Chapter 12: File System Implementation

Chapter 12: File System Implementation

- File-System Structure
- 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 discuss block allocation and free-block algorithms and trade-offs

Basics: 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, and retrieved easily
- Disk provides in-place rewrite and random access
  - It is possible to read a block from the disk, modify the block, and write it back into the same place.
  - I/O transfers performed in **blocks** of 1 or more **sectors**
    - A sector is usually 512 bytes
- **File control block (FCB)** – storage structure consisting of information about a file

Directory Implementation

- The choice of the directory implementation is crucial for the efficiency, performance, and reliability of the file system.
- **Linear list** of file names with pointer to the data blocks
  - **Pros:** Simple to program
  - **Cons:** Time-consuming to execute -- Linear search time
  - **Solutions:**
    - Keep sorted + binary search
    - Use indexes for search, such as B+ tree
- **Hash Table** – linear list with hash data structure
  - Hash on file name
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Each hash entry can be a linked list - resolve collisions by adding new entry to linked list.

Allocation of Disk Space

- Low level access methods depend upon the disk allocation scheme used to store file data
  - Contiguous Allocation
  - Linked List Allocation
  - Indexed Allocation

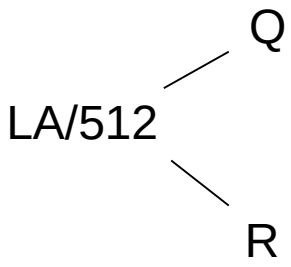
Allocation Methods - Contiguous

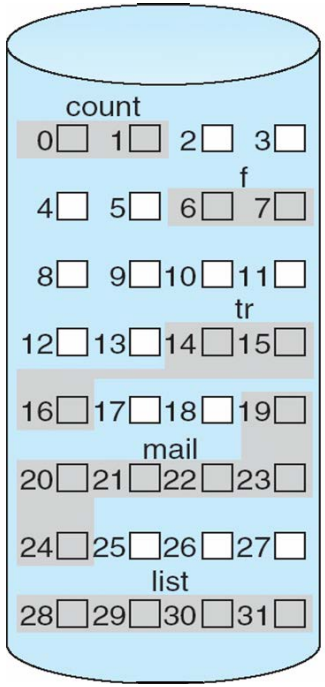
- An allocation method refers to how disk blocks are allocated for files

- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
    - Commonly, hardware is optimized for sequential access
    - For a magnetic disk – reduces seek time, head movement
  - Simple – only info required:
    - starting location (block #) and
    - length (number of blocks)
  - Problems include
    - finding space for file,
    - knowing file size,
    - external fragmentation,
    - need for **compaction off-line** (**downtime**) or **on-line**
      - Can be costly

Contiguous Allocation

- For simplicity, assume 1 block = 1 sector
- Mapping from logical to physical - <Q, R>





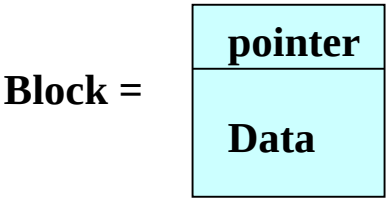
directory

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

Block to be accessed = Q + starting address
Displacement into block = R

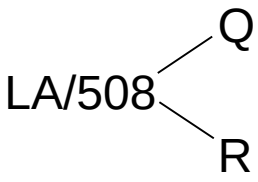
Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
- Blocks may be scattered anywhere on the disk.
- Each block contains **pointer** to the next block
  - Disk space is used to store pointers,
    - if disk block is 512 bytes, and pointer (disk address) requires 4 bytes, user sees 508 bytes of data.
  - Pointers in list not accessible
  - File ends at nil pointer
- **Pros:** No external fragmentation
  - No compaction
- **Cons:** Locating a block can take many I/Os and disk seeks



Linked Allocation

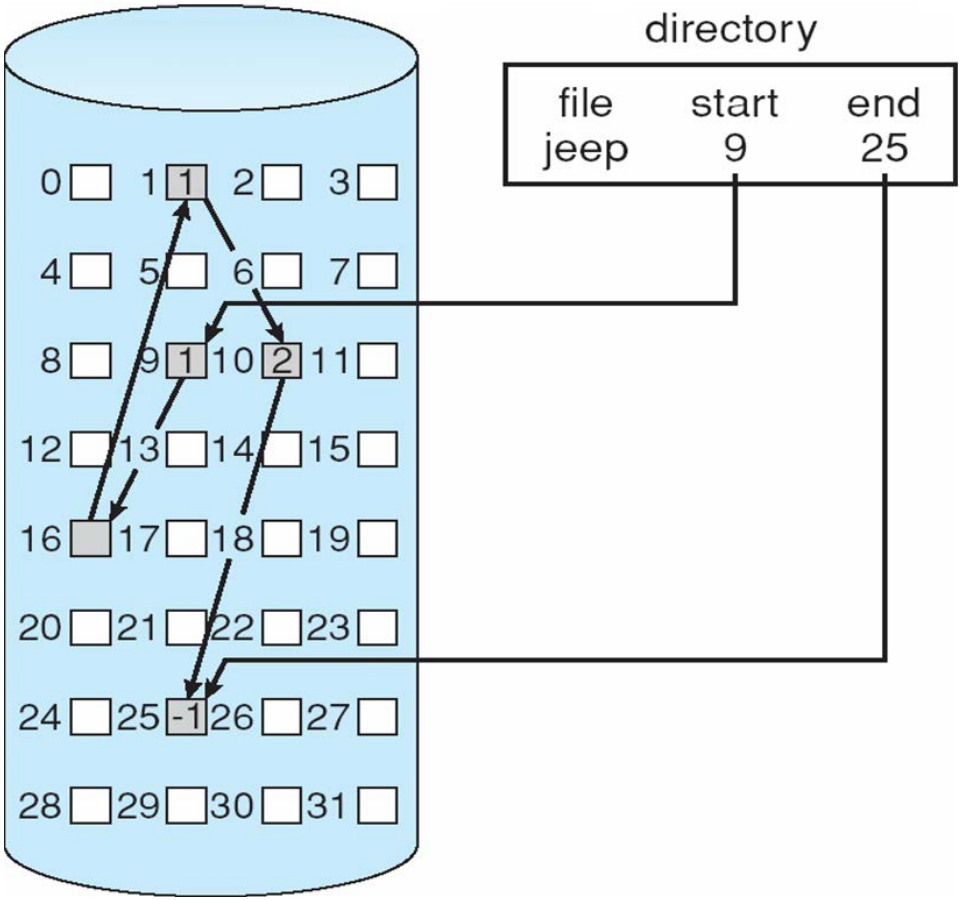
- Mapping from logical address o physical



Block to be accessed is the Q-*th* block in the linked chain of blocks representing the file.

Displacement into block = R + 4

Linked Allocation

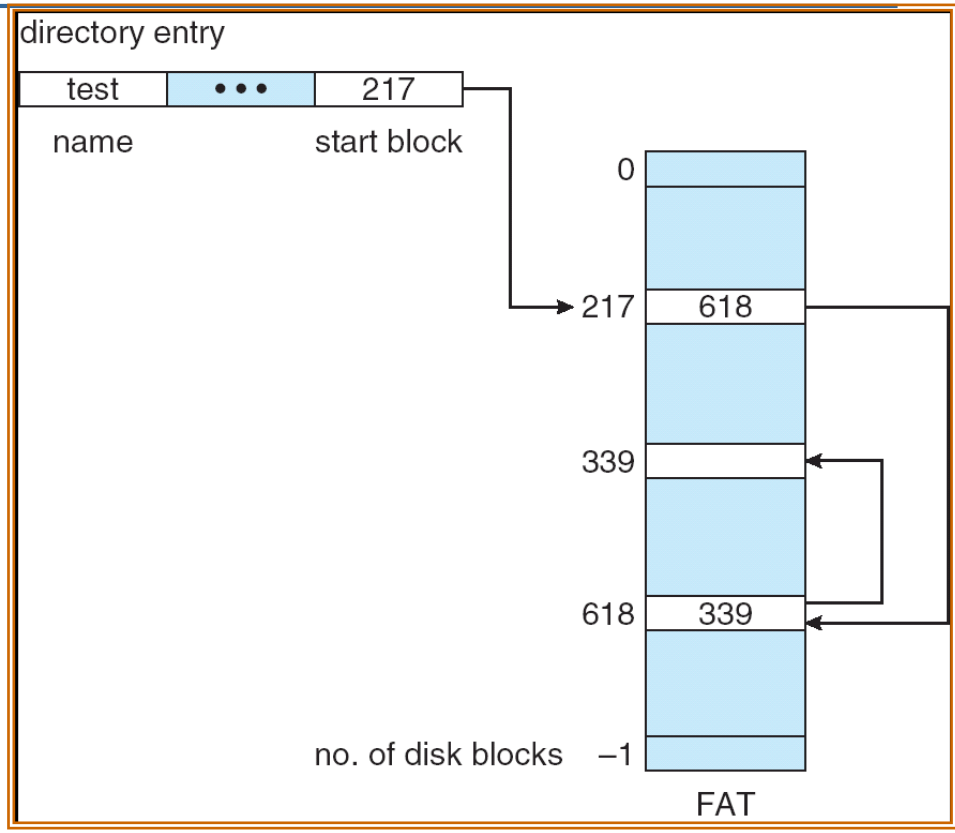


Linked Allocation (cont.)

- Slow - defies principle of locality.
  - Need to read through linked list nodes sequentially to find the record of interest.
- Not very reliable
  - System crashes can scramble files being updated.
- Important variation on linked allocation method
  - File-allocation table (FAT) - disk-space allocation used by MS-DOS and OS/2.

File Allocation Table (FAT)

- Instead of link in each block…
  - put all links **in one table**
  - the **File Allocation Table** (*FAT)*
- One entry per physical block in disk
  - Directory points to first & last blocks of file
  - Each block points to next block (or *EOF*)
  - Unused block: value =0



FAT File Systems

- Advantages
  - Advantages of Linked File System
  - FAT can be *cached* in memory
  - Searchable at CPU speeds, pseudo-random access
- Disadvantages
  - Limited size, not suitable for very large disks
  - FAT cache describes *entire* disk,
    - not just open files!
  - Not fast enough for large databases
- Used in MS-DOS, early Windows systems

Disk Defragmentation

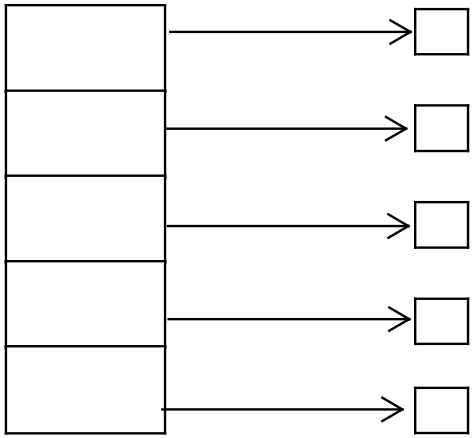
- Re-organize blocks in disk so that file is (mostly) contiguous
- Link or FAT organization preserved
- Purpose:
  - To reduce disk arm movement during sequential accesses

30

Allocation Methods - Indexed

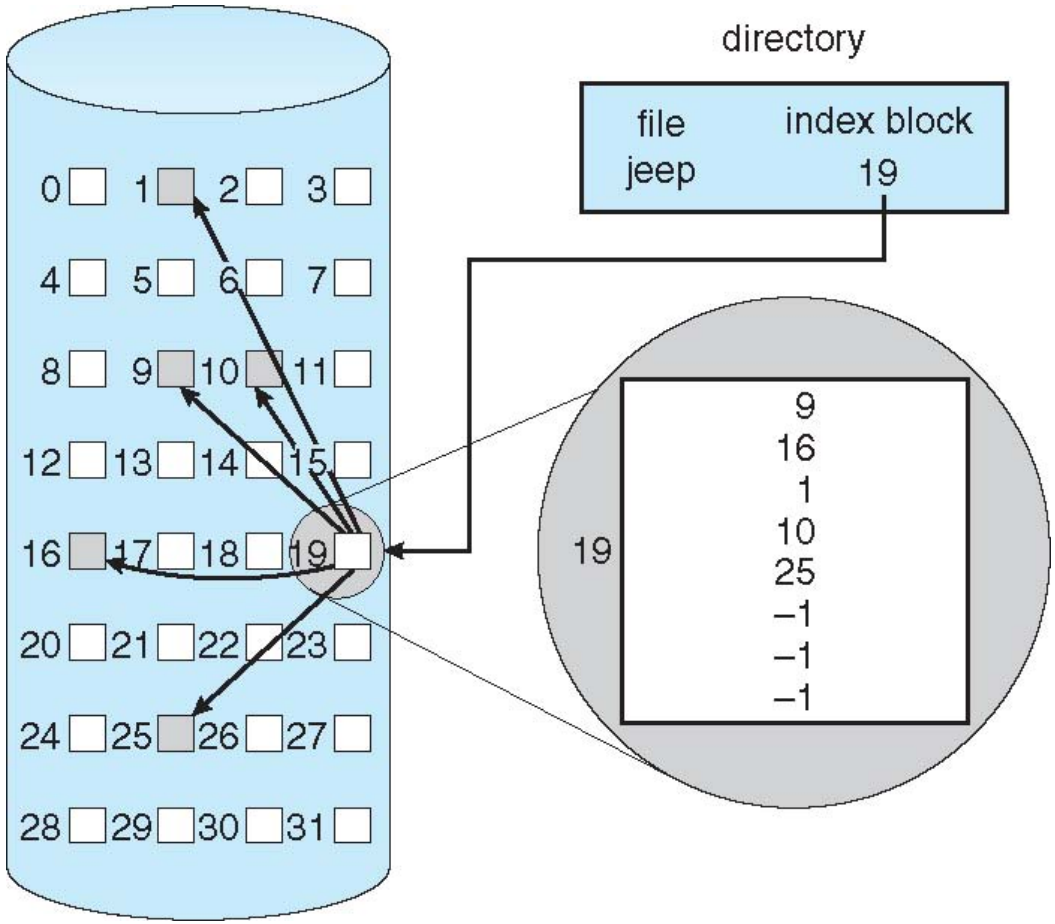
- If FAT is not used, linked allocation cannot support efficient direct access,
  - since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
  - How to solve this?
- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



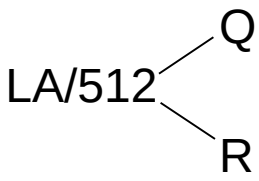
index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

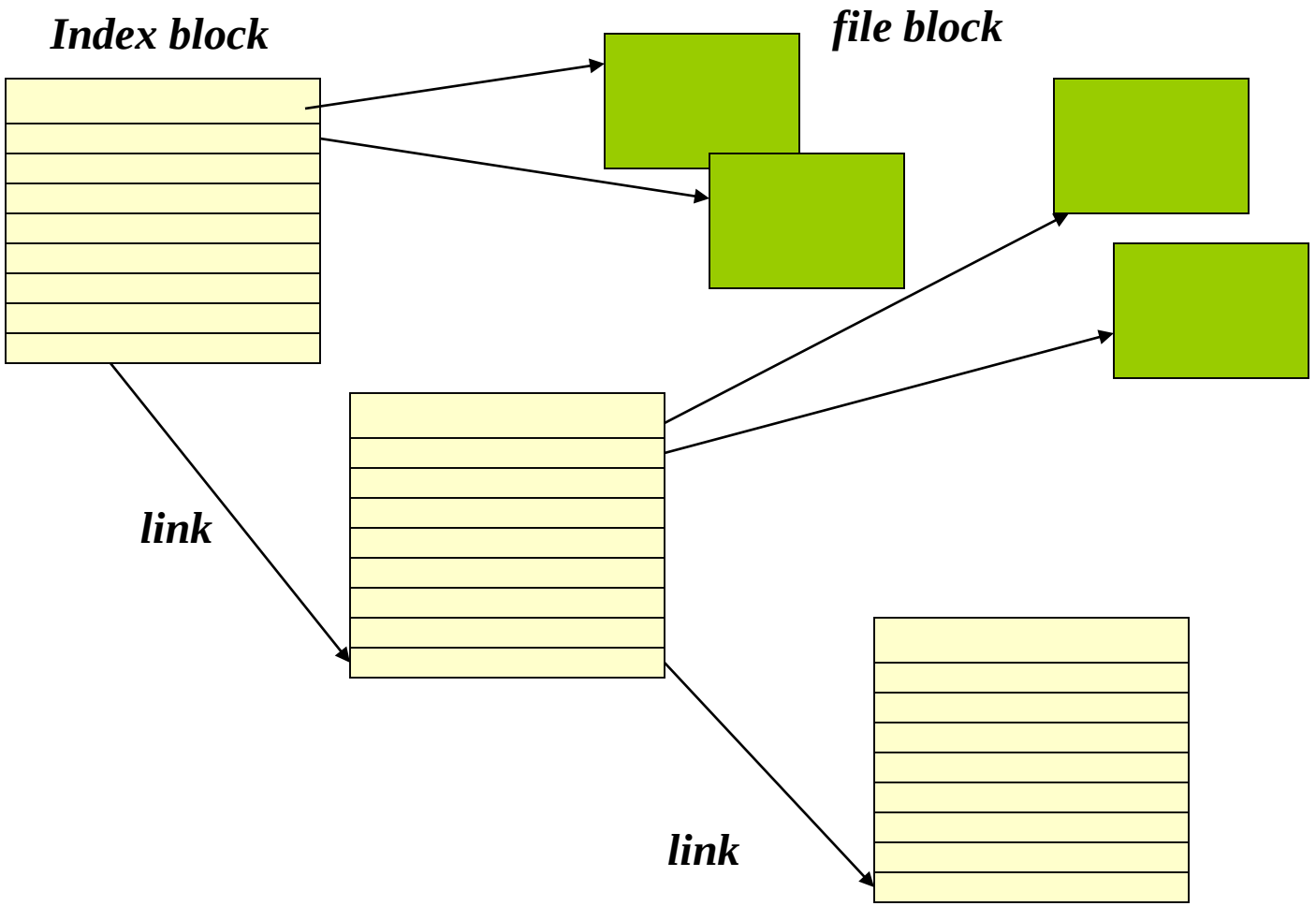
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical
  - in a file of maximum size of 256K bytes and
  - block size of 512 bytes.
  - We need only 1 block for index table



Q = displacement into index table
R = displacement into block

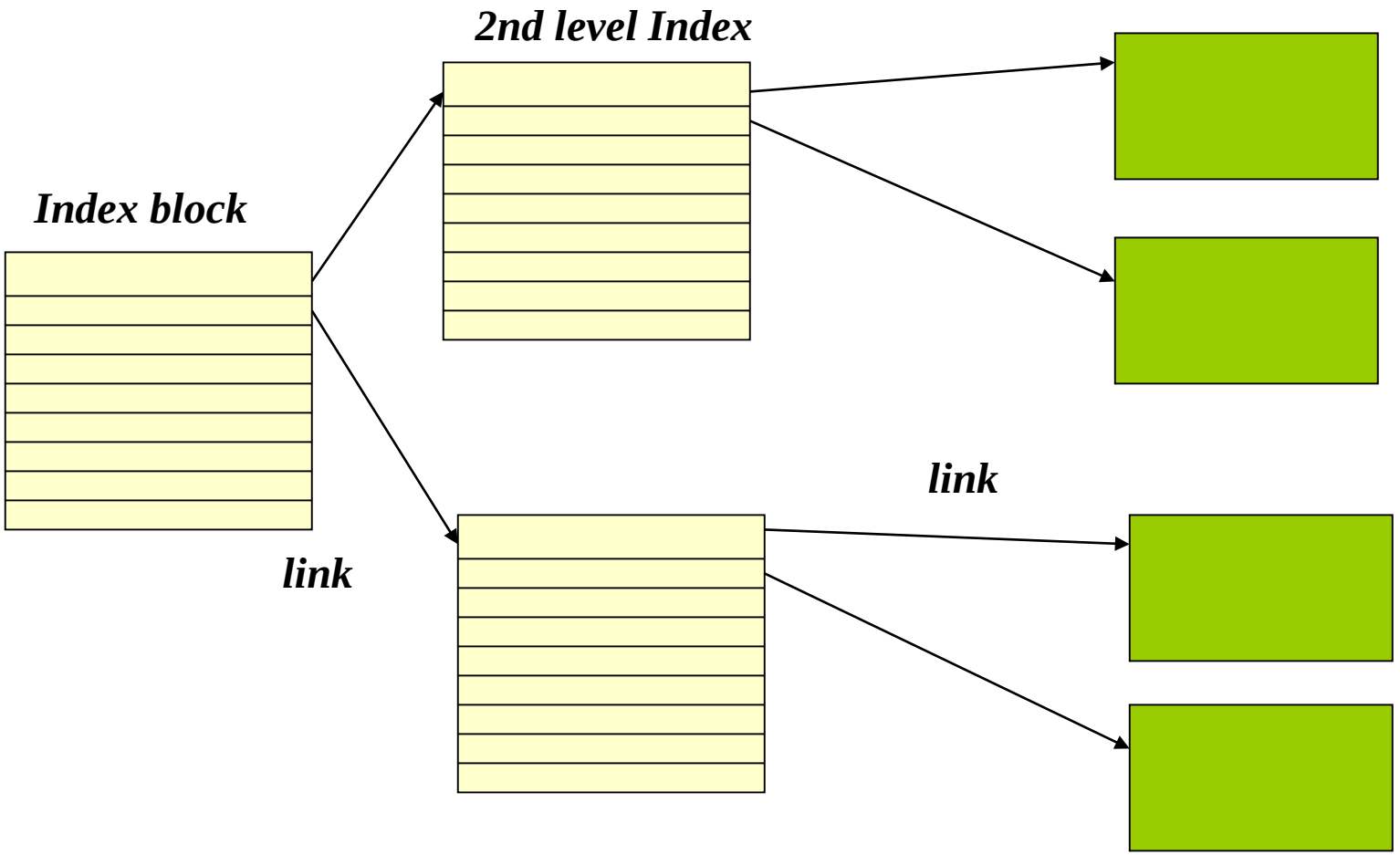
Indexed Allocation – Mapping (Cont.)

- A single index block might not be able to hold enough pointers for a large file
  - Several schemes to deal with this issue (e.g., linked, multi-level, combined)
- **Linked scheme** – Link blocks of index table (no limit on size)



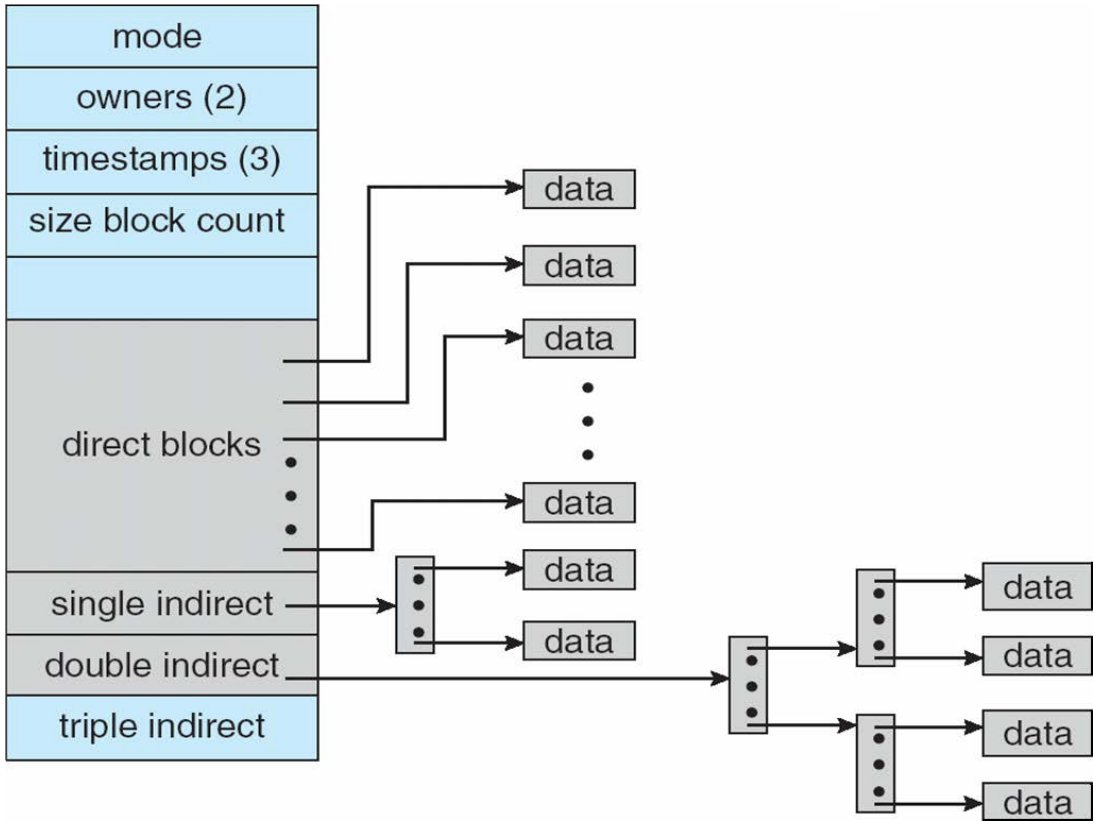
Indexed Allocation – Mapping (Cont.)

- **Two-level index** (generalizes to multi-level)
  - 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: used in UNIX UFS

4K bytes per block, 32-bit addresses



Combined Scheme

- Another alternative, used in UNIX-based file systems
  - Keep the first, say, 15 pointers of the index block in the file's inode.
  - First 12 of these pointers point to direct blocks
    - They contain addresses of blocks that contain data of the file.
    - Thus, the data for small files (of no more than 12 blocks) do not need a separate index block.
    - If the block size is 4 KB, then up to 48 KB of data can be accessed directly.
  - The next 3 pointers point to indirect blocks:
    1. The first points to a single indirect block,
       - which is an index block containing not data but the addresses of blocks that do contain data.
    2. The second points to a double indirect block,
       - which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.
    3. The last pointer contains the address of a triple indirect block.

Unix inode

- An **inode** (**index node**) is a control structure that contains key information needed by the OS to access a particular file.
  - Several file names may be associated with a single inode,
  - but each file is controlled by exactly ONE inode.

- On the disk, there is an **inode table**
  - Contains the inodes of all the files in the filesystem.
  - When a file is opened, its inode is brought into main memory and stored in a memory-resident inode table.

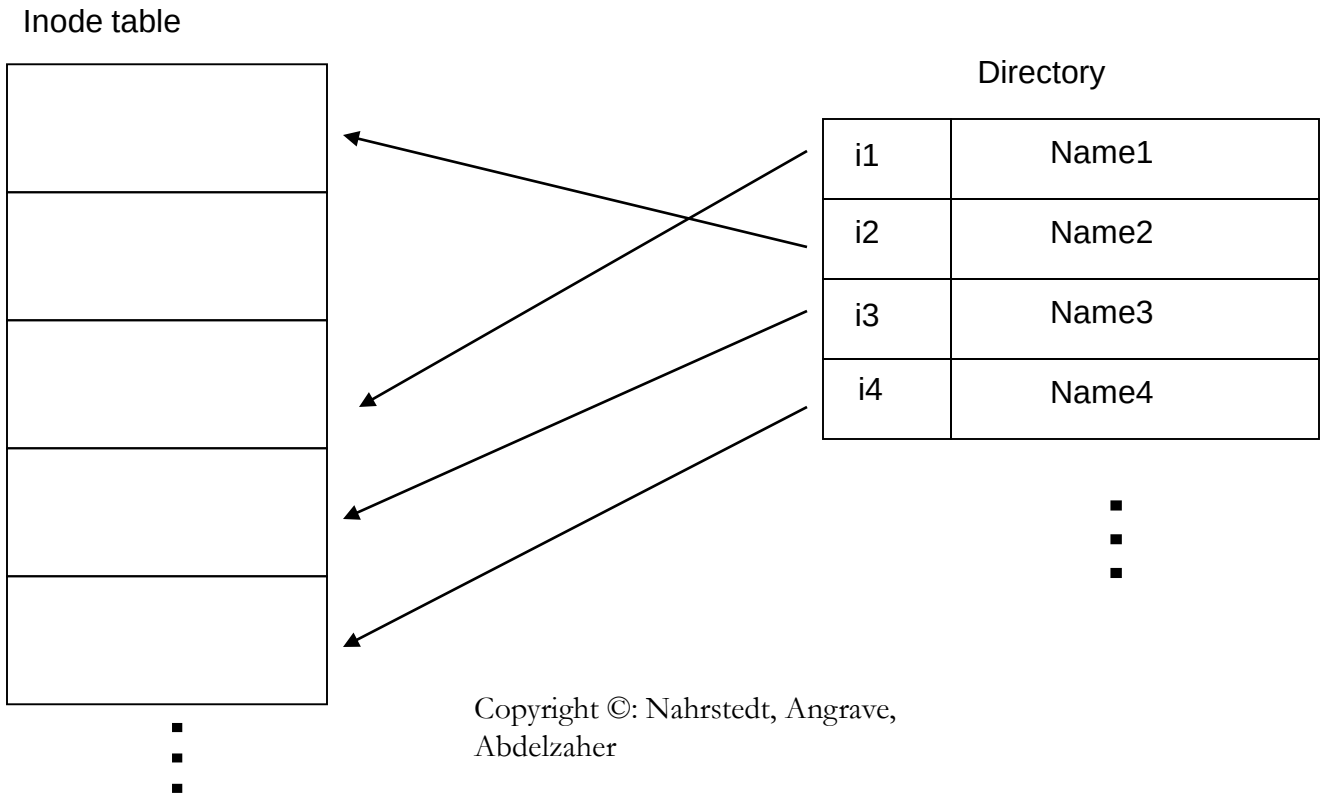
39

Information in the inode

File Mode	16-bit flag that stores access and execution permissions associated with the file. 12–14 File type (regular, directory, character or block special, FIFO pipe 9–11 Execution flags 8 Owner read permission 7 Owner write permission 6 Owner execute permission 5 Group read permission 4 Group write permission 3 Group execute permission 2 Other read permission 1 Other write permission 0 Other execute permission
Link Count	Number of directory references to this inode
Owner ID	Individual owner of file
Group ID	Group owner associated with this file
File Size	Number of bytes in file
File Addresses	39 bytes of address information
Last Accessed	Time of last file access
Last Modified	Time of last file modification
Inode Modified	Time of last inode modification

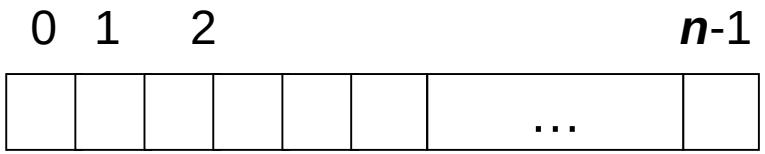
Directories

- In Unix a directory is simply a file that contains a list of file names plus pointers to associated inodes



Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term "block" for simplicity)
  - Several ways to implement
- **Bit vector** or **bit map** (***n*** blocks)



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

Bit Vector (contd.)

- **Pros:**
  - Relative simplicity
  - Efficiency in finding the first **n** consecutive free blocks on the disk.
    - Easy to get contiguous files

- **Example:** one technique for finding the first free block
  - Sequentially check each word in the bit map if it is not 0
    - 0-valued word contains only 0 bits
    - represents a set of allocated blocks.
  - First non-0 word is scanned for the first 1 bit,
    - which is the location of the first free block.

- The calculation of this free block number is
  - (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 = 4KB = 2^{12} bytes

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

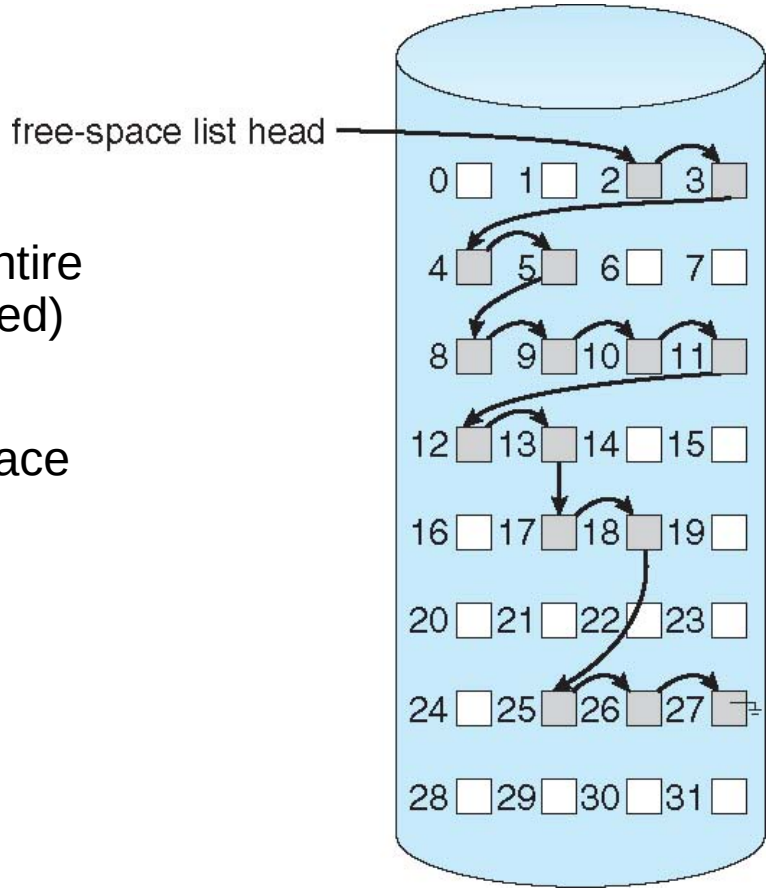
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory

- Example: BSD File system

Linked Free Space List on Disk

- **Linked list** (**free list**) -- keep a linked list of free blocks
- **Pros:**
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)
- **Cons:**
  - Cannot get contiguous space easily
  - not very efficient because linked list needs traversal.



Free-Space Management (Cont.)

- **Linked list of indices - Grouping**
  - A modification of the free-list approach
  - Keep a linked list of index blocks.
  - Each index block contains:
    1. addresses of free blocks and
    2. a pointer to the next index block.
  - **Pros:** A large number of free blocks can now be found quickly
    - Compared to the standard linked-list approach
- **Counting**
  - Linked list of contiguous blocks that are free
  - Free list node contains pointer and number of free blocks starting from that address.

Efficiency and Performance

- In general, the efficiency of a file system dependents on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Fixed-size or varying-size data structures used

- Even after the basic file-system algorithms have been selected, we can still improve performance in several ways.

Efficiency and Performance (Cont.)

- Performance improved by:
  - Keeping data and metadata close together – generic principle
    - Do not want to perform a lot of extra I/O just to get file information
  - Using **buffer cache** – separate section of main memory for frequently used blocks

- Optimize caching - depending on the access type of the file.
  - E.g., a file being accessed sequentially then use read-ahead
  - **Read-ahead** -- a requested page and several subsequent pages are read and cached.
    - These pages are likely to be requested after the current page is processed.
    - Retrieving these data from the disk in one transfer and caching them saves a considerable amount of time.

Recovery

- Files and directories are kept both in main memory and on disk
  - Care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency.
  - How to recover from such a failure
- **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
- Recover lost file or disk by **restoring** data from backup

End of Chapter 12