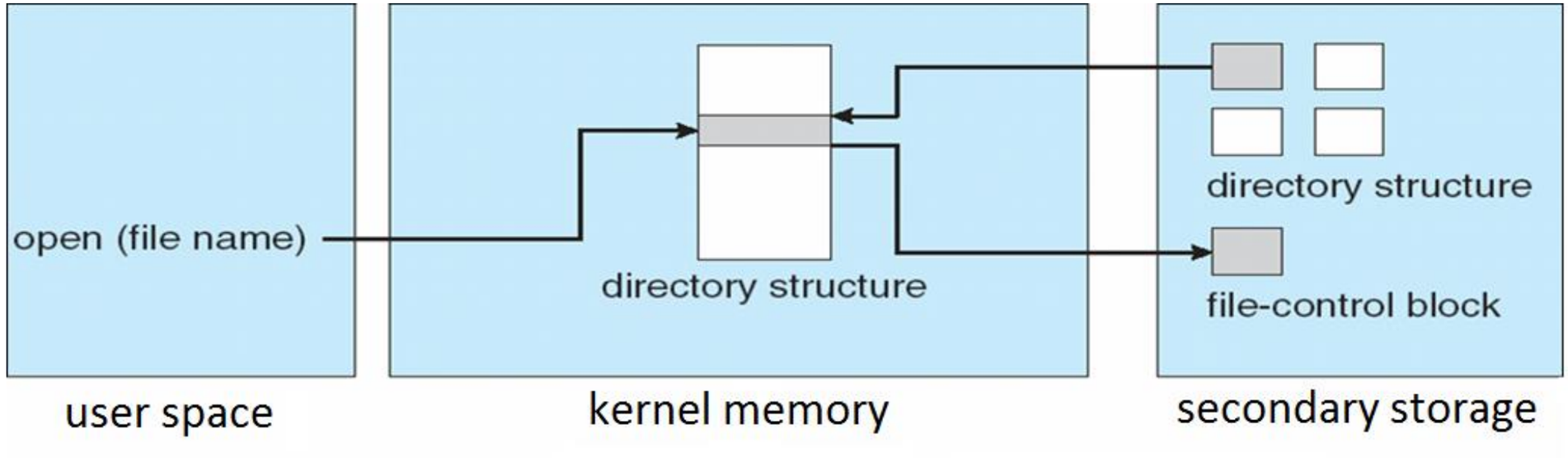
In-Memory File System Structures

- In secondary storage file-system structure:
  - **directory structure, file control block, data block**
- **In-memory file-system structure**:
  - **directory structure**, **per process open-file table**, **system wide open-file table**
- The figures in the next slide illustrate the necessary file system structures provided by OS
  - Open returns a file handle for subsequent use
  - Data from read eventually copied to specified user process memory address
- Mount table storing file system mounts, mount points, file system types

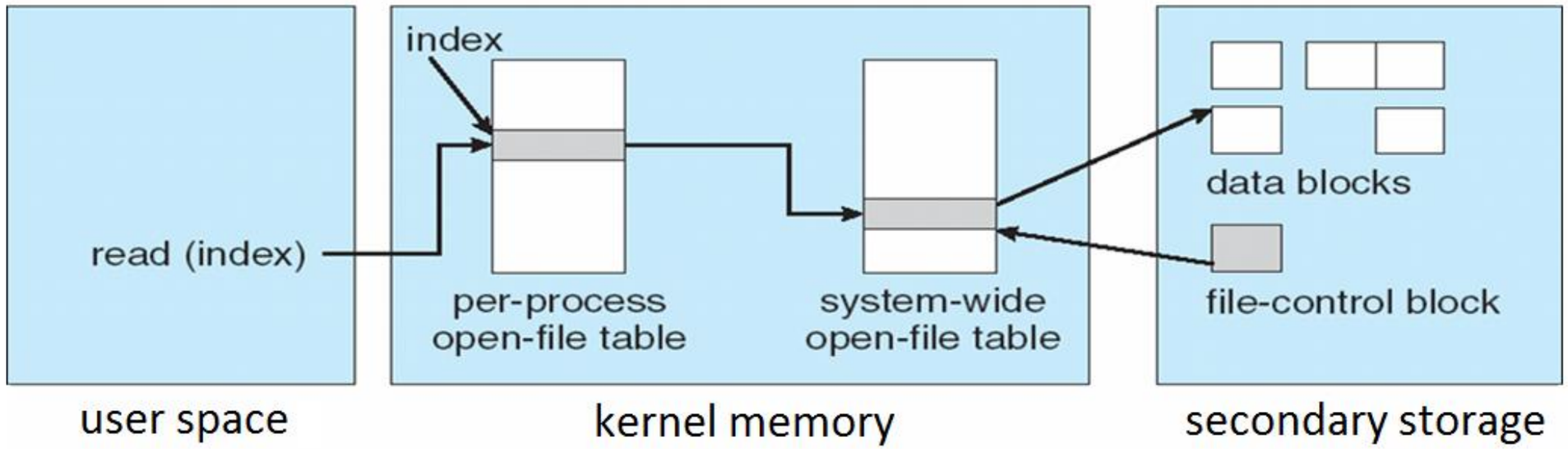


In-Memory File System Structures

File open



File read



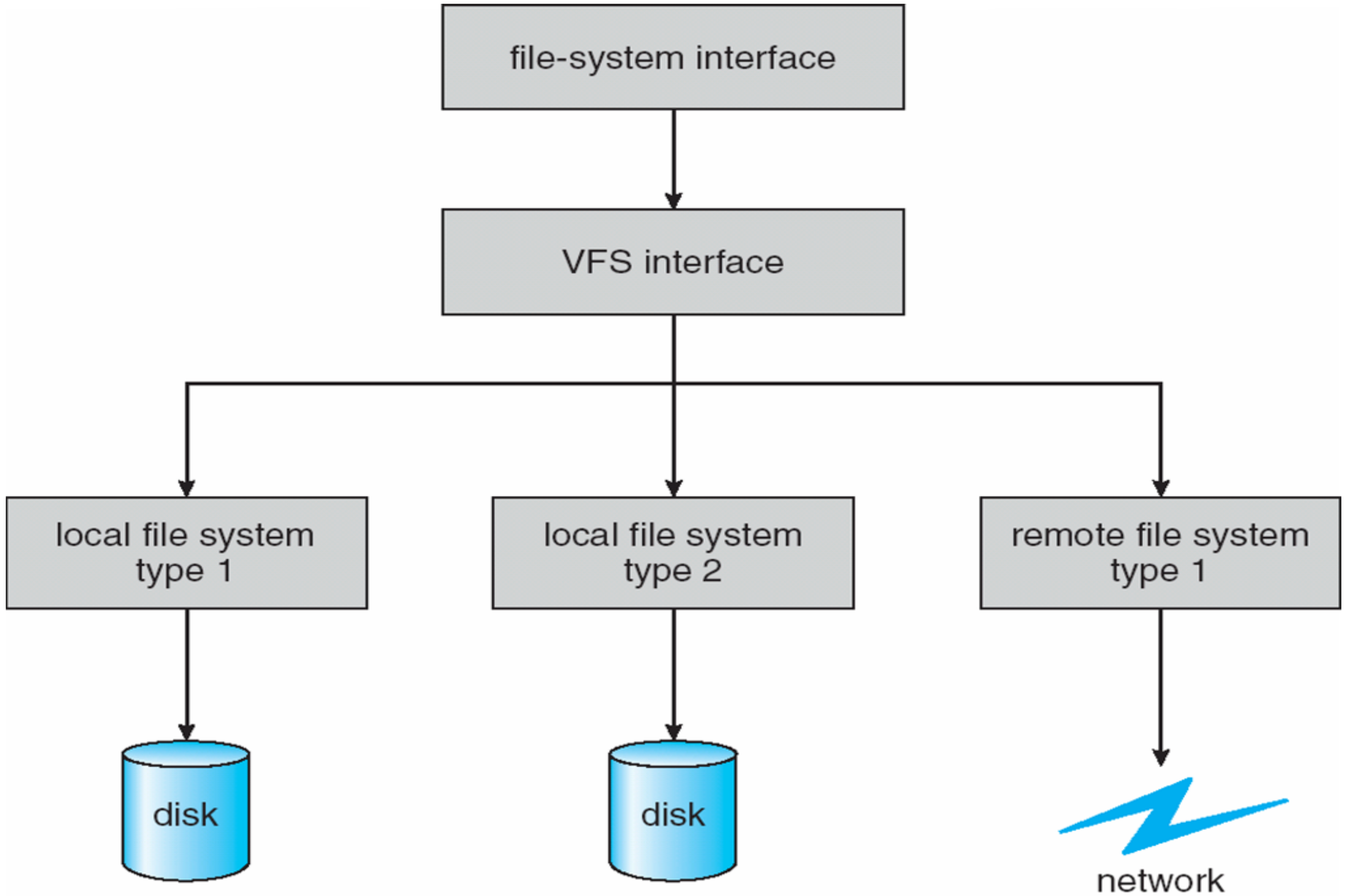
Partitions and Mounting

- A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks (the latter is a form of RAID which is discussed in Ch12)
- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

Virtual File Systems (VFS)

- VFS: provides an object-oriented way of implementing file systems
- **First (upper) layer**: general file system interface, such as open(), read(), write(), close() based on file descriptors. The API is to VFS interface, rather than specific type of file system
- **Second (middle) layer**: VFS which 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
  - It provides a mechanism for uniquely representing a file throughout a network
    - VFS Vnode: network-wide unique
    - UNIX inode: unique within a single file system
  - It distinguishes local files from remote ones and activates file-system-specific operations to handle local requests and calls the NFS protocol procedures for remote requests.
- **Third layer**: implements the file-system type or remote-file-system protocol.

Schematic View of Virtual File System



Virtual File System Implementation

- For example, Linux has four object types:
  - **inode**: an individual file
  - **file**: an open file
  - **superblock**: an entire file system
  - **dentry**: an individual directory entry

- For each of the four object types, VFS defines a set of operations that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of actual functions that implement the defined operations for that particular object
    - E.g., functions for **file** object includes: open(), close(), read(), write(), mmap()

Chapter 11: Implementing File Systems

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

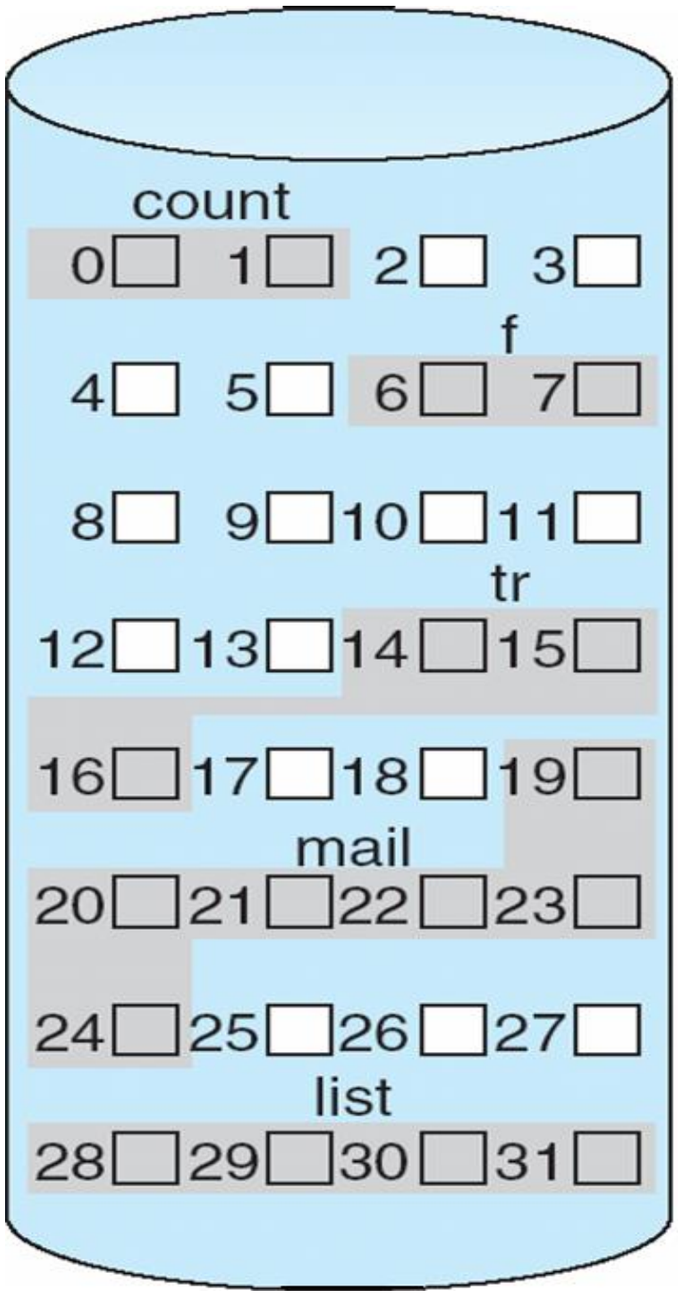
Directory Implementation

- **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 a balanced 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
    - Chained-overflow hash table: each hash entry is a linked list instead of an individual value ➔ search might require stepping through a linked list of colliding table entries. ➔ But still much faster than a linear search through the entire directory.

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 (disk seek time is minimal in general)
  - Simple – only starting location (block #) and length (# of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line** (**downtime**) or **on-line**
    - finding space: first fit, best fit, worse fit, …
    - unknown file size: file cannot grow, or move the file to new space
- **Extent-based system**: a modified contiguous allocation scheme
  - A contiguous space is allocated initially. If not enough, **extent** is added.
  - **Extent:** a contiguous block of disks. A file may consist of one or more extents
    - Internal fragmentation: if the extents are too large
    - External fragmentation: extents of varying sizes are allocated and deallocated.

Contiguous Allocation of Disk Space



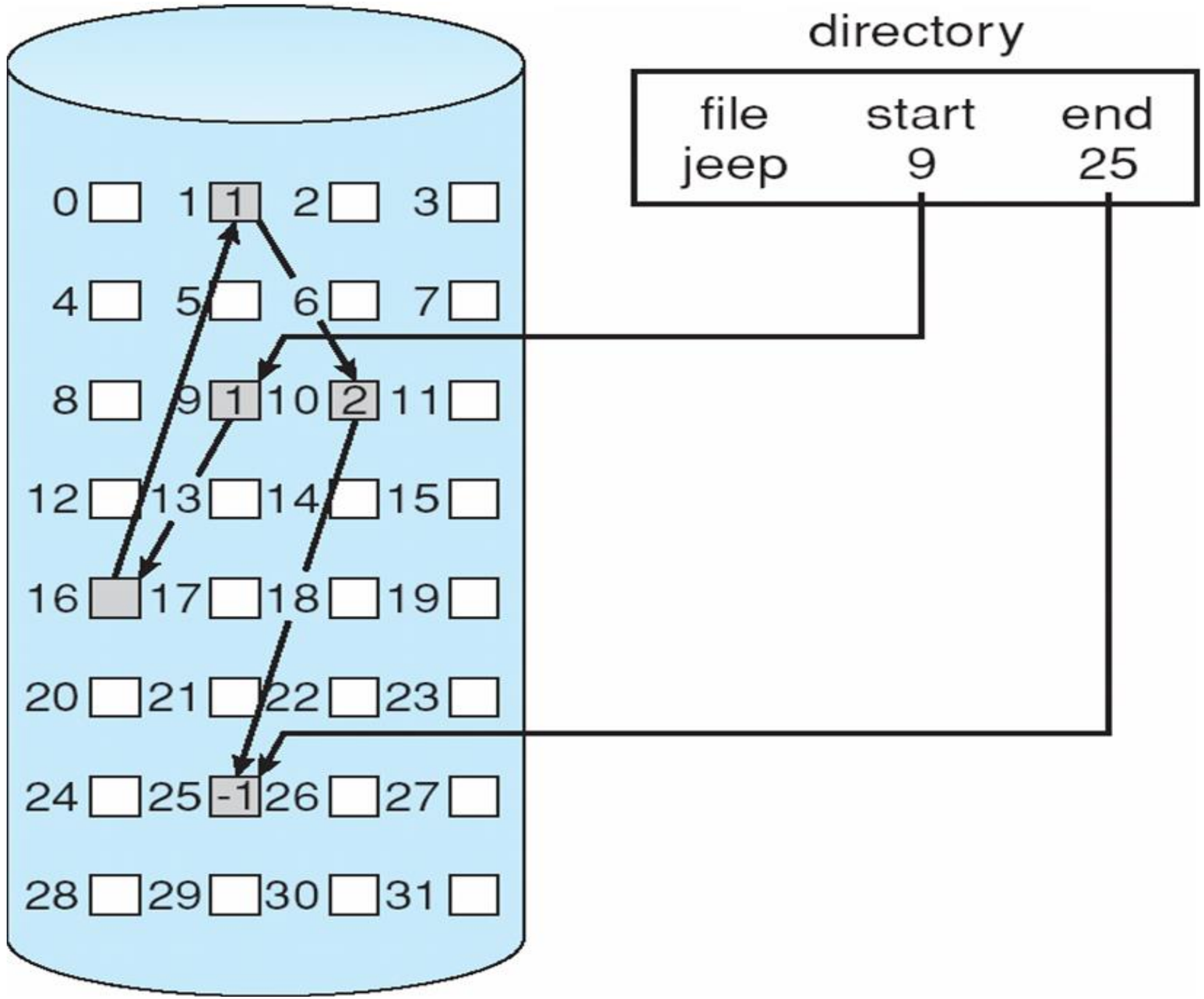
directory

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

Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks scattered anywhere on the disk
  - Directory entry has a pointer to the first block of the file
  - Each block contains pointer to next block, the file ends at nil pointer
  - No compaction, external fragmentation
  - File size can be unknown when it is created, the file can continue to grow
  - Free space management system called when new block needed
  - Space required to store the pointers - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem – pointer lost or damaged
  - Locating a block can take many I/Os and disk seeks – finding the i-th block must start from the beginning of that file and follow the pointers until the i-th block.

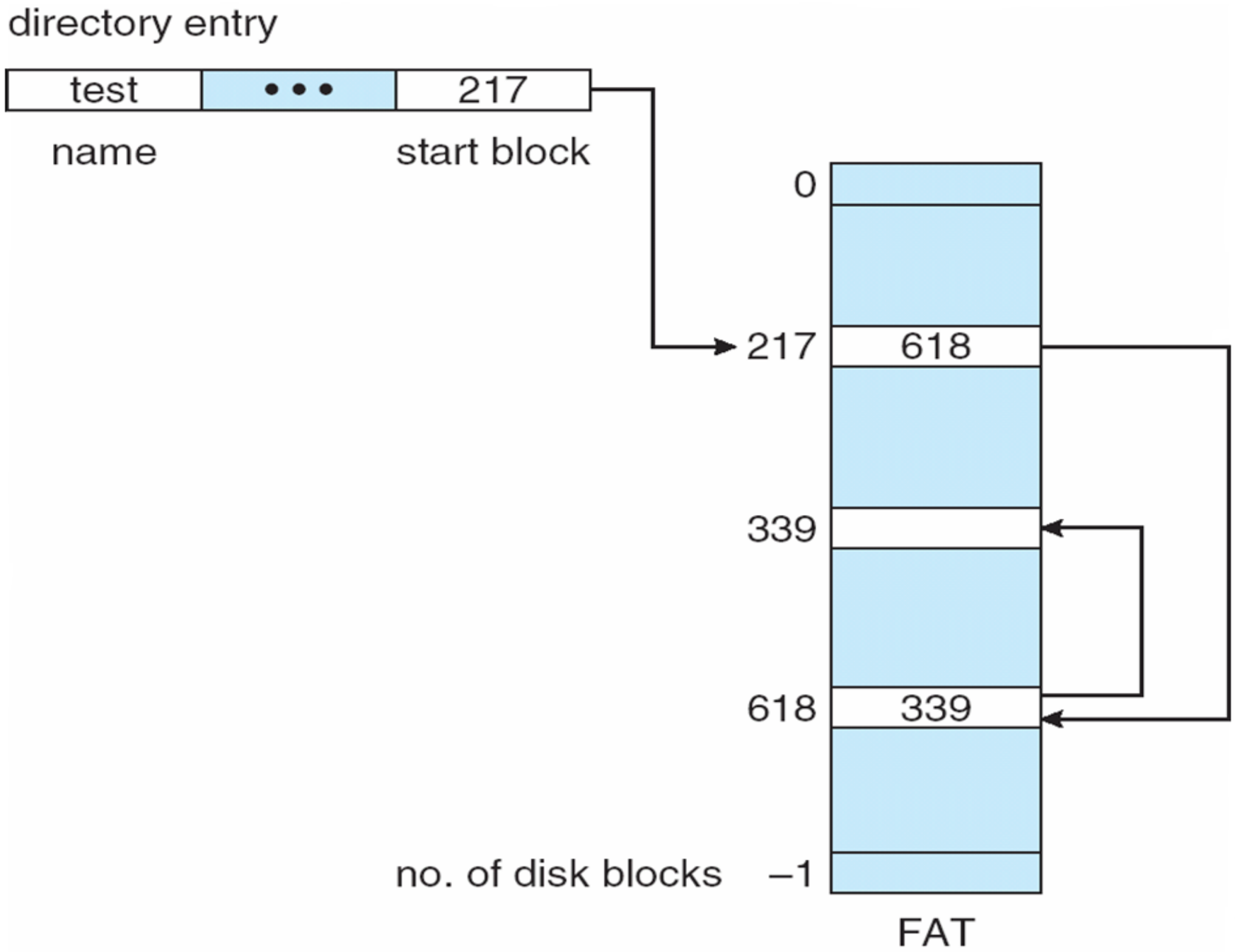
Linked Allocation



Allocation Methods - Linked

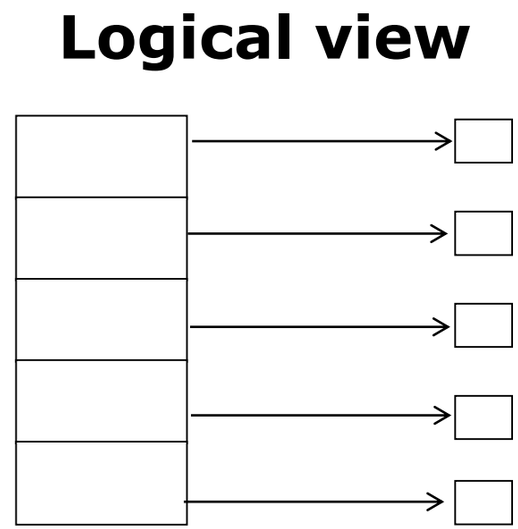
- **FAT (File Allocation Table) variation** – a variation of linked allocation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple
- Example (see next slide)
  - Directory entry contains the block number of the first block of the file
  - Table entry indexed by that block number contains the block number of the next block in the file. This chain continues till the last.
  - The last block of the file is indicated by a special end-of-file value as the table entry.
  - An unused block is indicated by a value of 0.
  - Allocating a new block is simply by finding the first 0-valued table entry.

File-Allocation Table

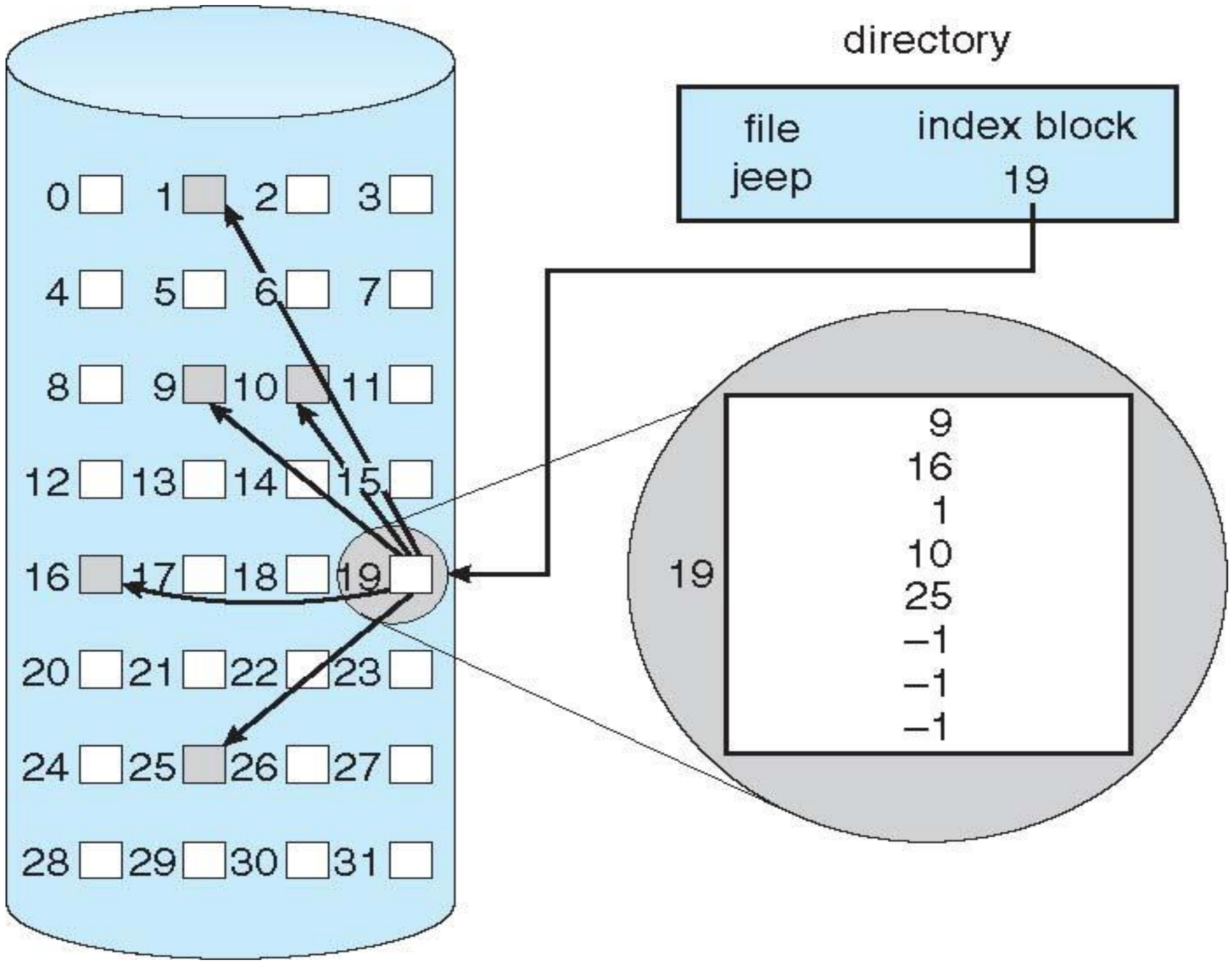


Allocation Methods - Indexed

- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks
  - The i-th entry in the index block points to the i-th block of the file.
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
  - Space overhead of index block is greater than the overhead in linked allocation because an entire index block must be allocated, even if only one or two pointers are non-null.
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table



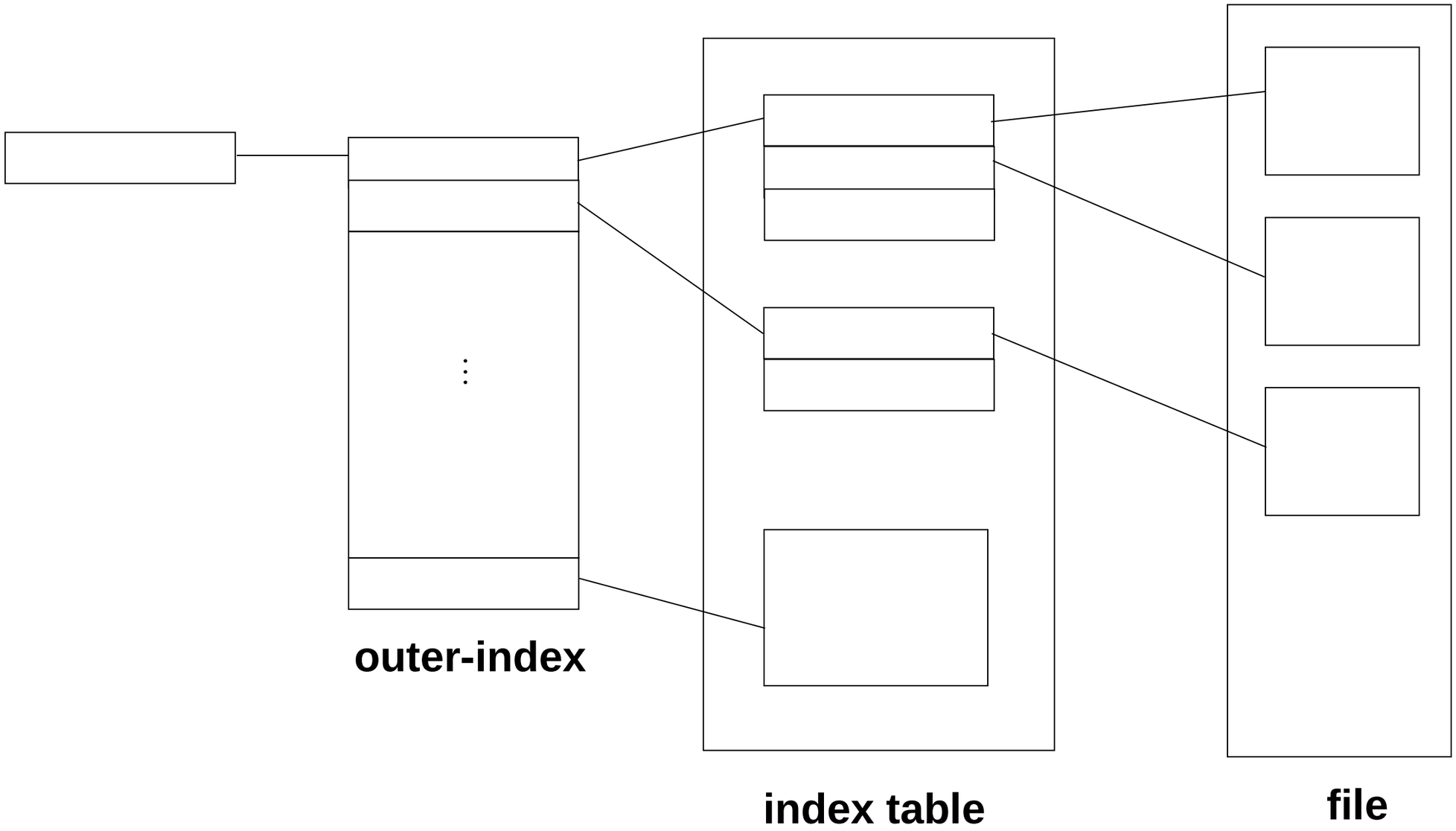
Example of Indexed Allocation



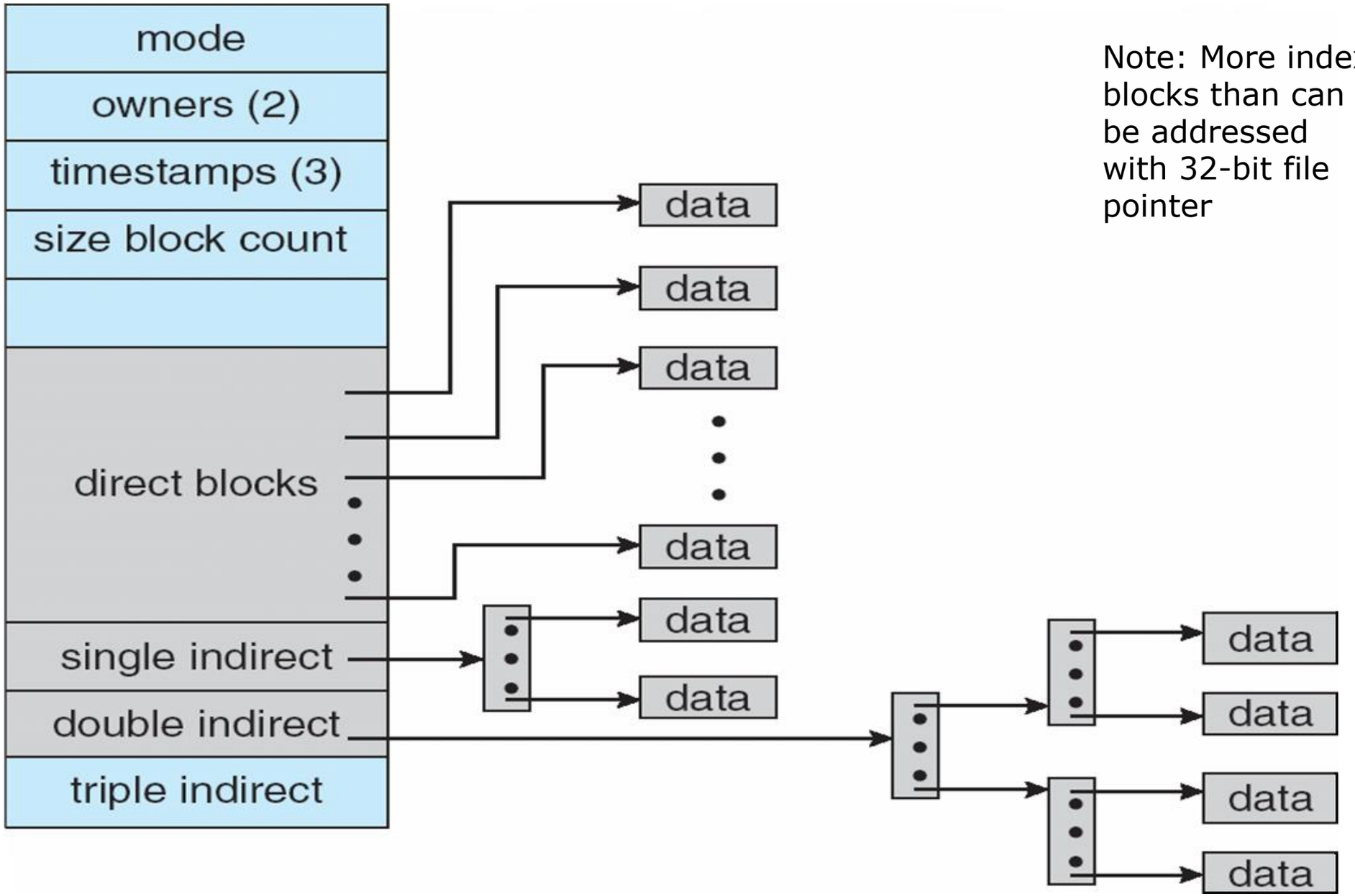
Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- **Linked scheme** – Link together several index blocks (no limit on file size)
- **Multilevel index scheme**
  - A first-level index block points to a set of second-level index blocks, which in turn point to the file blocks. ➔ can be extended to a third or fourth level.
  - e.g., Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)
- **Combined scheme**
  - The index block contain pointers point to
    - direct blocks, indirect blocks, double indirect block, or triple indirect block
  - Used in UNIX-based file systems
  - Index blocks can be cached in memory, but data blocks may be spread all over a volume.

Indexed Allocation – Mapping (Cont.)



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



Note: More index blocks than can be addressed with 32-bit file pointer

Performance

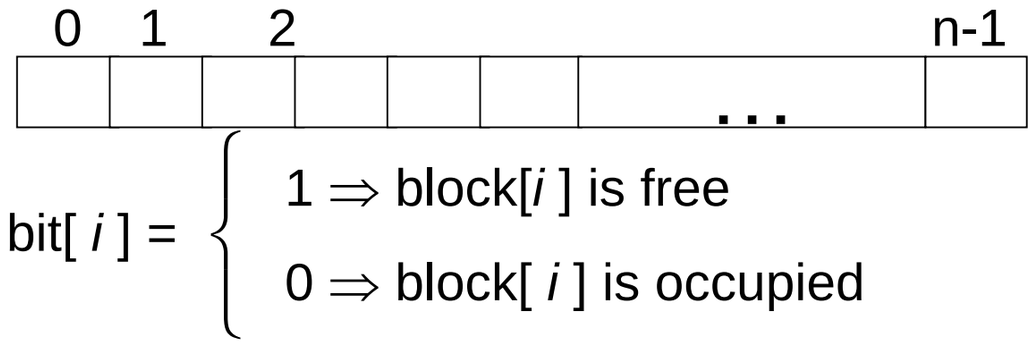
- Best method depends on file access type
  - Contiguous great for sequential and random (or say direct access)
  - Linked good for sequential, not random
  - Indexed more complex: single block access could require index block read then data block read
- Declare access type (sequential or random) at creation ➔ select either contiguous or linked
- Some system combine contiguous with indexed allocation ➔ contiguous allocation for small files and switch to an indexed allocation if the file grows large
- Adding instructions to the execution path to save one disk I/O is reasonable
  - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
  - Typical disk drive at 250 I/Os per second
    - 159,000 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Chapter 11: Implementing File Systems

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

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) ➔



  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

- Bit map requires extra space

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

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

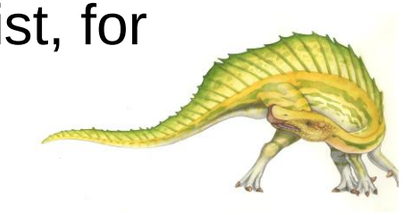
  $n = 2^{40}/2^{12} = 2^{28}$ bits (or 256 MB)

  } if clusters of 4 blocks ➔ 64MB of memory

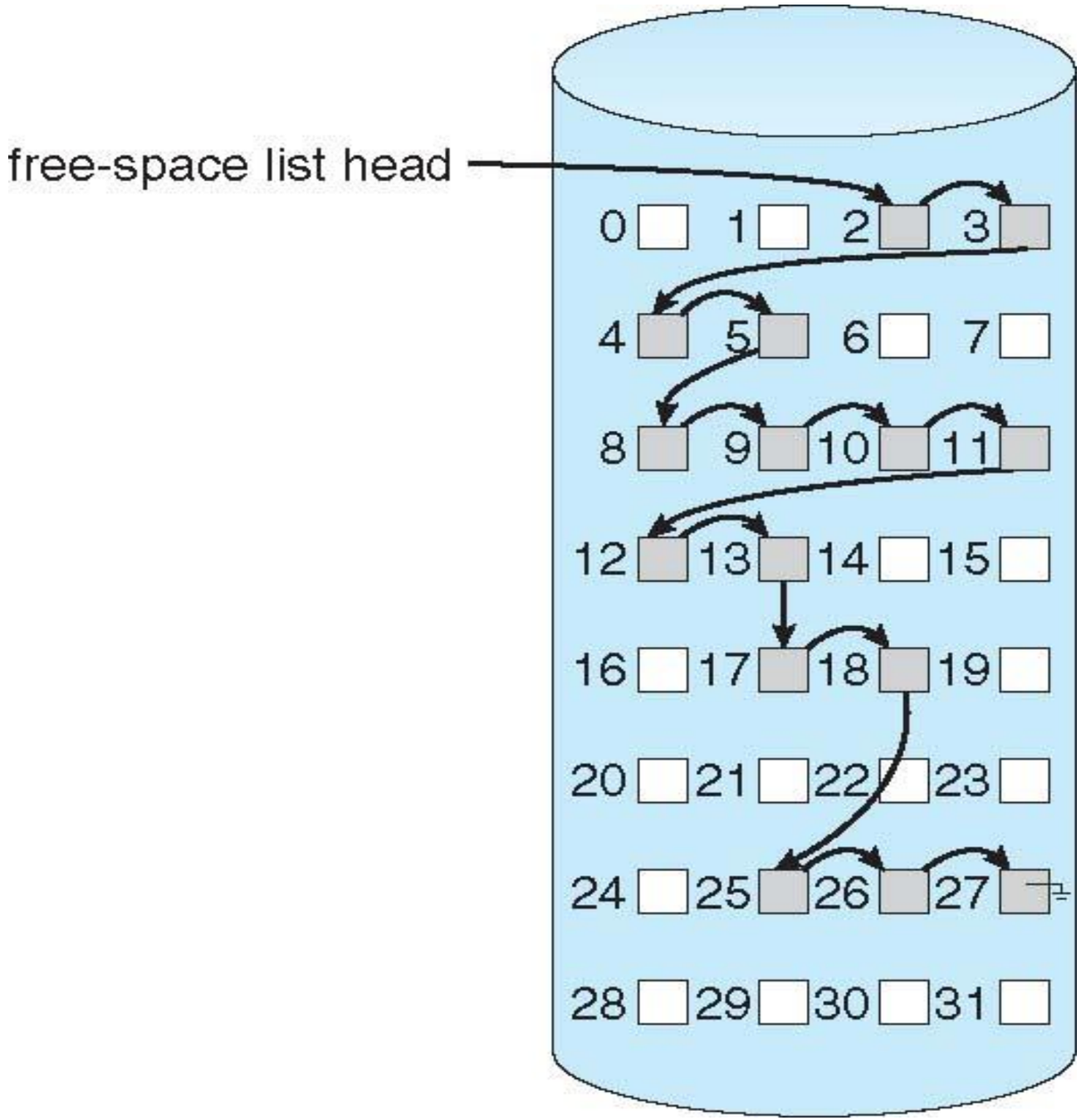
- Easy to get contiguous files

Free-Space Management (Cont.)

- **Linked list (free list)**
  - Keep a pointer to 1st free block which contains a pointer to the next one, and so on.
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list
- **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 the addresses of another n free blocks, and so on.
  - A large number of free blocks can be found quickly.
- **Counting**
  - Because space is frequently contiguously used and freed (e.g., 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
    - The entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.



Linked Free Space List on Disk



Free-Space Management (Cont.)

- **Space Maps**
  - Used in ZFS
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - Given volume can contain hundreds of metaslabs
  - Each metaslab has associated space map
    - Uses counting algorithm and stores to log file rather than file system
    - The space map is a log of all block activity (allocating and freeing), in time order, in count formating.
  - To allocate or free space from a metaslab ➔ load the associated space map into memory in balanced-tree structure, indexed by offset
    - Replay log into that structure
    - Combine contiguous free blocks into single entry
    - Finally, the free-space list is updated on disk.

Chapter 11: Implementing File Systems

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

Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
    - In UNIX, inodes are pre-allocated on a volume. Even an empty disk has a percentage of its space lost due to inodes.➔ but it improves performance.
    - Cluster scheme suffers from internal fragmentation (poor space efficiency) ➔ but it improves file-seek and file-transfer performance.
  - Types of data kept in file's directory entry
    - last access date ➔ whenever the file is read, the directory entry must be updated ➔ performance cost is high
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
    - old Solaris: fixed-length process table (or open-file table), allocated at system startup. ➔ when table full, no more processes (or files) could be created.
    - new Solaris, they are allocated dynamically ➔ Manipulate these tables become complicated and slower

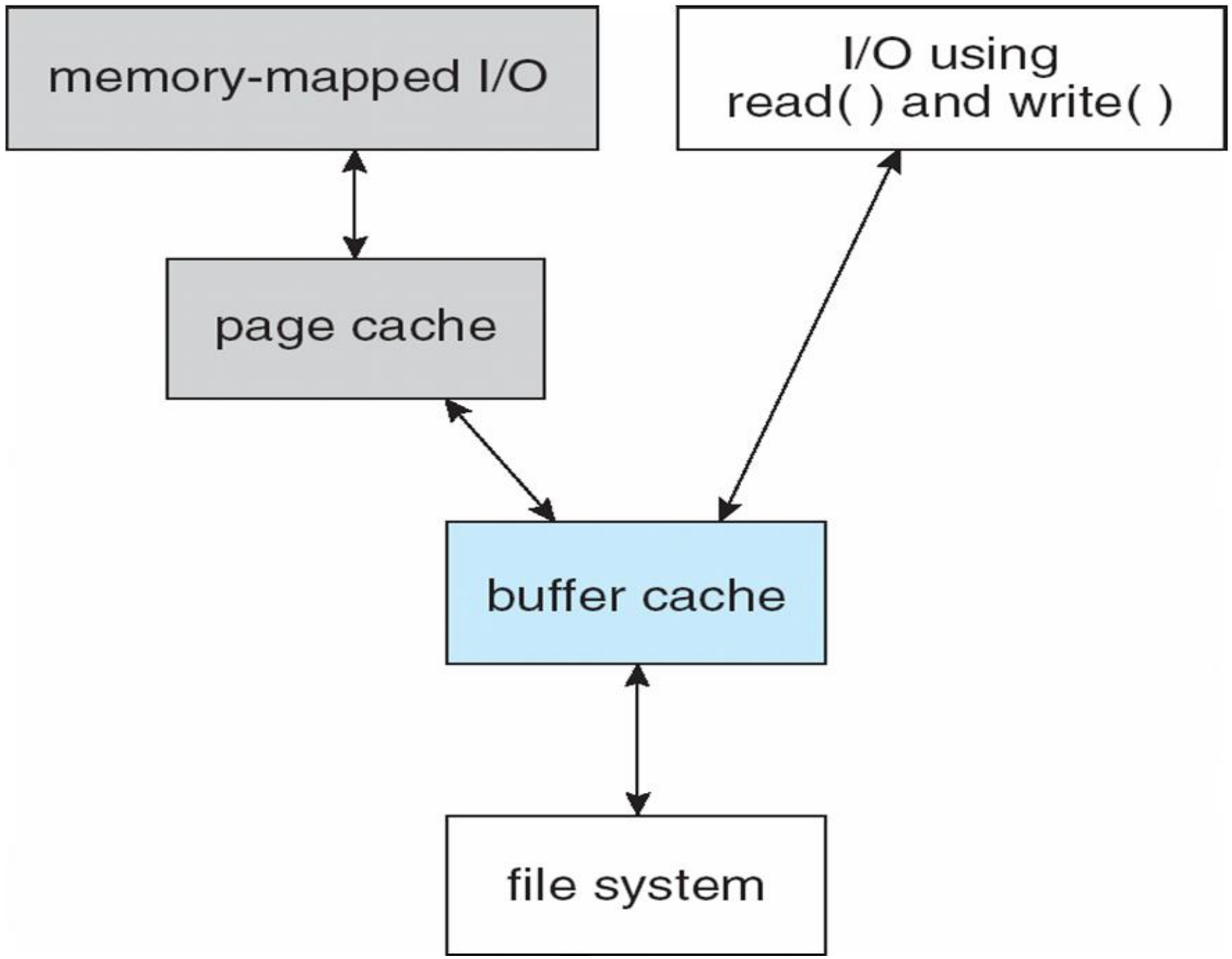
Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together – In UNIX, it keeps a file's data blocks near that file's inode block to reduce seek time.
  - **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

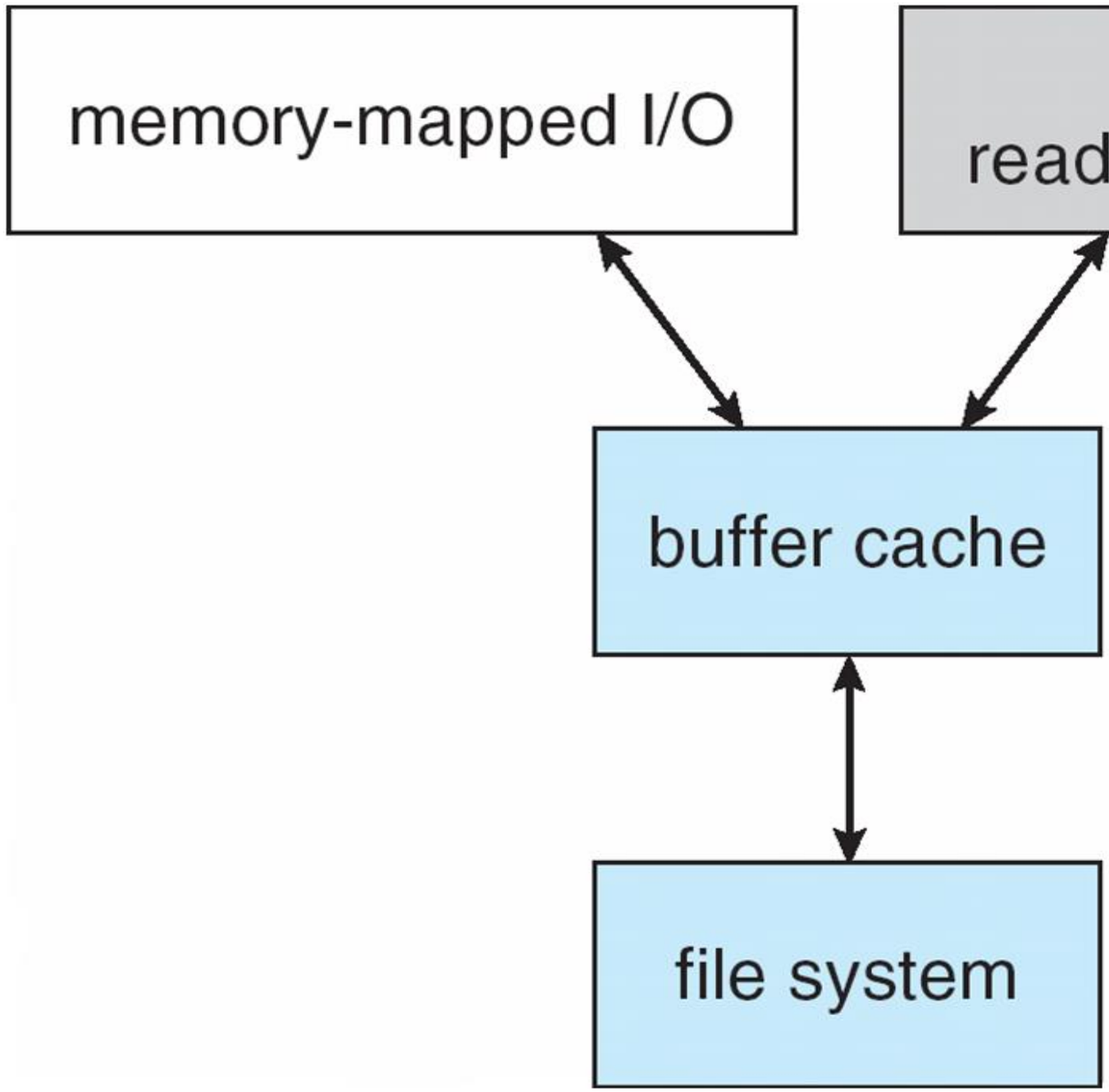
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
  - The page cache uses virtual memory techniques to cache file data as pages rather than as file-system-oriented disk blocks.
  - Access interface with virtual memory is far more efficient than interface with file system
  - Solaris, Linux, Windows – use page caching to cache both process pages and file data. (known as unified virtual memory)
- 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



I/O Using a Unified Buffer Cache



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

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
- Transactions in the log are asynchronously written to 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