Introduction to OS

File Management

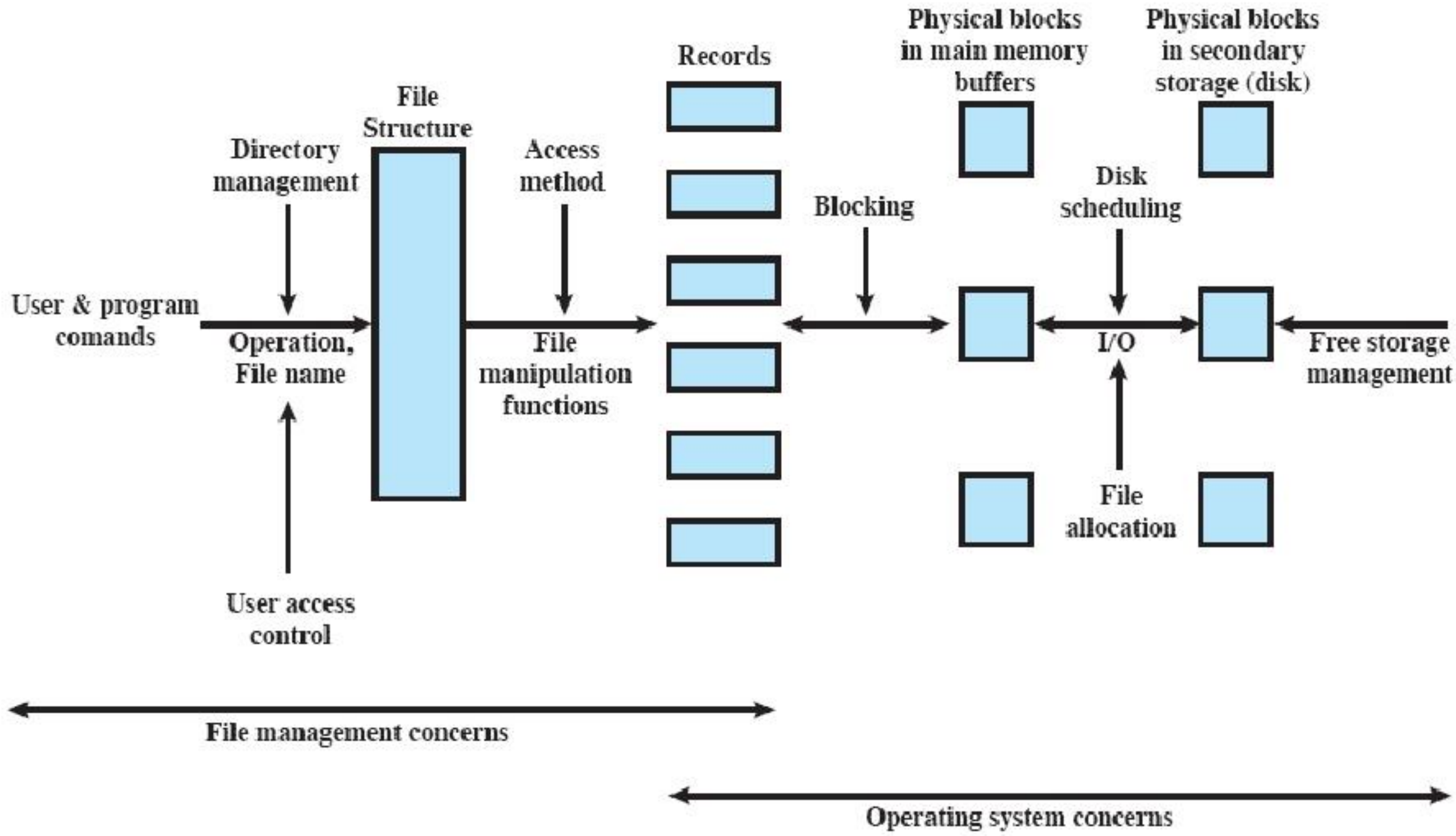
MOS Ch. 4

Mahmoud El-Gayyar
elgayyar@ci.suez.edu.eg

File Management Objectives

- Provide I/O support for a variety of storage device types
- Provide a standardized set of I/O interface routines to user processes
- Provide I/O support for multiple users
- Guarantee that the data in the file are valid
- Optimise performance

File Magement



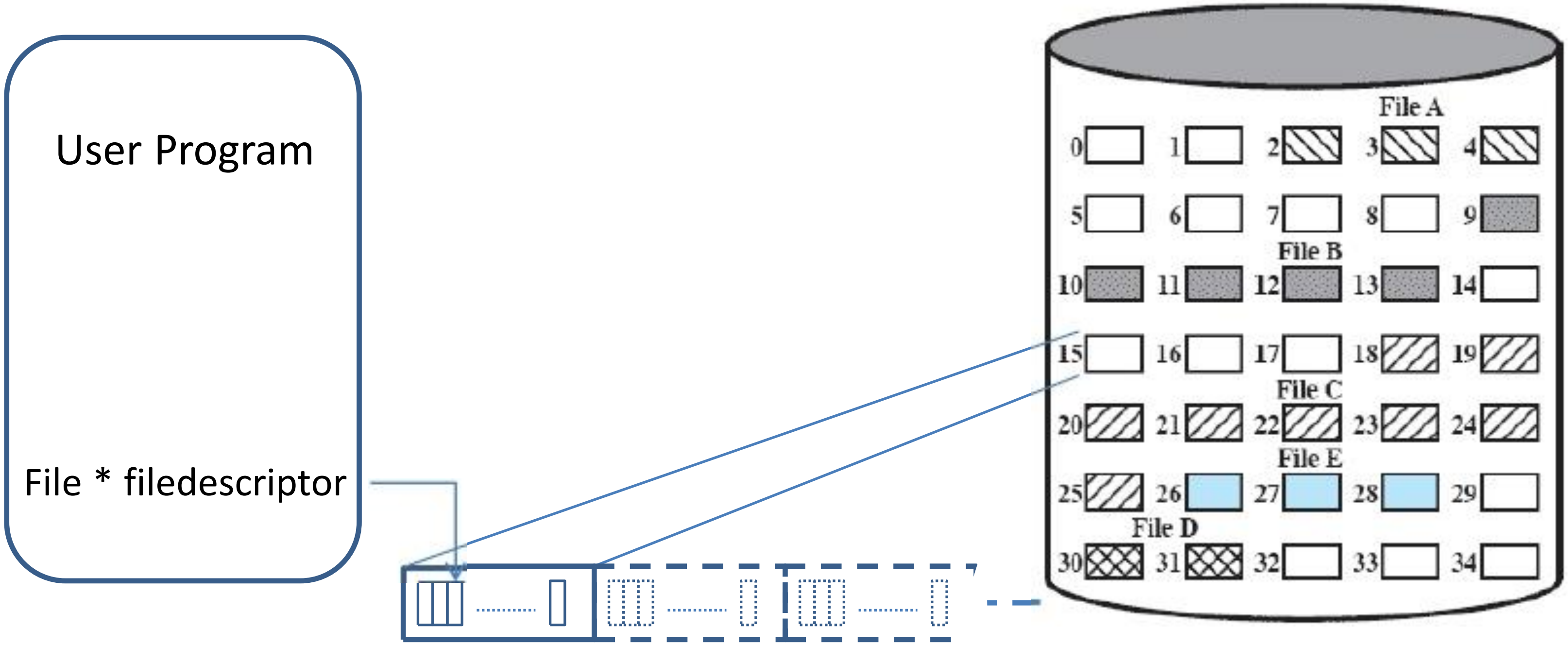
Files - I

- Files provide a way to store information on disk
- Properties
  - Persistence / long-term existence
  - Shareable between processes
    - Have associated file permission, attributes that express ownership, allow a controlled sharing of files
  - Organisational Structure / File System
    - Files can be organised into hierarchical structures to reflect the relationships among files

Files - II

- Files are an abstraction concept
  - Users operate with a simple model of a byte stream being written to or read from disk
  - Operating system hides details how disk space is allocated to store the information represented by a file
- ***File systems*** manage files on disk space

Files - III



- Program regards file as a byte stream, file descriptor points to buffer
- Operating system loads disk blocks belonging to a file
- Questions:
  - Which disk blocks belong to a file?
  - Which block is the next block in sequence?

File Abstraction

- Operations
  - read, write, seek, create, delete
- Meta-data that describes a file
  - Directory entry stores file attributes
  - File attributes
    - ***Name***, ***type***
    - ***Location***: where to find the actual data on disk
    - ***Size***
    - ***Access control***: who may read / write /execute
    - ***Time***: creation, last access, etc
    - ***Version***

File Systems

- Manages storage of data on disk
- Organisation unit is a file:
  - Data object that occupies disk space
- File systems organise disk space
  - The disk itself becomes a data object – container for files
- Concerns:
  - ***Localization***: Records where and how files are stored
  - ***Structure***: Files are organised in directories / folders
  - ***Access***: Allows the creation of files, read and write operations
  - ***Performance***: reduce I/O operations
  - ***Reliability***: can recover from system crash and faults
  - ***Security***: Protection and ownership

File Allocation, Storage Management

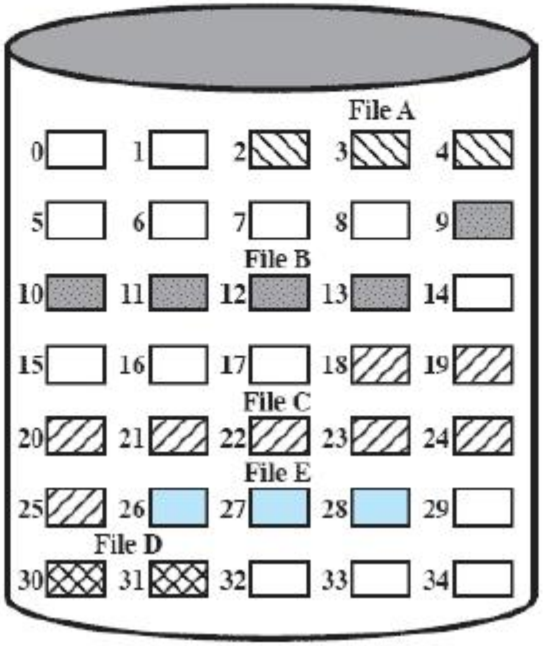
- On secondary storage, a file consists of a collection of ***blocks***
- The operating system / file management is responsible for allocating blocks to files
- Two issues
  – ***Allocated-space management***: record how space on secondary storage is allocated to files
  – ***Free-space management***: OS must keep track of the space available for allocation

File Allocation Method

- Disks are organised in a block structure, each block of a particular size
- A file is stored on disk as a collection of these blocks
  - Blocks are allocated to files
- Block allocation strategies
  - ***Contiguous*** allocation
  - ***Non-contiguous*** allocation:
    - chained allocation
    - Indexed allocation
      - FAT, i-Nodes

Contiguous Allocation of Blocks

- Simplest form, simple to implement, excellent read performance as a file spans across a contiguous set of disk blocks
- Over time, disk becomes fragmented, compaction necessary, external fragmentation
- Infeasible for disk management, was used on magnetic tapes
- Is again important for write-once optical devices such as CD-ROMS
  - File size is known in advance, file is written in one action, occupies a contiguous space

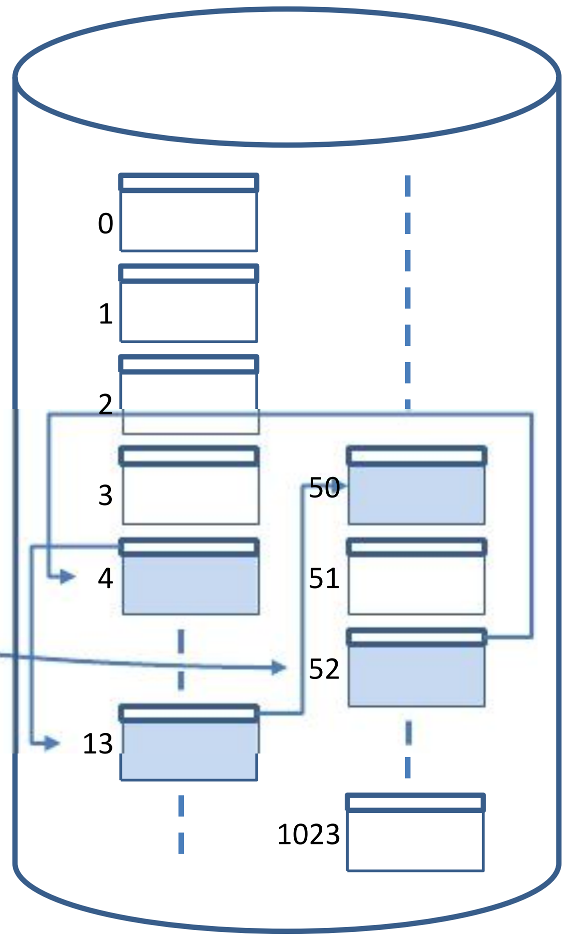


File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

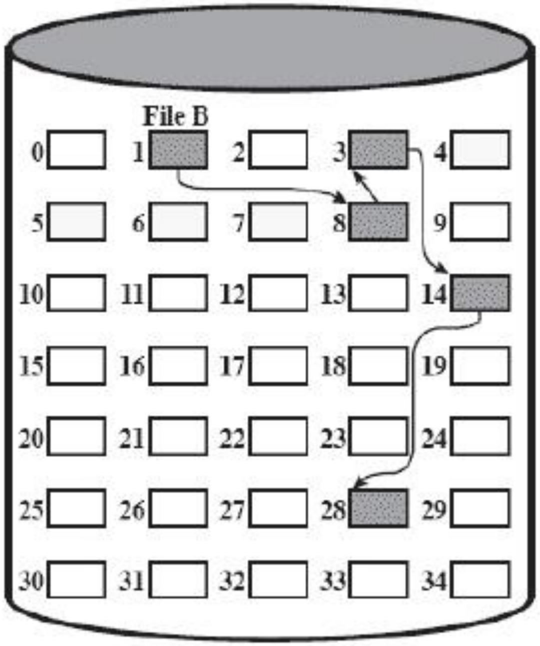
Chained Allocation
Non-contiguous Allocation

- A file may occupy a non-contiguous disk area
- The blocks allocated to a file form a chain:
  - Each block points to its successor block
- Advantage
  - No external fragmentation



Chained Allocation - II

- Each block of a file contains a pointer to a next block – blocks form a chain
- **No space lost due to disk fragmentation**
  - No external fragmentation
- **Reading a file sequentially is straight-forward**
  - Follow the pointer to the next chain element
- **Random access extremely slow**
  - We have to follow the chain pointers until we find the right disk block
  - I/O operations for each visited block: must be read to access pointer and read next block
- **Waste of space**
  - Chain pointer is part of disk block
  - A small part (32-bit or 64-bit address, 4 or 8 bytes) are wasted on these pointers

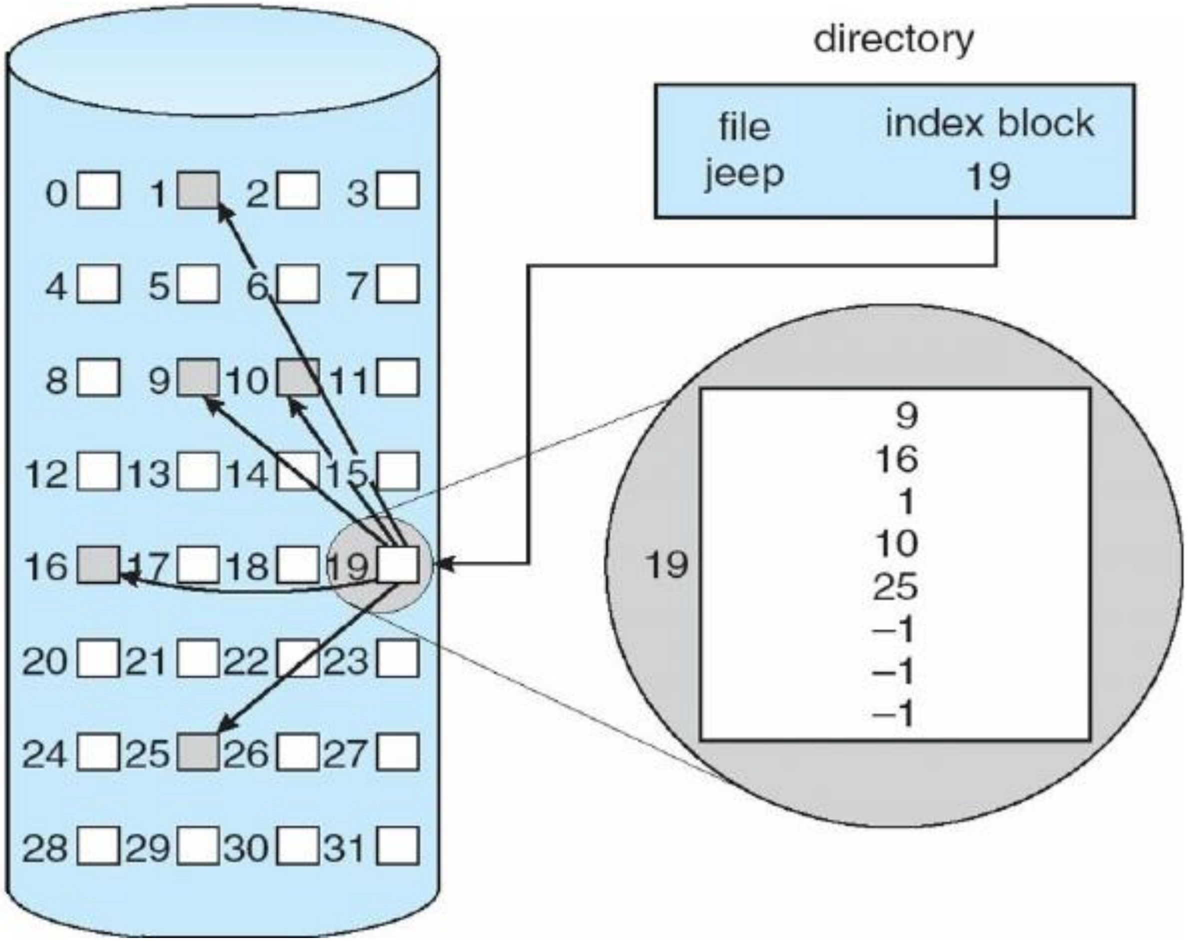


File Allocation Table

File Name	Start Block	Length
•••	•••	•••
File B	1	5
•••	•••	•••

Indexed Allocation - I

- A directory entry points to a disk block that contains an index table for a file



Indexed Allocation - II

- Eliminates disadvantages of chained allocation
  - takes the pointers out of the data disk blocks and collects them in an extra table

- Two important fans
  - File Allocation Table FAT (MSDOS / Windows)
  - i-Nodes (Unix)

File Allocation Table (FAT)

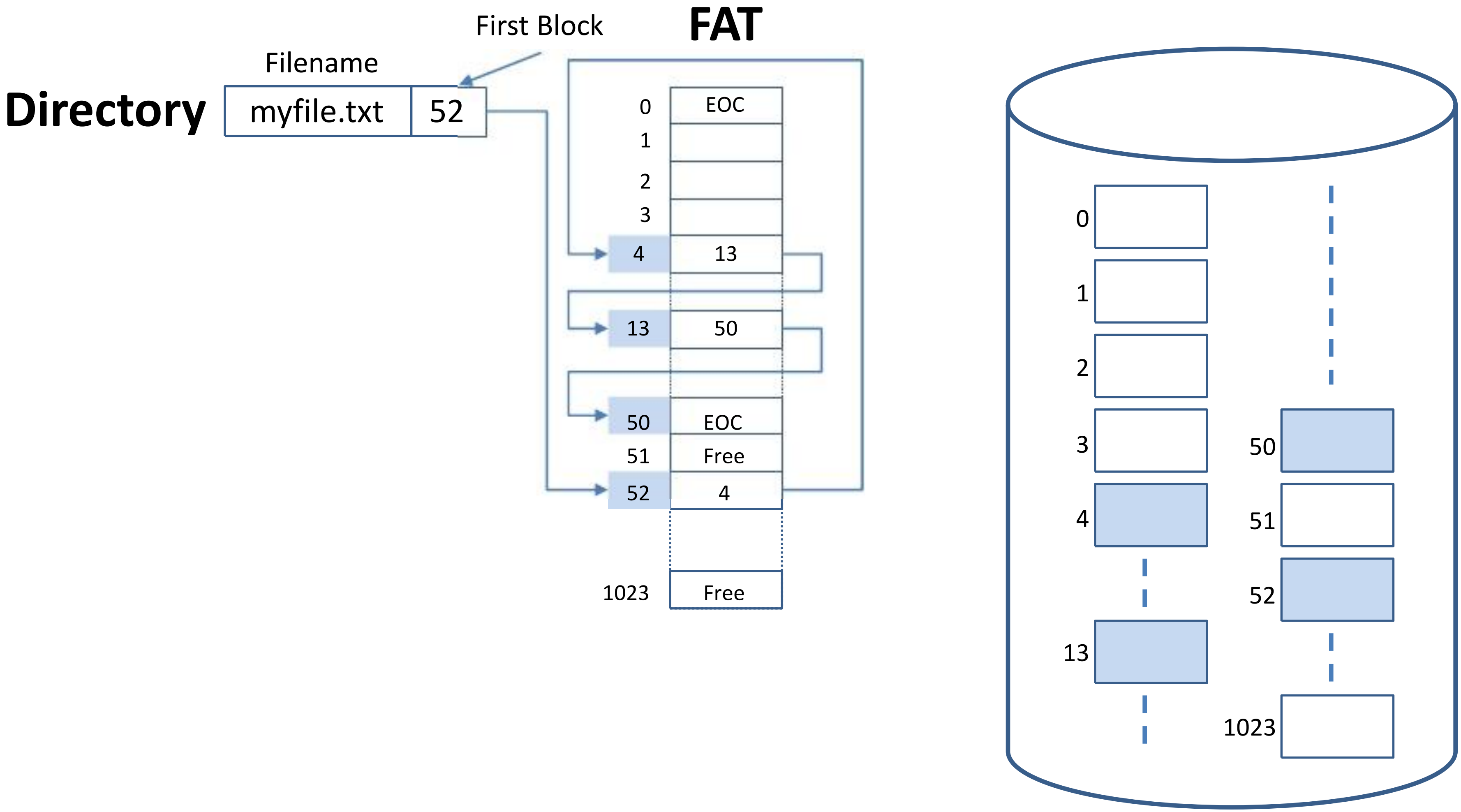
- Combines chained allocation with a separate index table – the "File Allocation Table" (FAT)
  - takes the pointers out of the disk blocks and collects them it in an extra table – the File Allocation Table (FAT)
- FAT table itself is stored at the beginning of the disk, occupies itself a couple of blocks

- **Advantage:**
  - FAT can be traversed very fast for block chains
  - Good for direct access to a single block as well as a sequential read of a file

- **Disadvantage:**
  - FAT itself may be large, has to be held in memory, must be saved on the disk as well

  - 200-GB disk and a 1-KB block size, 200 million FAT entries. Each entry has to be a minimum of 3 bytes. Thus the table will take up 600 MB

Example: File Allocation Table FAT

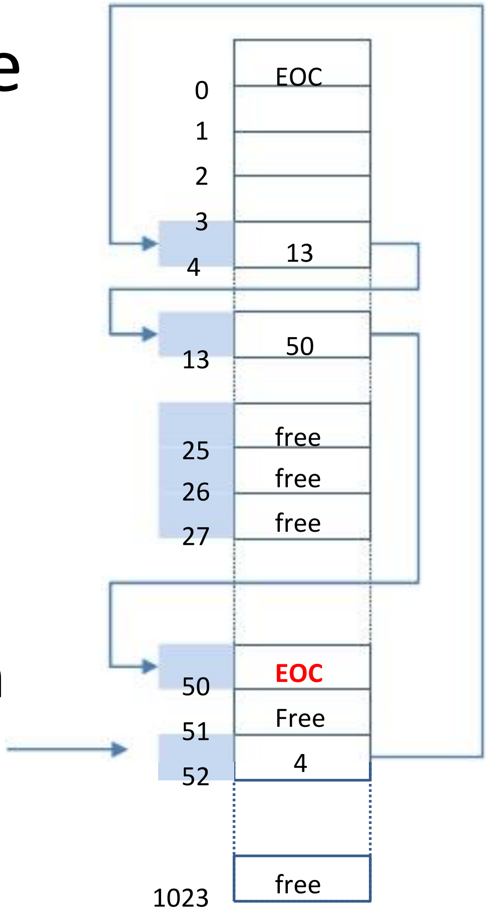


File Allocation Table (FAT)

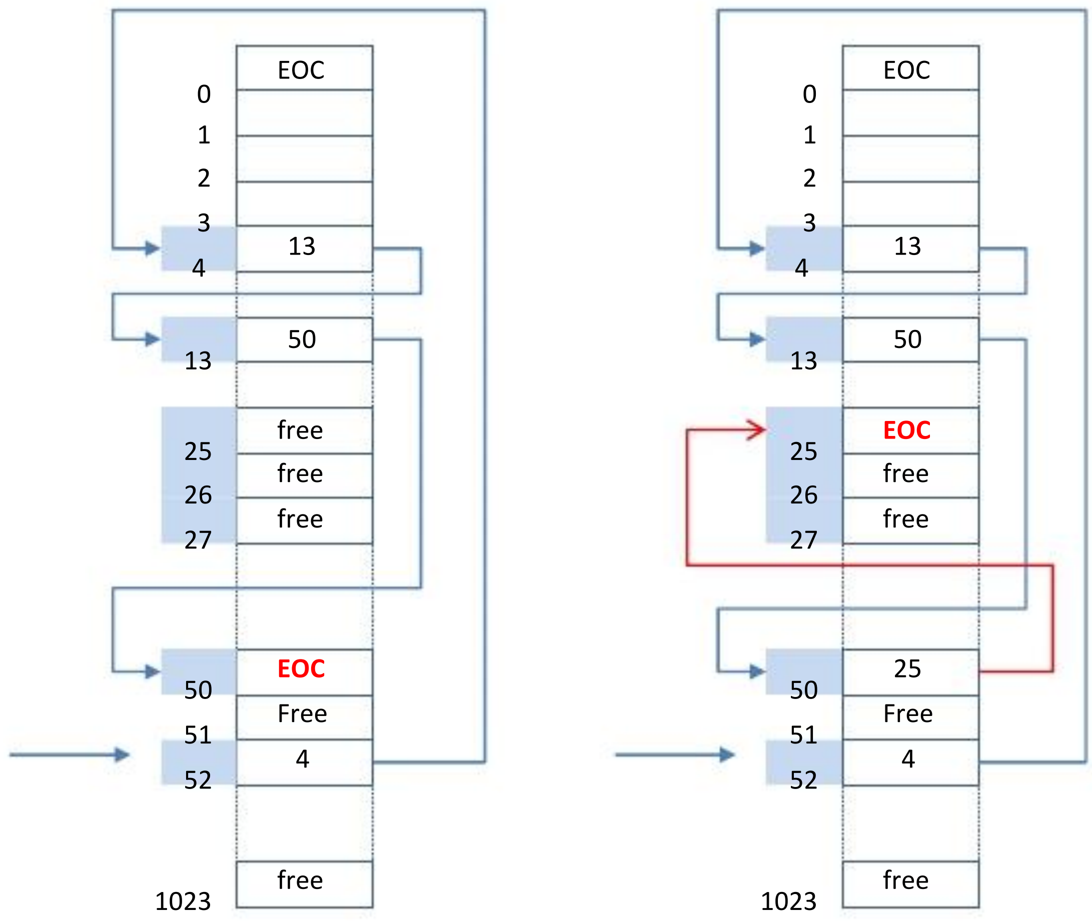
- File Allocation Table is **loaded into memory**, when disk is mounted by operating system
  - All chain pointers now in main memory
  - can easily be followed to find a block address
  - I/O action only needed to load actual disk block
- Entries in FAT form a block chain for a file
  - The index of the FAT entry is the block address of a file
  - The content of the FAT entry is the index of the next FAT entry in the chain and the block address of the next disk block of the file

FAT: Extending a File

- Allocating a new disk block for a file
  – Information about free blocks are held in the FAT
- Find a FAT entry that is marked as "free" and extend the block chain
- I/O operation for FAT:
  – As FAT is changed, it has to be written to disk – can be immediate or deferred

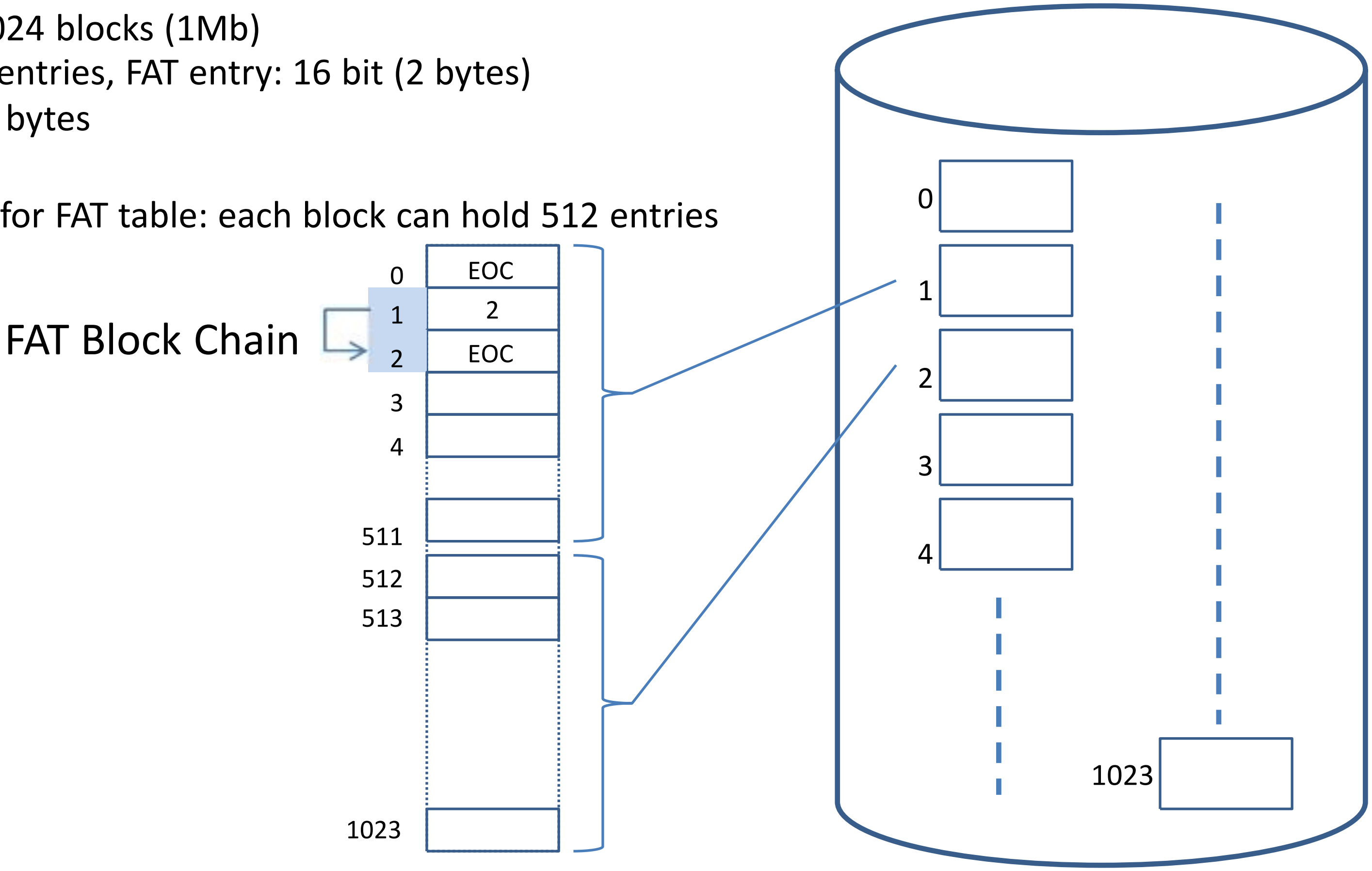


FAT: Extending a File



Storing the FAT

- The FAT itself has to be permanently stored on disk
  - Will occupy disk blocks
  - There is a block chain for the FAT itself
- ***Given:***
  - Hard disk has 1024 blocks (1Mb)
  - FAT table: 1024 entries, FAT entry: 16 bit (2 bytes)
  - Block size: 1024 bytes
- ***We need***
  - Two disk blocks for FAT table: each block can hold 512 entries



Size of FAT

- Storage: 8GB USB drive, Block size: 4KB
- How many blocks do we need on the disk for the FAT?
- Remember:
  - 8GB = 8 x 1024 x 1024 x 1024 bytes
  - 4KB = 4 x 1024 bytes
- We calculate:
  - 8GB / 4KB = 8 x 1024 x 1024 x 1024 bytes / 4 x 1024 bytes = 2 x 1024 x 1024 = 2 Mio blocks
- Addressing:
  - We need at least 2^{21} entries in the FAT to address all 2 Mio blocks (2x 2^{20})
  - We choose a 32-bit format for FAT entries (4 bytes), 1 block can hold 1024 entries: 4 x 1024 bytes / 4 bytes = 1024 entries
- Space for FAT on disk
  - 2 x 1024 x 1024 entries / 1024 entries = 2 x 1024 = **2048 blocks for the FAT**

Bytes	Exponent			
1,024	2^{10}	1kb	1024bytes	
1,048,576	2^{20}	1MB	1024kb	1024 x 1024
1,073,741,824	2^{30}	1GB	1024MB	1024 x 1024 x 1024
4,294,967,296	2^{32}	4GB	4 x 1024MB	4 x 1024 x 1024 x 1024
1,099,511,627,776	2^{40}	1TB	1024GB	1024 x 1024 x 1024 x 1024
1,125,899,906,842,620	2^{50}	1PB	1024TB	1024 x 1024 x 1024 x 1024 x 1024
1,152,921,504,606,850,000	2^{60}	1EB	1024PB	1024 x 1024 x 1024 x 1024 x 1024 x 1024
18,446,744,073,709,600,000	2^{64}	16EB		16 x 1024 x 1024 x 1024 x 1024 x 1024 x 1024

FAT: Deleting Files

- Deleting a file is fast
- Two actions
  - The directory entry for a file is marked as deleted
    - First character of filename is set to some non-printable value to make it “invisible” (in the FAT implementation, it is set to 0xE5)
  - All entries of the block chain are set to “free”
- I/O operation for FAT only:
  - As FAT is changed, it has to be written to disk – can be immediate or deferred

Free Space Management FAT

- Information in FAT table determines whether a block on disk is free
  - All free blocks are marked as “unused”
- When a file is deleted
  - The directory entry for a file is marked as deleted
    - First character of filename is set to 0xE5
  - The block chain for this file is cleared in the FAT
    - All FAT entries of such a chain are set to a value indicating that it is “unused”
- Blocks on disk are ***untouched***, no update of their content is needed

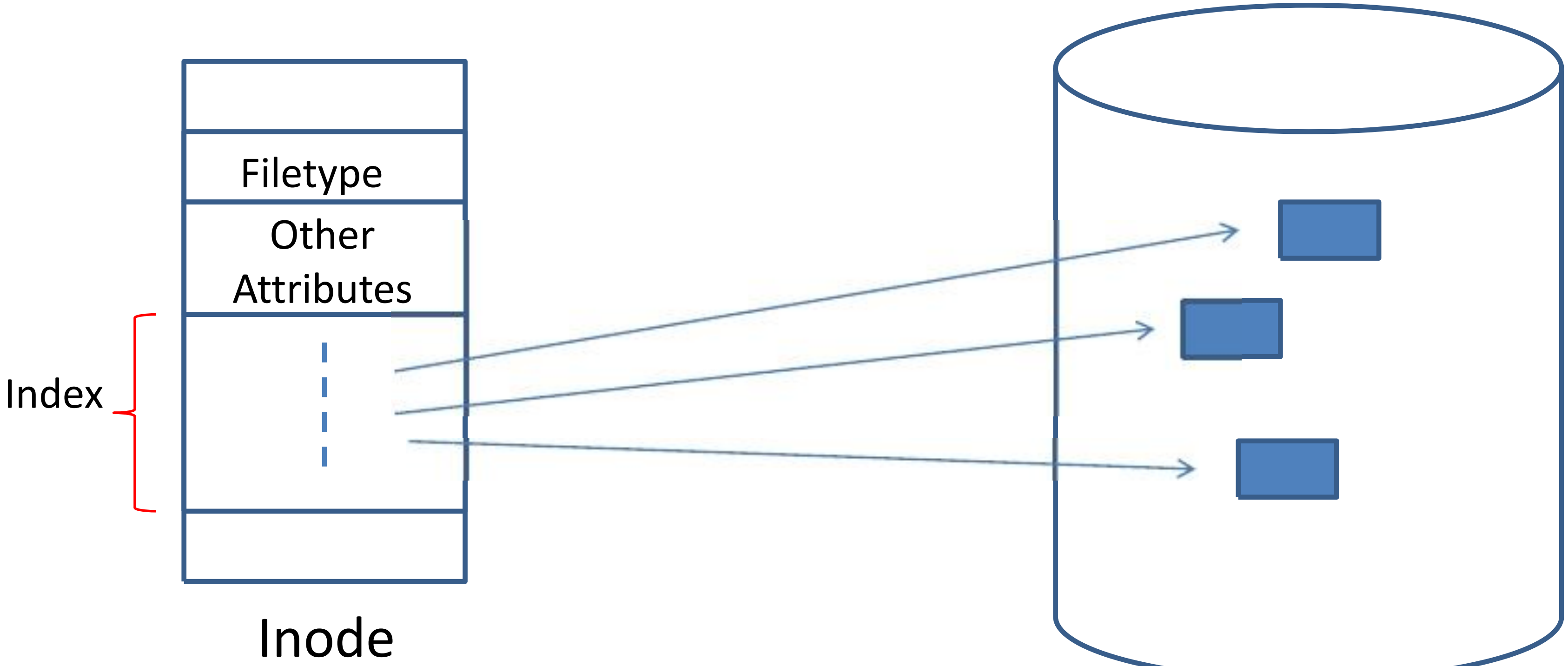
i-Nodes

- All types of Unix files are managed by the operating system by means of i-Nodes
  - A control structure (“index” node) that contains the key information needed by the operating system for a particular file
    - Describes its attributes
    - Points to the disk blocks allocated to a file
- The i-Node is an index to the disk blocks of a file
  - One i-Node per file
- There can be several file names for a single i-Node
  - Under Unix, we can create “links” as aliases for files
- But:
  - An active i-Node is associated with exactly one file
  - Each file is controlled by exactly one i-Node

Indexed Allocation: i-Nodes

- Hierarchical index
  - One disk block can only contain a small list of addresses to disk blocks
  - Has therefore multiple levels: an entry may point to a sub-index table
- Can address very large files
- Most popular form of file allocation (Unix, other systems)
- The i-Node records only the blocks allocated to a file
- Requires a management of a ***separate list*** of free blocks

Inode



- A simple list of block references (single-level) allows fast access to all blocks of a file
- But: it restricts the maximum size of a file

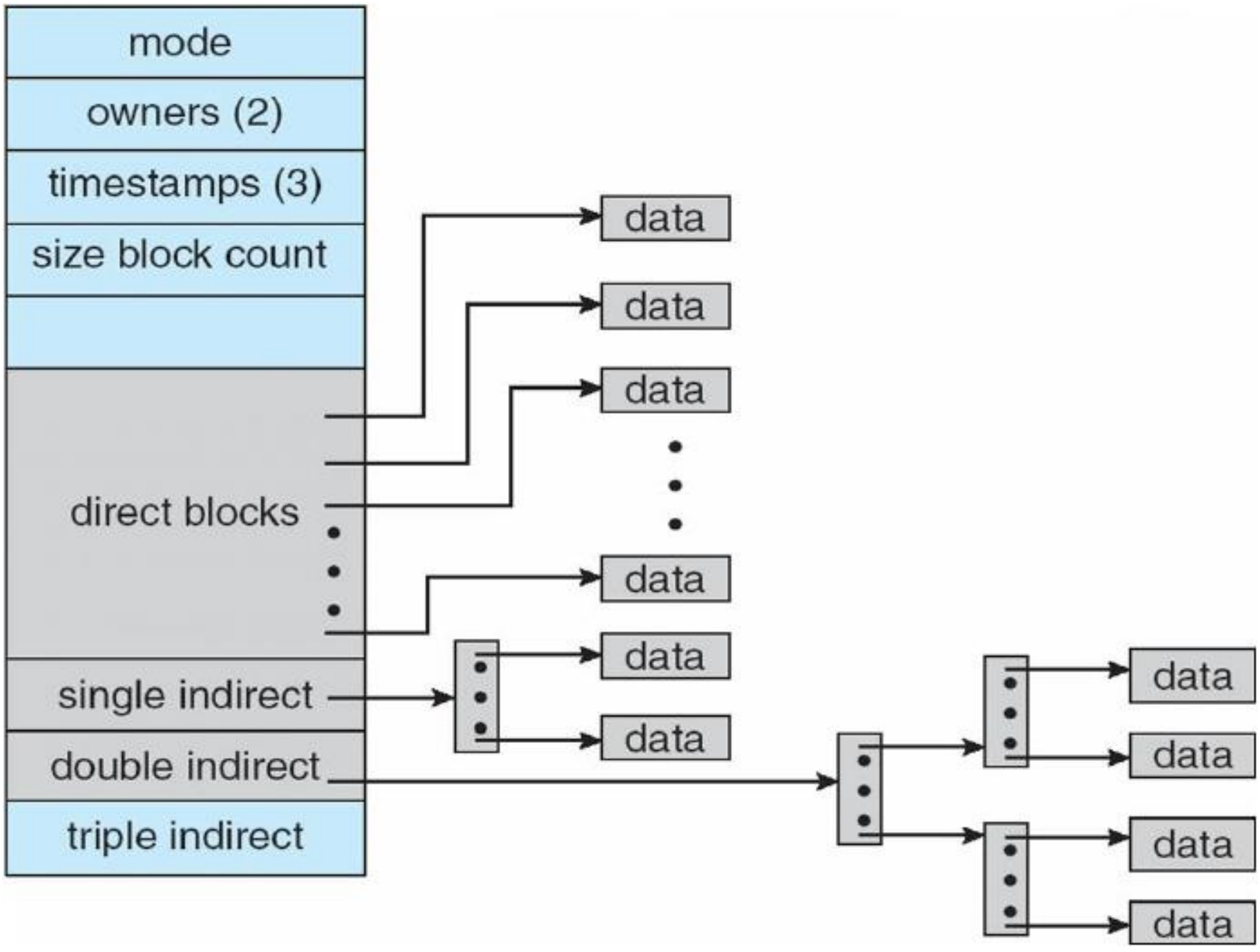
Indexed Allocation: i-Nodes

- i-Node manages n-level index
  - Entry in the i-Node points to a block on disk that contains pointers to other blocks
- How can we distinguish between index blocks and data blocks?
- How do we know how many levels the index has?

File Allocation with Inodes

- Inode contains index referencing allocated blocks
  - First N entries point directly to the first N blocks allocated for the file
  - If file is longer than N blocks, more levels of indirection are used
  - Inode contains three index entries for “indirect” addressing
    - “single indirect” address:
      - Points to an intermediate block containing a list of pointers
    - “double indirect” address:
      - Points to two levels of intermediate pointer lists
    - “triple indirect” address:
      - Points to three levels of intermediate pointer lists
- The initial direct addresses and the three multi-level indirect addressing means form the index

i-Node Indexed References of Disk Blocks



i-Node Direct and Indirect Indexing - I

- Example implementation with 13 index entries:
  - i-Node contains a list of 13 index entries that combine four different forms of index
  - Direct block references:
    - 10 entries of this list point directly to file data blocks
  - Single indirect (two levels):
    - Entry 11 is regarded as always pointing to an index disk block: this index block contains address of actual file data blocks
  - Double indirect (three levels): entry 12 is regarded to be the starting point of a three-level index
  - Triple indirect (four levels): entry 13 is regarded to be the starting point of a four-level index

i-Node Direct and Indirect Indexing - II

- Based on which entry in the i-Node is used, the file system management can distinguish whether an indexed block is a data block or another level of one of the indices
- Assumption
  - There are many small files, the number of directly referenced blocks may be enough
  - For larger files, the additional indices are used

i-Node Table

- Operating system has to manage the i-Node table
  - When a file is opened / created, its i-Node is loaded into the i-Node table
  - The size of this table determines the number of file that can be held open at the same time

File Allocation with i-Nodes

- What is maximum size of a file that can be indexed:
  - Depends of the capacity of a fixed-sized block
- Example implementation with 15 index entries:
  - 12 direct, single (13) / double (14) / triple (15) indirect
  - Block size 4kb, holds 512 block addresses (32-bit addresses)

Level	Number of Blocks	Number of Bytes
Direct	12	48K
Single Indirect	512	2M
Double Indirect	512 × 512 = 256K	1G
Triple Indirect	512 × 256K = 128M	512G

i-Nodes

- **Advantage**
  - i-Node is only loaded into memory when a file is opened
  - Good for managing very large disks efficiently
  - We need a list of i-Nodes of open files: size of this list determines how many files may be open at the same time
- **Disadvantage**
  - The i-Node only has a fixed list for block references
  - If a file is small, fast and efficient management
  - If file is large, the i-Node has to be extended with a hierarchy of indirect block lists connected to the i-Node, ***needs extra I/O operations to scan the index***

Free Space Management - I

- Just as allocated space must be managed, so must unallocated space
- It is necessary to know which blocks are available
- Methods
  - Bit tables: for each block one bit (used, unused)
    - As small as possible
  - Free portions chained together
    - Each time a block is allocated, it has to be read first get the pointer to the next free block
  - Indexing
    - Treats free space as a file
    - Create pool of free i-nodes and free disk blocks

Free Space Management - II

- Bit Table
  - Vector of bits: each bit for one disk block
  - Is as small as possible
- Can still be of considerable size:
  - Amount of memory (bytes) needed:

(Disk size / block size) /8

- Example:
  - 16 GB hard disk, block size 512 bytes: bit table occupies 4 MB, requires 8000 disk blocks when stored on the disk

Free Space Management - III

- Chained Free blocks
  - We can chain free blocks together
  - Each free block contains a pointer to next free block
- Problem
  - When a free block is allocated, it has to be read from disk first to retrieve the "next free block pointer"

Free Space Management - IV

- Indexing:
  - Free space is treated like a file collecting all the free blocks
- Free Block List:
  - Each block is assigned a number sequentially
  - The list of numbers of all free blocks is maintained in a reserved portion of the disk

Directories

- Directories maintain information about files
  - File name
  - Location of actual data related to such a file name
- File name is a symbolic representation of data stored on disk
- Directory entry
  - File name
  - File attributes
  - Physical address of the file data
- Directory structure
  - Simple list
  - Hierarchical, tree structure: directories contain sub-directories

Hierarchical Directories

- Unix uses a hierarchy of directories
- Top-level directory: root
  - All other directories are sub-directories of root
- Path:
  - Is the sequence of subdirectories to reach a file
- Path name:
  - Absolute: uniquely identifies a file within the directory hierarchy
    - Starts with root
    - Example: “/usr/local/myname/myfile.txt”
  - Relative: identifies a file, starting from the current working directory
    - Example:
      - working directory: “/usr”
      - Path name: “local/myname/myfile.txt”
- Special files in a directory:
  - “.” points to the directory itself: “./myfile.txt”
  - “..” points to the parent directory: “../myname/myfile.txt”

Directories in Unix

- Structured as a tree
  – Each directory contains files and/or other sub-directories
- Implementation:
  – A directory is a file that contains a list of file names and a reference to the corresponding inode in the inode table of a volume
  – Inode reference:
    - Is the so-called "i-number": index into the inode table

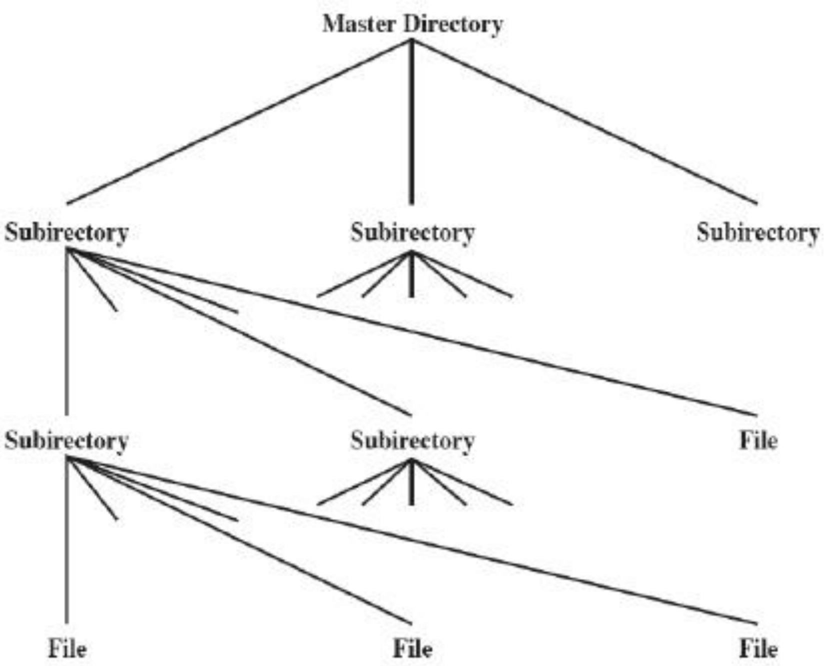


Figure 12.4 Tree-Structured Directory

Unix Directories and i-Nodes

- Directories are structured as a tree
- Directory entries contain filename and associated i-number
  - The index into the i-Node table

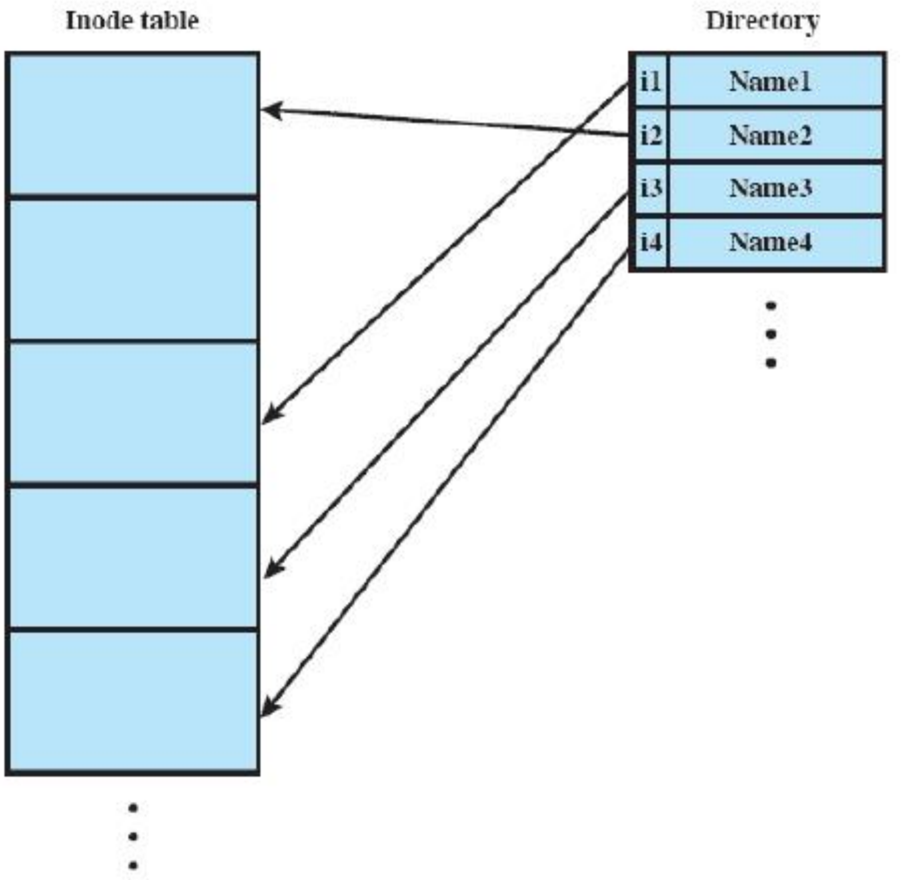


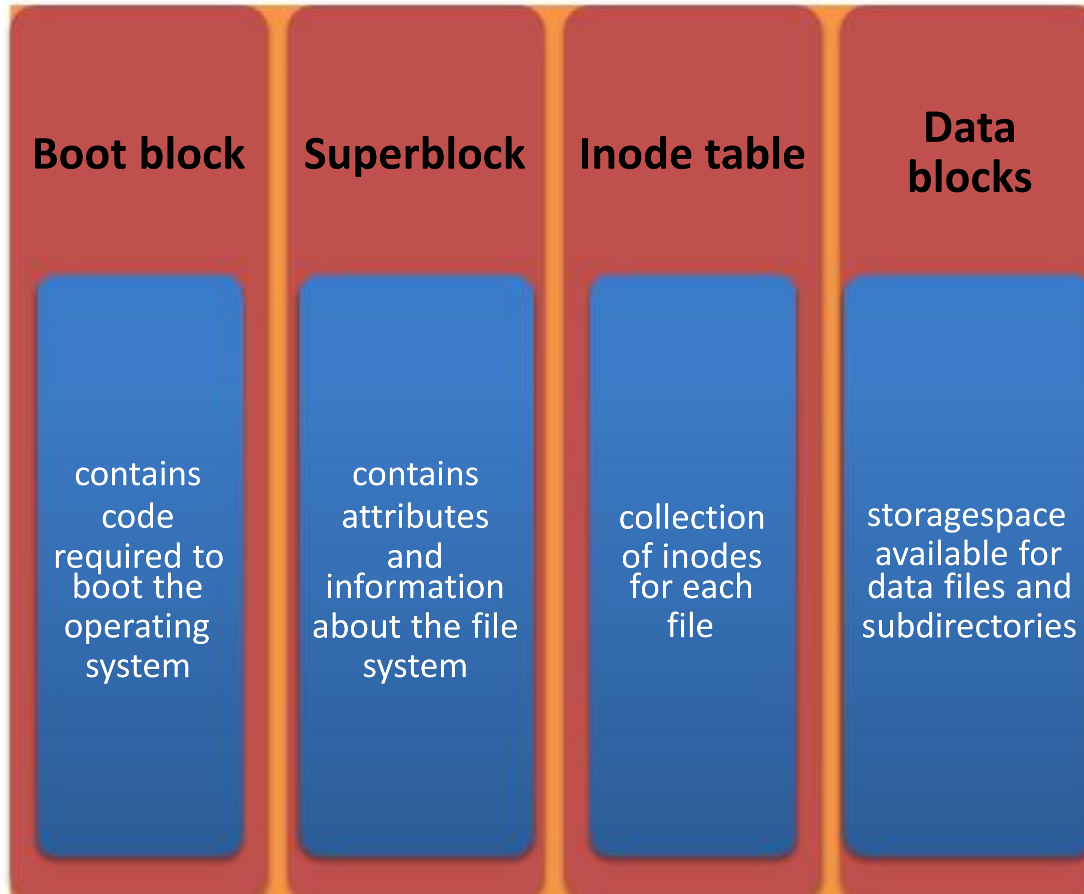
Figure 12.15 UNIX Directories and Inodes

File System Performance

- Is achieved with caches
- Buffer cache
  - Hold data in memory, perform read / write operation much faster
  - Needs some form of block management
    - When a disk block is updated, it must be found in cache
  - Is a danger to file system integrity
  - Unix: system call “sync()” that allows to force a write of cache content
- Write-through cache
  - Disk access for each write operation, data is kept in cache for fast read
  - More secure, less performance

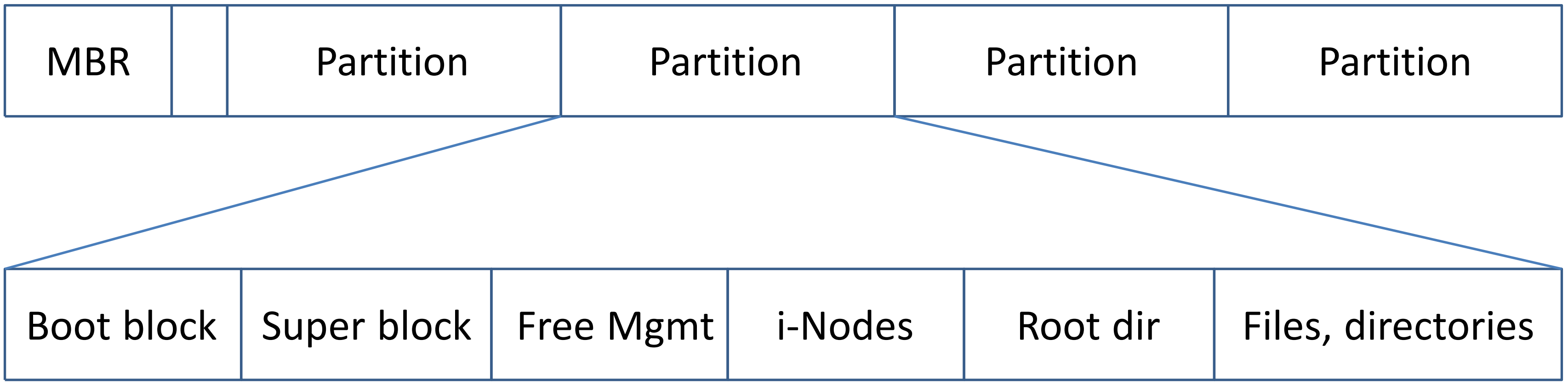
Unix Volume Management

- A UNIX file system resides on a single logical disk or disk partition
- Has a particular layout
  - Boot block
  - Superblock
  - Inode table
  - Data blocks



Unix Disk and File System Layout

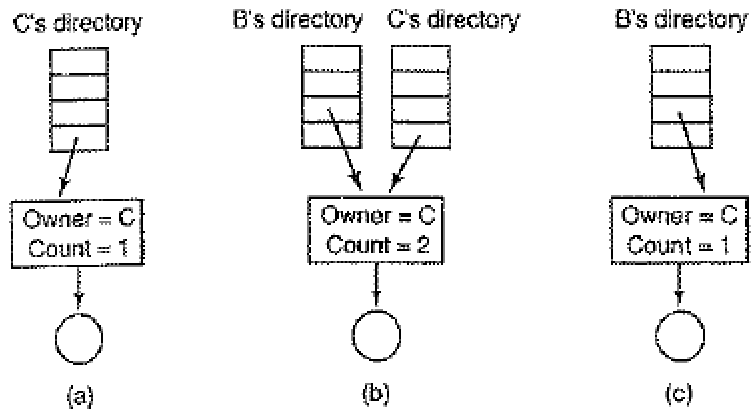
- Master Boot Record (MBR)
  - Sector 0 of disk: Contains boot code
  - Partition table
- System start
  - MBR is loaded into memory
  - Program contained in MBR
    - loads the boot block of the active partition, or
    - Provides menu for loading a particular partition



Unix File Management

- Unix distinguishes six types of files
  - Regular or ordinary
    - Contains arbitrary data in zero or more data blocks
  - Directory
    - Contains a list of file names plus pointers to associated indexing information (inodes) pointing to allocated disk blockes
  - Special
    - Contains no data, are not real files, but used to map physical devices to filenames, usual file management functions can be used for read / writes
  - Named pipes
    - Also a kind of file used to create pipes
  - Links
    - Alternative file name for existing file (multiple directory entries for the same file on the disk), data accessible as long as one hard link exists
  - Symbolic links
    - A special file that contains the name of a file it is linked to

Hard vs. Symbolic Links



- If C tries to remove the file,
  - clears the i-node, *B* will have a directory entry pointing to an invalid i-node. If the i-node is later reassigned to another file, B's link will point to the wrong file.
- With symbolic links this problem does not arise
  - because only the true owner has a pointer to the i-node. Users who have linked to the file just have path names.
  - when destroyed, symbolic link will fail when the system is unable to locate the file.

Review

- Some digital devices need to store data, for example as files. Name a modern device that requires file storage and for which contiguous allocation would be ideal.

- How does MS-DOS implement random access to files?