CS3600 — Systems and Networks

NORTHEASTERN UNIVERSITY

Lecture 11: File System Implementation

Prof. Alan Mislove (amislove@ccs.neu.edu)

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

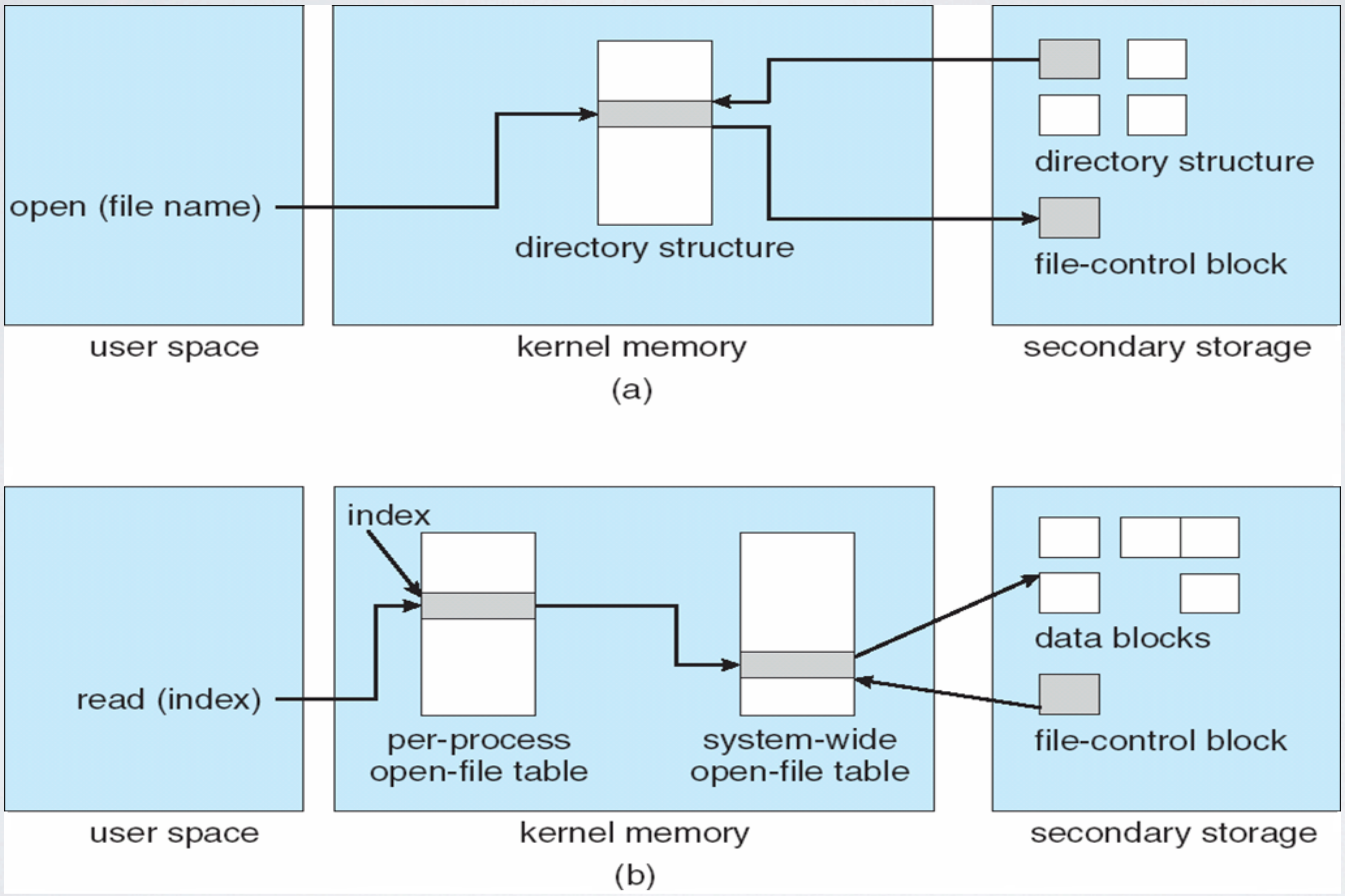
File-System Implementation

- 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



Partitions and Mounting

- 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

Directory Implementation

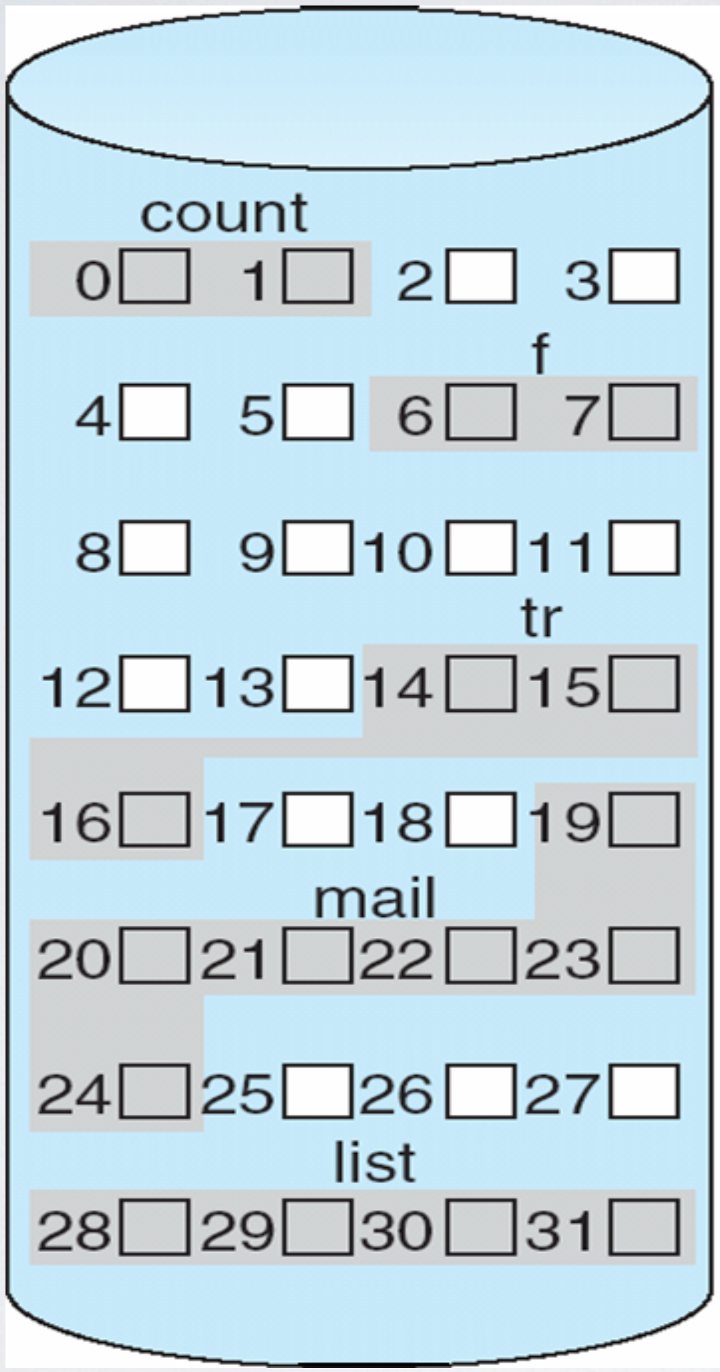
- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **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

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

- **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 of Disk Space



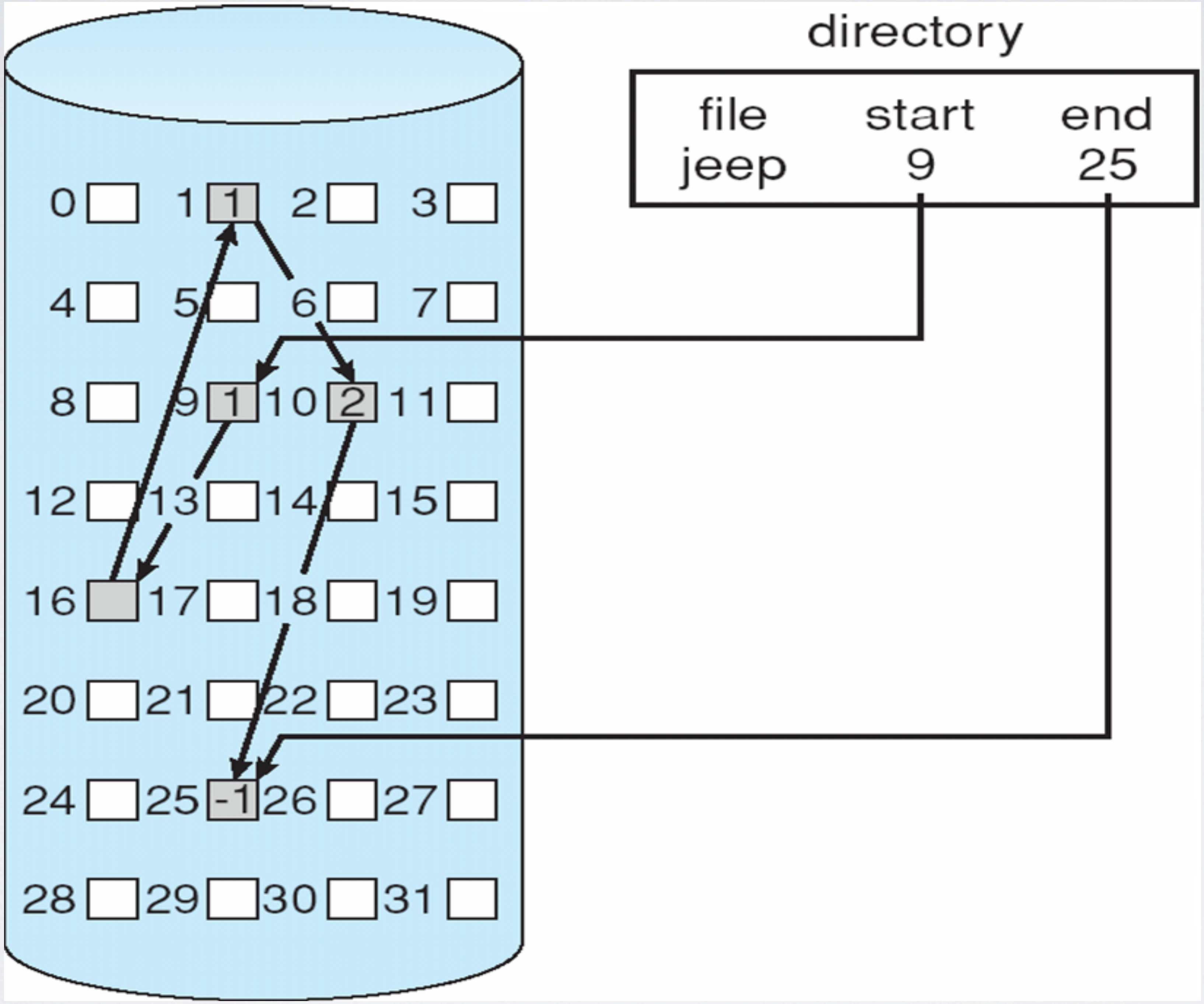
directory

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

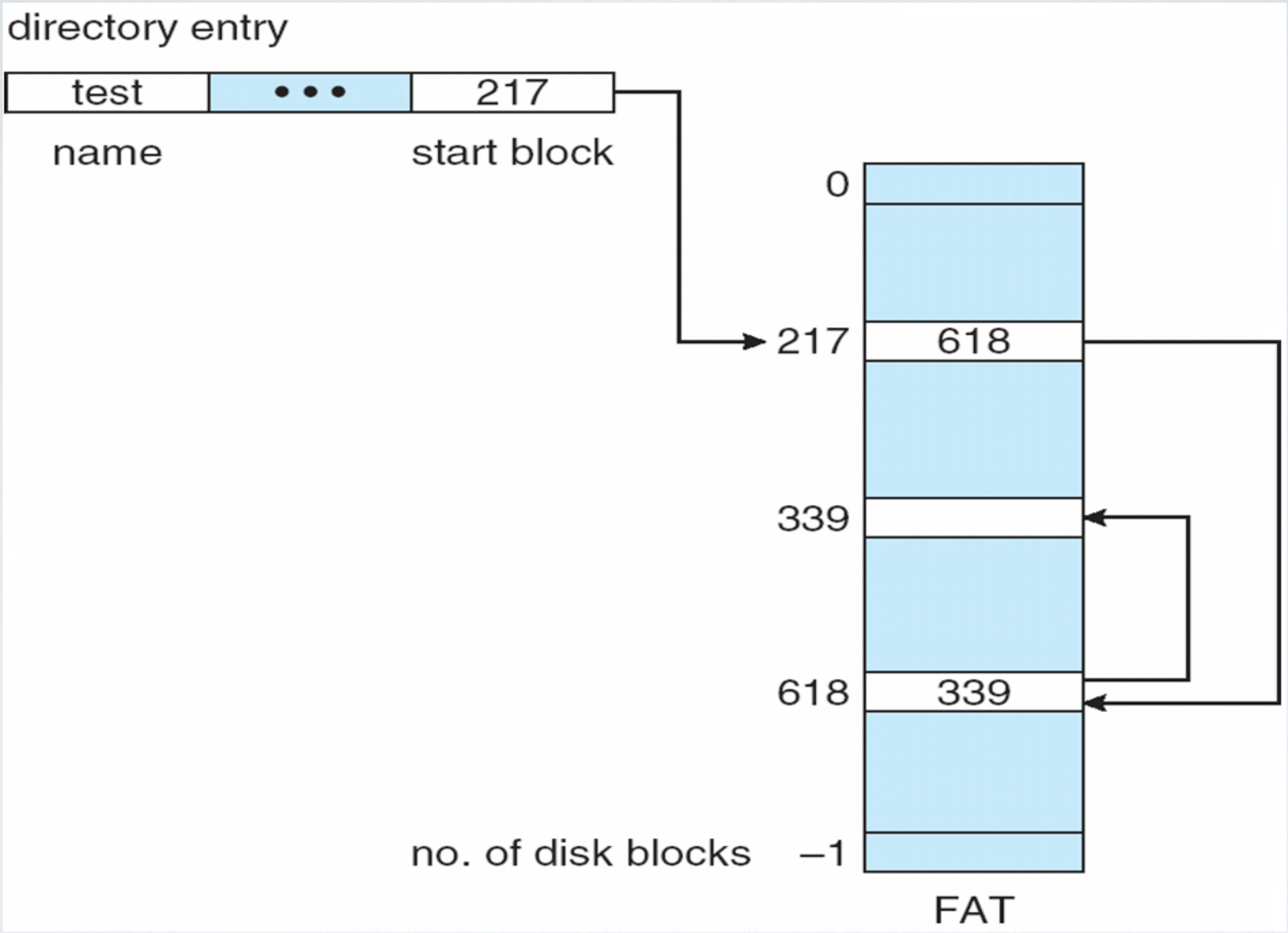
Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - 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



File-Allocation Table

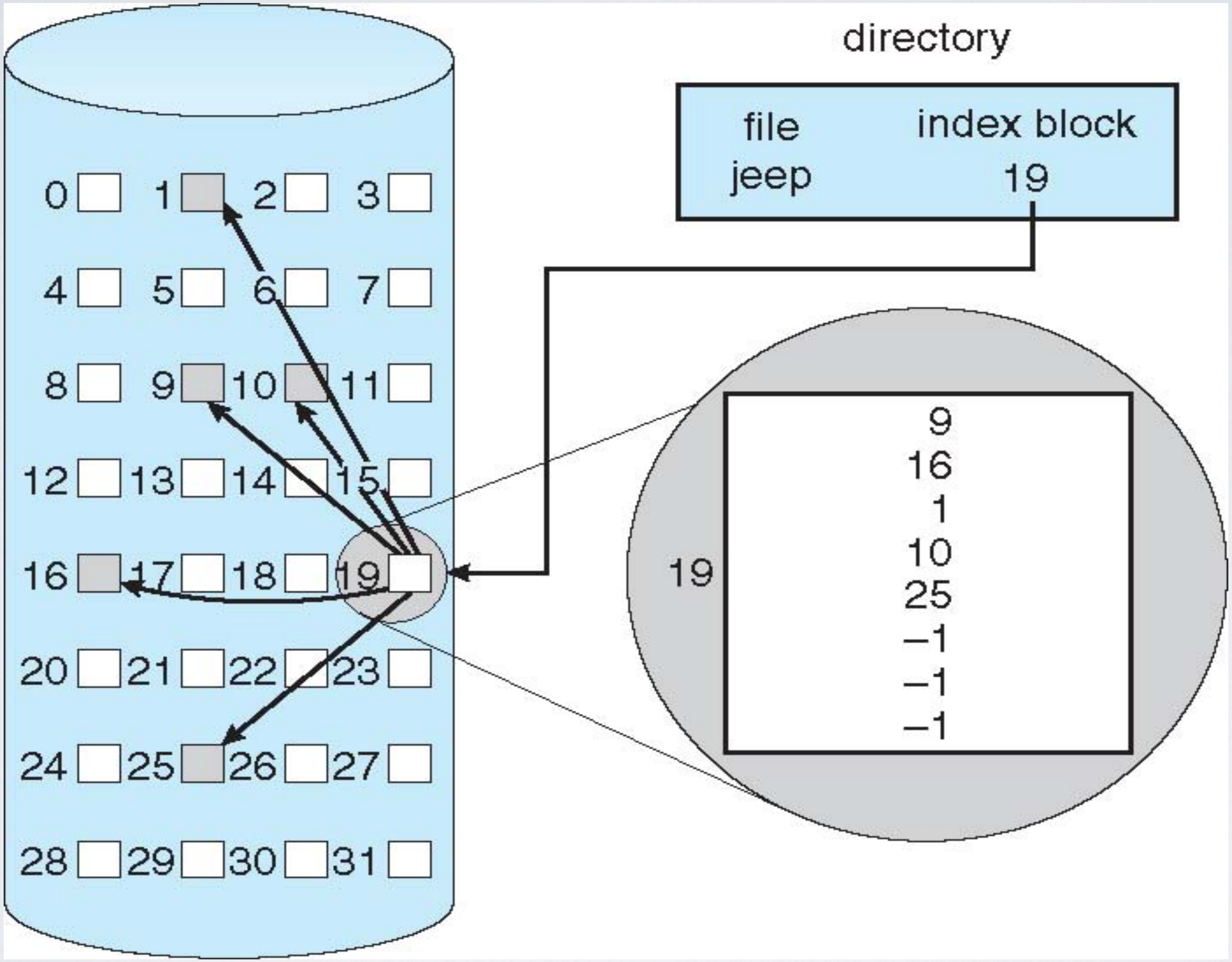


Allocation Methods - Indexed

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

- 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 bytes and block size of 512 bytes. We need only 1 block for index table

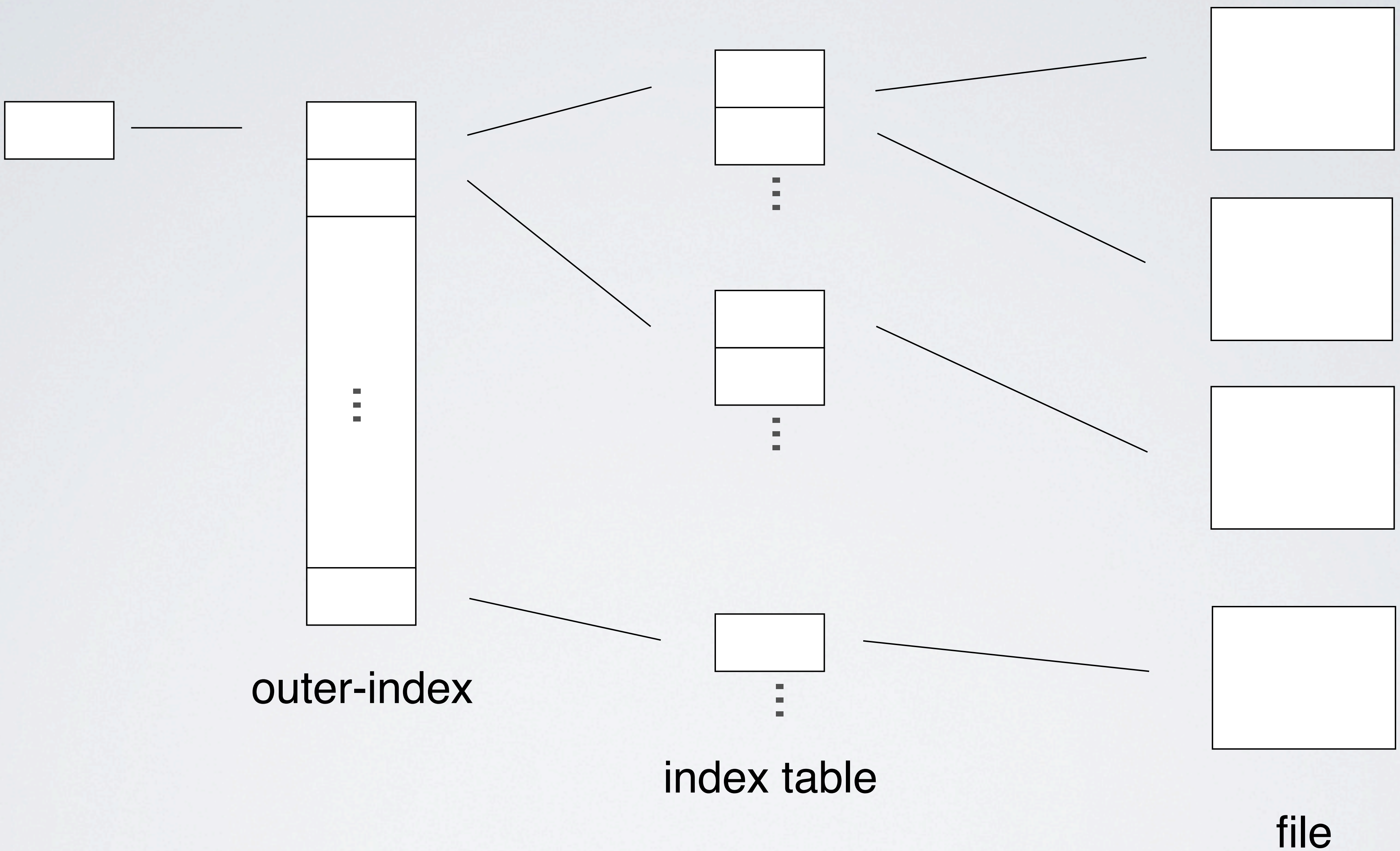
Example of Indexed Allocation



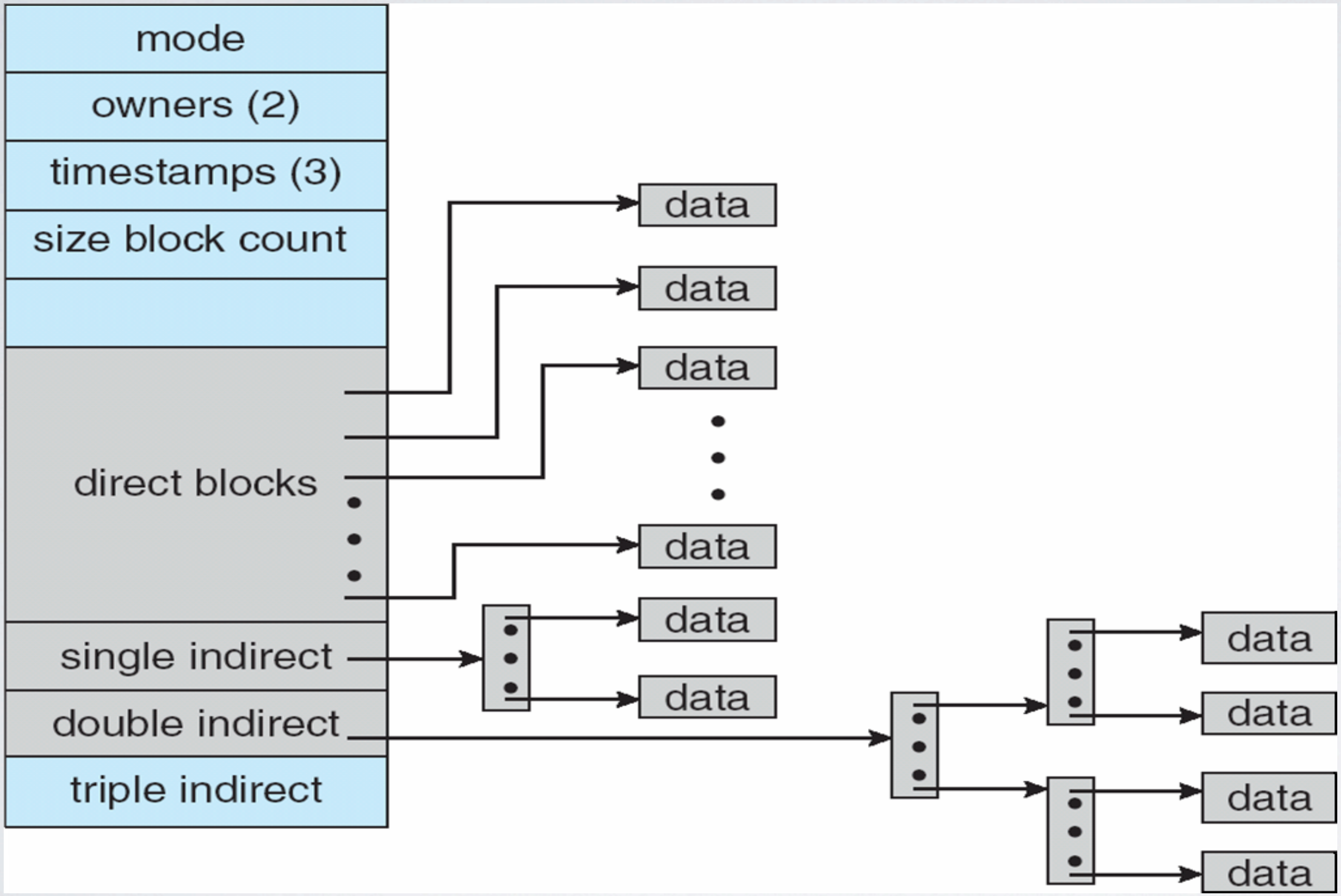
Indexed Allocation – Mapping (Cont.)

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

Indexed Allocation – Two-level index



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
- 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
  - Clustering can help improve throughput, reduce CPU overhead

Performance (Cont.)

- 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
    - http://en.wikipedia.org/wiki/Instructions_per_second
  - 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 million instructions during one disk I/O

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)

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

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

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

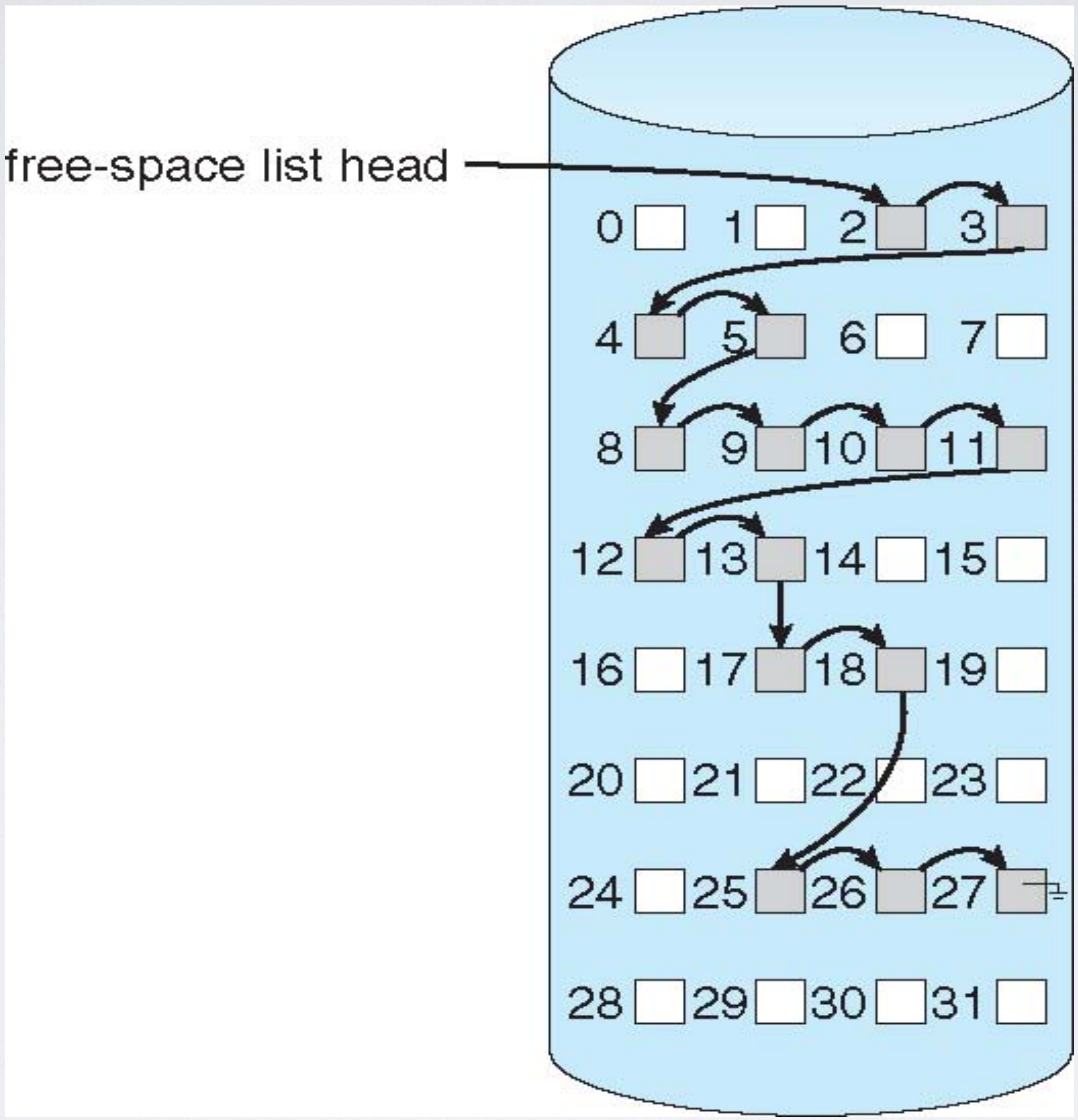
    disk size = 2^{40} bytes (1 terabyte)

    $n = 2^{40}/2^{12} = 2^{28}$ 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



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