Distributed File Systems

Chien-Min Wang

Institute of Information Science

Academia Sinica

Contents

Lecture 1
File System Overview

Outline

- Files and Directories
- Implementation Issues
- Example File Systems

Why file systems?

- The data must survive after the termination of the process using it.
- It must be possible to store very large amount of data.
- Multiple processes must be able to access data concurrently.

ð Solution is to store those data in units called files on disks and other media.

Files: an abstraction

- A (potentially) large amount of data that lives a (potentially) very long time.
  - Often *much* larger than the memory of the computer.
  - Often *much* longer than any computation.
  - Sometimes longer than life of the machine itself.
- (Usually) organized as a linear array of bytes or blocks.
  - Internal structure is imposed by applications.
  - (Occasionally) blocks may be variable length.
- (Often) requiring concurrent access by multiple processes
  - Even by processes on different machines!

File Systems

- Files are managed by the Operating System.
- The part of the Operating System that dealt with files is known as the File System.
  - A file is a collection of disk blocks.
  - File System *maps* file names and offsets to disk blocks.
- How files are structured, used, protected and implemented are major concerns of file systems.

Files: Naming$_1$

- The exact rules of naming depend on the operating system.
- However, most of them allow files to be
  - 1 – 8 characters
  - Digits and several special symbols
  - Modern ones support up to 255 characters
- Some file systems are case sensitive.
  - DOS, Windows: Case insensitive
  - UNIX, Linux: Case sensitive

Files: Naming_2

- Many operating systems support two-part file names.
  - Parts are separated by a period (.)
  - Format: `<file name>.<extension>`
  - Extension indicates something about the file.
- Not all operation systems are aware of extensions.
  - Unix or Linux does not depends on extensions.
  - But some applications may depend on extensions.

Files: Types

- 2 major types
  - Regular files – ones that contain user data. These can be either text (ASCII) or binary.
  - Directories – are special system files which are used to maintain the structure of the file system.
- In Unix, it also has
  - Character files – are used to model serial I/O devices such as terminals and printers

    `/dev/tty, /dev/lp, /dev/net`
  - Block files – are used to model disks

    `/dev/hd1, /dev/hd2`

Files: Attributes

- A file includes a set of other characteristics than just name and extension
- Some common attributes
  - Owner – current owner of the file
  - Creator – the person who creates the file
  - Protection – who can access and who can't access
  - Size – length of the file in number of bytes
  - Read-only flag – can it be modified or not
  - Hidden flag – display or not when listed
  - Archive flag – to be backup or not
  - Last modified date, created date, etc.

Files: Access

- Sequential Access
  - Read all the data starting from the beginning
  - Used in early days with magnetic tapes
  - Example: simple text files
- Random Access
  - Can read the data in a file out of order
  - Were possible with the introduction of magnetic disks
  - Example: Data bases, movies

Files: Operations$_1$

- File systems allow operations to store and retrieve data from files
  - Create – create a new file with no data and set initial attributes
  - Delete – remove the file from system and free up disk space
  - Open – gain access to a file
  - Read – return a sequence of bytes from a file
  - Write – replace a sequence of bytes in a file and/or append to the end
  - Close – relinquish access to a file

Files: Operations$_2$

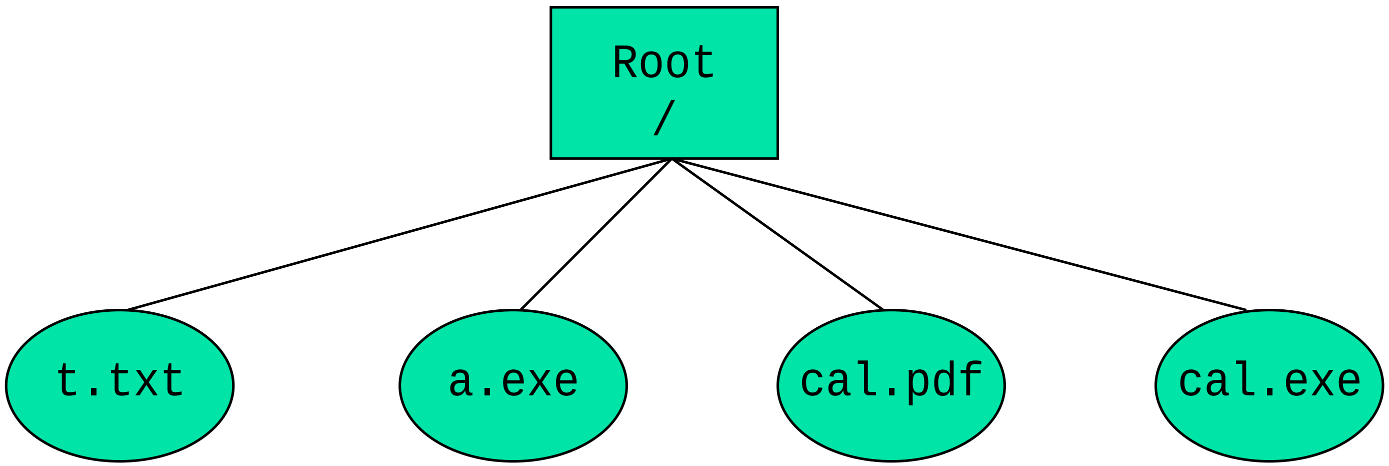
- Seek – reposition *file pointer* for subsequent reads and writes; used in random access
- Get attributes – get the attributes of a file
- Set attributes – set the attributes of a file
- Rename – change the name or the extension of a file

Directories

- Used to organize or keep track of files.
- Are also called folders.
  - DOS, UNIX and Linux call them as directories.
  - Windows call them as folders.
- Most operating systems consider directories as files.

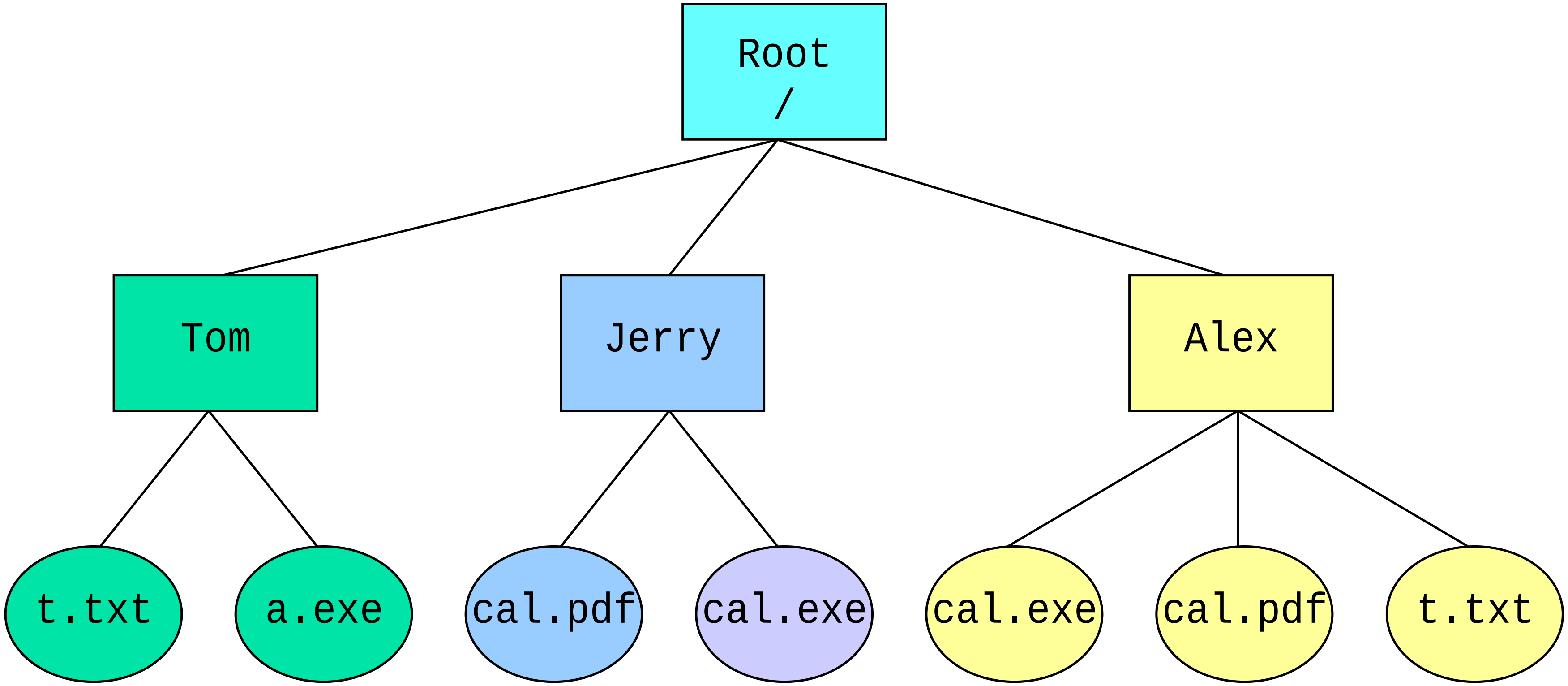
Directories: Single Level System

- Simplest form of directory system where a single directory contain all the files
- This single directory is called the root.
- Problem – in a multi-user system, it can't have files with the same name



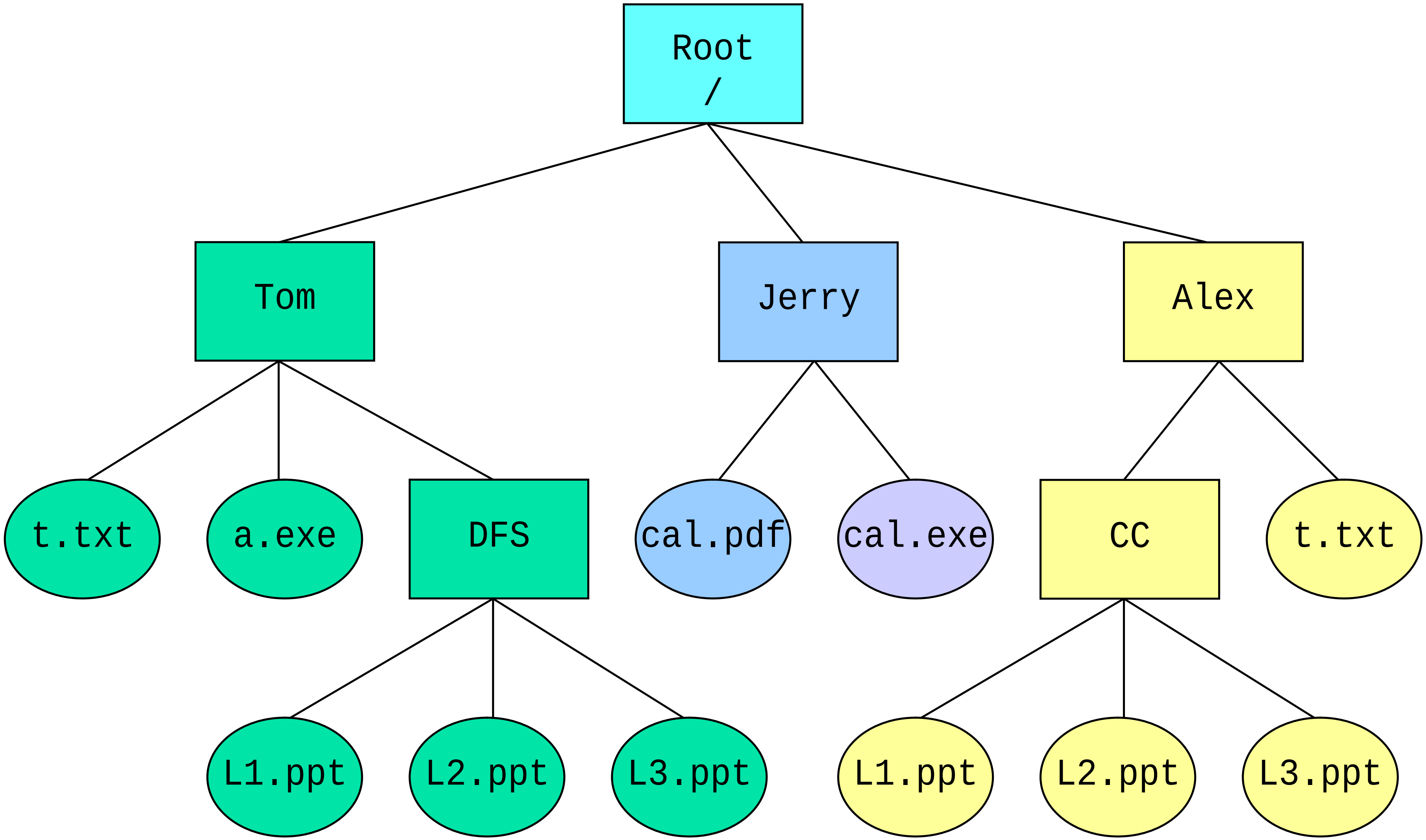
Directories: Two Level System

- To avoid the conflict, each user is given a separate directory.



Directories: Hierarchical Structure

- Two level directory structure is not enough when users want to manage their own files.
- Almost all the commercial operating systems support multiple directory levels.
- However, CD-ROM file system has a limit in number of levels in the hierarchy.
  - 8 levels, including the root directory, in the ISO 9660 file system.


Root
/
Tom
Jerry
Alex
t.txt
a.exe
DFS
cal.pdf
cal.exe
CC
t.txt
L1.ppt
L2.ppt
L3.ppt
L1.ppt
L2.ppt
L3.ppt

Directory Considerations

- *Efficiency* – locating a file quickly.
- *Naming* – convenient to users.
  - Separate users can use the same name for separate files.
  - The same file can have different names for different users.
  - Names need only be unique within a directory
- *Grouping* – logical grouping of files by properties
  - e.g., all Java programs, all games, …

Directories: Operations

- Create – create a new directory
- Delete – delete an existing directory
- List – enumerate directory entries
- Lookup – find an existing entry
- Rename – change the name of the directory
- Link – allow files to appear in more than one directories; related to file sharing.

Directories: Path Name$_1$

- When files are in a directory tree, there should be a mechanism to name them.
- Absolute path names
  - Path from the root to the directory
    `/Tom/DFS/L1.ppt`
- Relative path names
  - Relative to the current working directory
  - If currently in `/Tom/DFS` directory, the path name is
    `L1.ppt`
  - If currently in `/Tom` directory, the path name is
    `DFS/L1.ppt`

Directories: Path Name$_2$

- Regardless of the current working directory, absolute path names will always work.
- There are two special entries in each directory
  - . (dot) – refers to the current working directory
  - .. (double dot/dotdot) – refers to the parent directory
  - Examples: If currently in `/Tom` directory

    ```
    ./DFS/L1.ppt
    ../Jerry/cal.exe
    ```

Path Name Translation

- Assume that I want to open "/home/lauer/foo.c"

```
fd = open("/home/lauer/foo.c", O_RDWR);
```

  - Opens directory **`"/"`** – the root directory is in a known place on disk
  - Search root directory for the directory home and get its location
  - Open **`home`** and search for the directory **`lauer`** and get its location
  - Open **`lauer`** and search for the file **`foo.c`** and get its location
  - Open the file **`foo.c`**
  - The process needs the appropriate permissions at every step.
- It spends a lot of time walking down directory paths.
  - This is why **`open`** calls are separate from other file operations.
  - File System attempts to cache prefix lookups to speed up common searches.
  - Once open, file system caches the metadata of the file.

Outline

- Files and Directories
- Implementation Issues
- Example File Systems

Implementation of Files

- Files are stored as blocks on the disk.
- Need to keep track of where a file is located on the disks.
  - Map *file* abstraction to *physical* disk blocks.
- Goals
  - Efficient in time, space, use of disk resources
  - Fast enough for application requirements
  - Scalable to a wide variety of file sizes
    - Many small files (< 1 page)
    - Huge files (100's of gigabytes, terabytes, spanning disks)
    - Everything in between

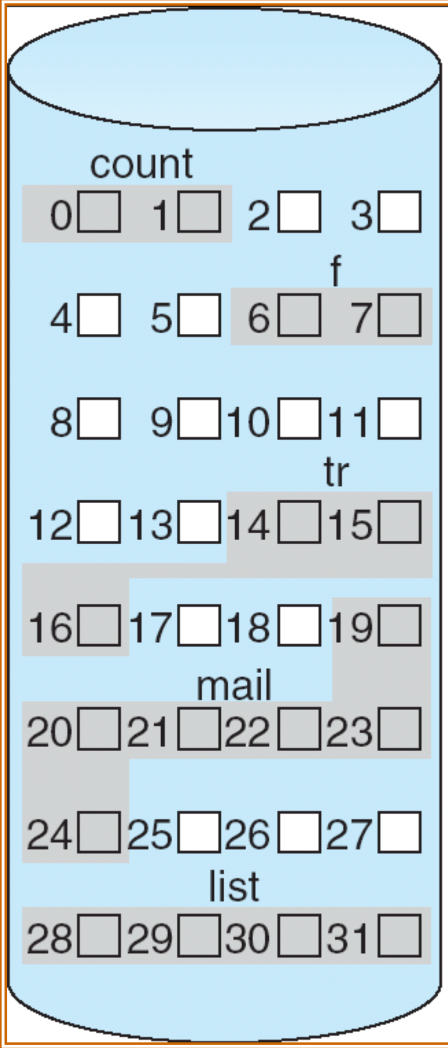
File Allocation Schemes

- Contiguous
  - Blocks of file stored in consecutive disk sectors
  - Directory points to first entry
- Linked
  - Blocks of file scattered across disk, as linked list
  - Directory points to first entry
- Indexed
  - Separate index blocks contain pointers to file blocks
  - Directory points to index blocks

Contiguous Allocation

- Ideal for large and static files
  - Static Databases, OS code
  - Multi-media video and audio
  - CD-ROM, DVD-ROM
- Simple address calculation
  - Directory entry points to first block
  - File block *i* ⇔ disk block address
- Fast multi-block reads and writes
  - Minimize seeks between blocks

Contiguously Allocated Files



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

Contiguous Allocation: File Creation

- Search for an empty sequence of blocks
  - First-fit
  - Best-fit
- Prone to fragmentation when …
  - Files come and go
    - For example, a new file needs 7 contiguous blocks.
  - Files change size
    - For example, the file `tr` changes its size to 6 blocks.

Contiguous Allocation – *Extents*

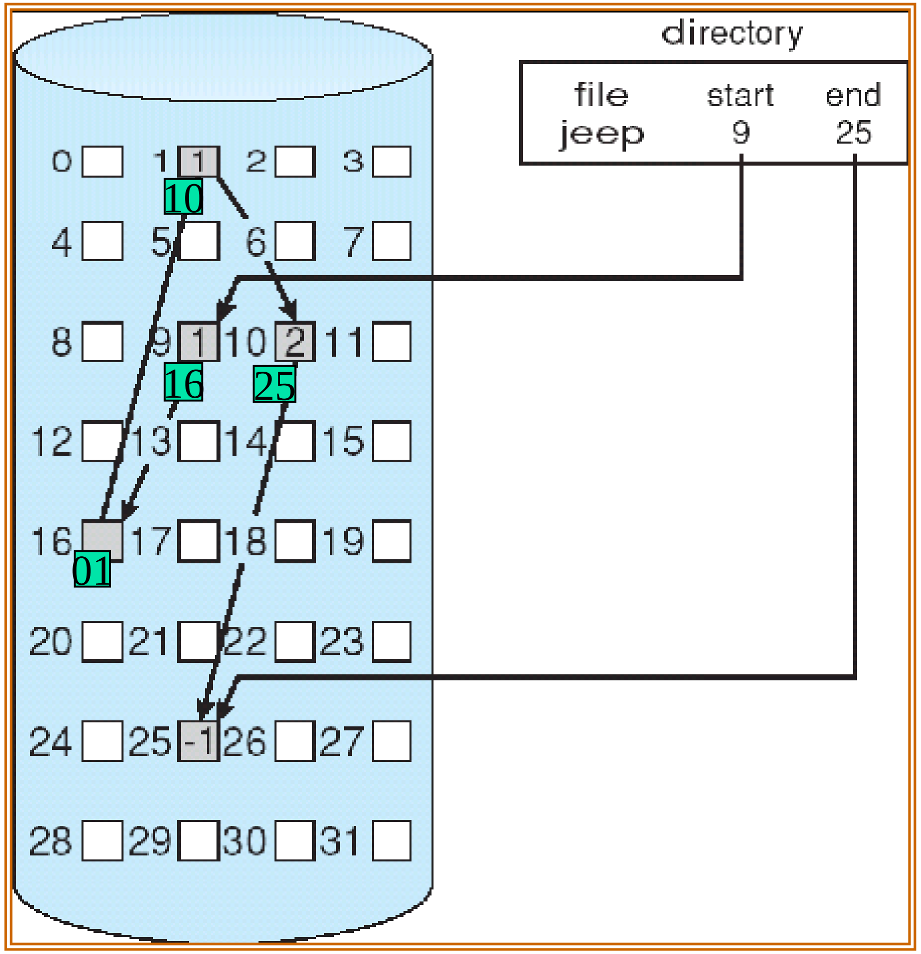
- *Extent:* a contiguously allocated subset of a file
- Directory entry points to
  - (For file with one extent) the extent itself
  - (For file with multiple extents) pointer to an *extent block* describing multiple extents
- Advantages
  - Speed, ease of address calculation of contiguous file
  - Avoids (some of) the fragmentation issues
  - Can be adapted to support files across multiple disks

Contiguous Allocation – *Extents*

- Disadvantages
  - Too many extents $\Rightarrow$ degenerates to *indexed* allocation
    - As in Unix-like systems, but not so well
- Popular in 1960s & 70s
- Currently used for large files in NTFS
- Rarely mentioned in textbooks

Linked Allocation

- Blocks scattered across disk
- Each block contains pointer to next block
- Directory points to first and last blocks
- Block header:
  - Pointer to next block
  - ID and block number of the file

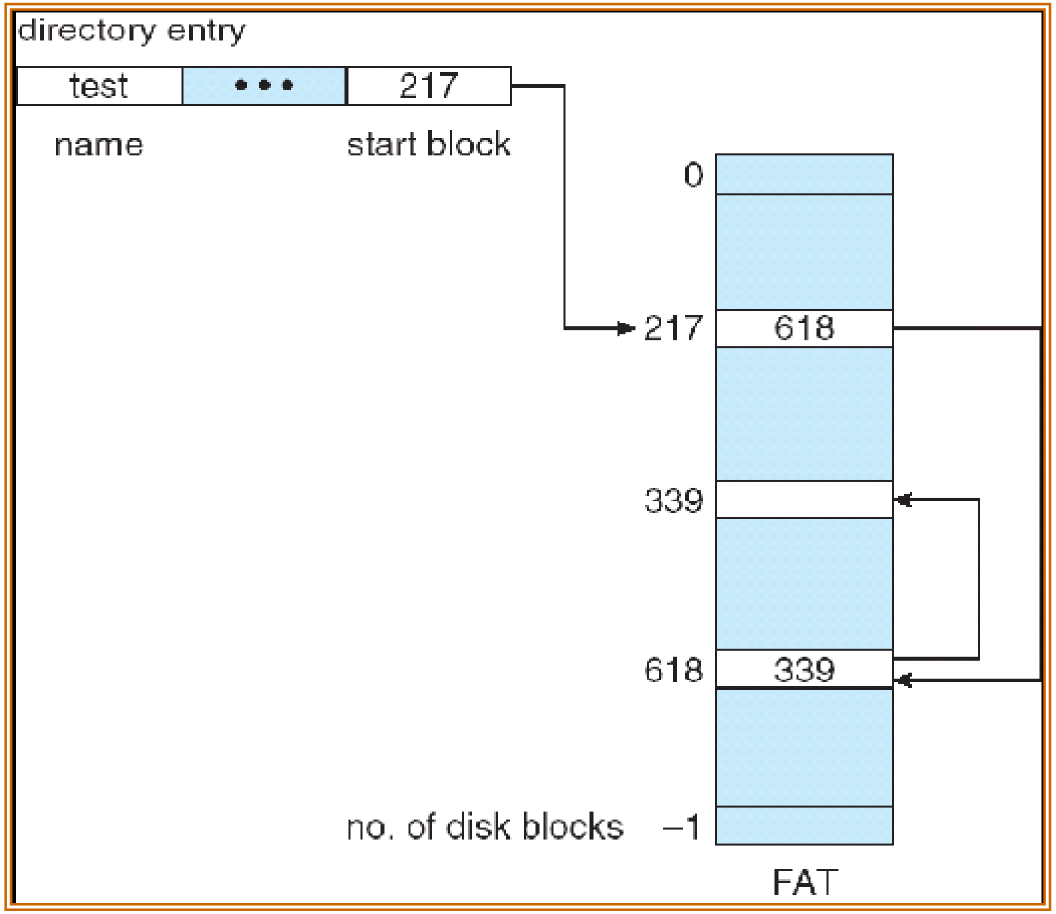


Linked Allocation

- Advantages
  - No space fragmentation!
  - Easy to create and extend files
  - Ideal for lots of small files
- Disadvantages
  - Lots of disk arm movement
  - Space taken up by links
  - Sequential access only!

Linked Allocation – FAT

- Instead of link on each block, put all links in one table
  - the *F*ile *A*llocation *T*able — i.e., *FAT*
- One entry per physical block in disk
  - Directory points to first & last blocks of file
  - Each block points to next block (or *EOF*)

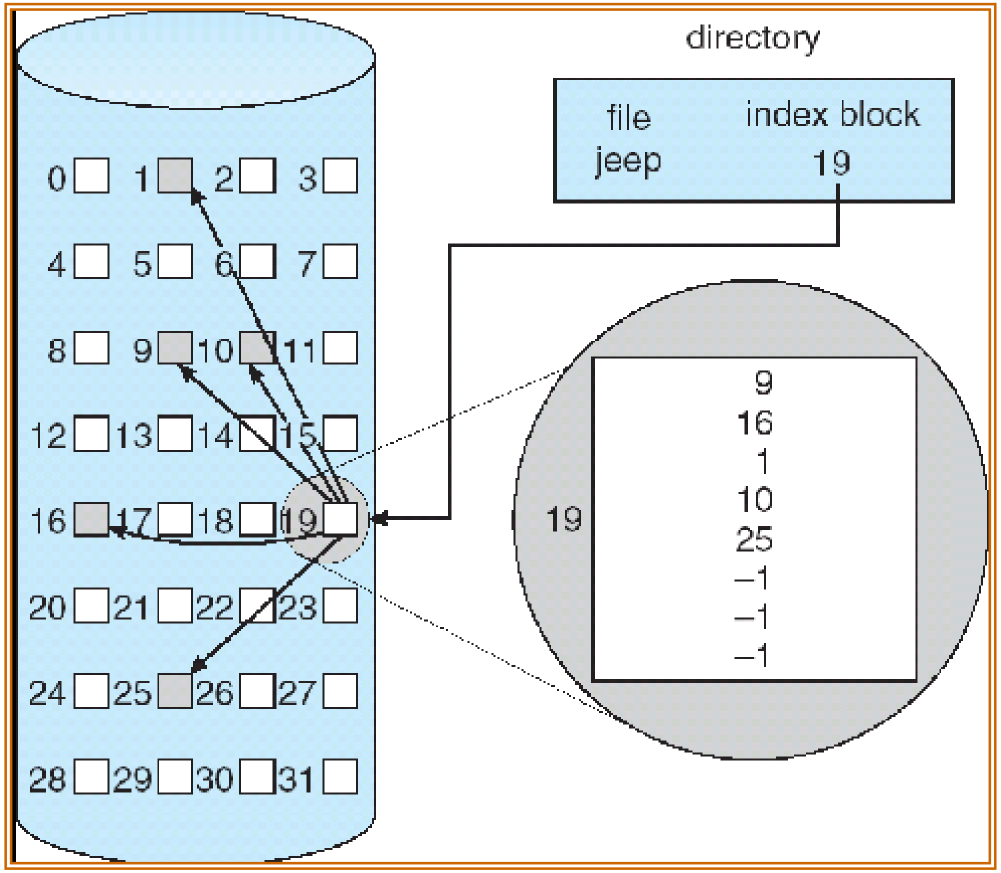


FAT File Systems

- 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

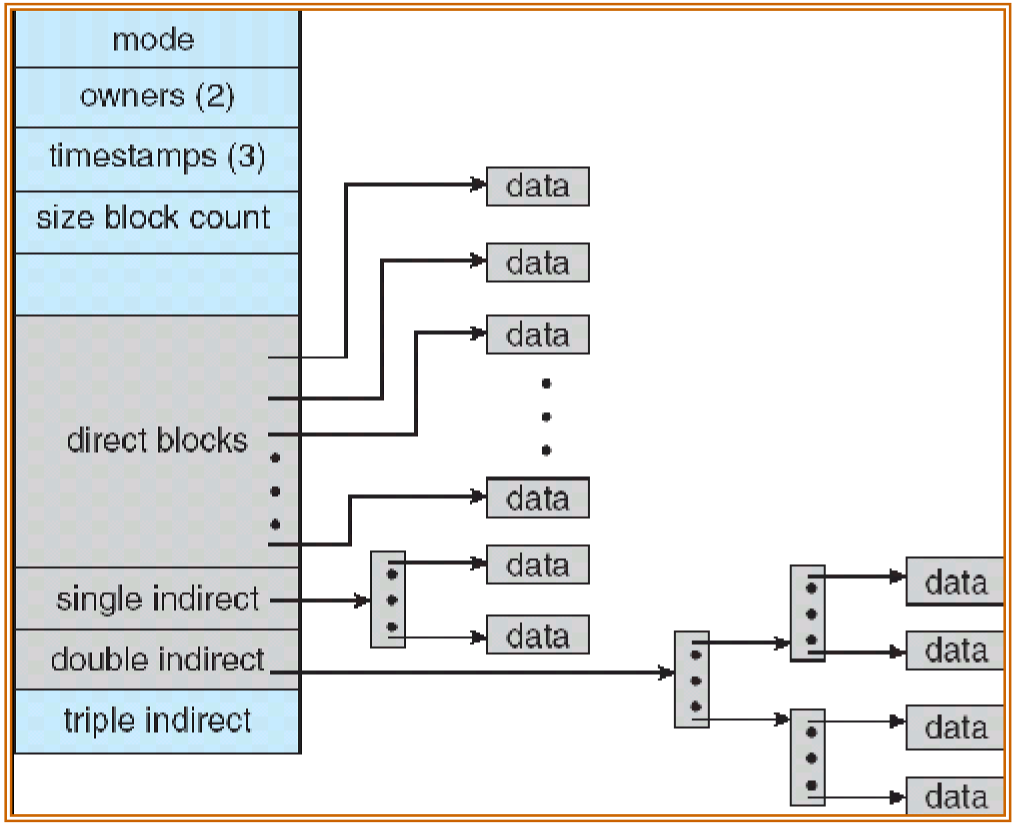
Indexed Allocation

- *i-node:*
  - Part of file metadata
  - Data structure lists the address of each block of a file
- Advantages
  - True random access
  - Only i-nodes of open files need to be cached
  - Supports small and large files



Unix/Linux i-nodes

- *Direct blocks:*
  - Pointers to first *n* blocks
- *Single indirect table:*
  - Extra block containing pointers to blocks *n+1 .. n+m*
- *Double indirect table:*
  - Extra block containing single indirect blocks
- …



Indexed Allocation

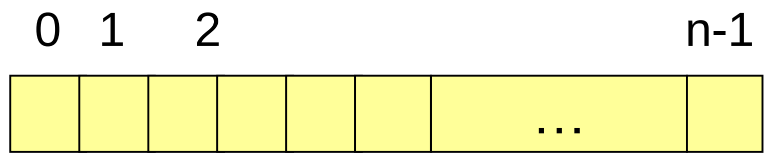
- Access to *every* block of file is via *i-node*
- Disadvantage
  - Not as fast as contiguous allocation for large databases
    - Requires reference to *i-node* for every access

      vs.
    - Simple calculation of file block to disk block address

Free Block Management

- Bitmap
  - Very compact on disk
  - Expensive to search
  - Supports contiguous allocation
- Free list
  - Linked list of free blocks
    - Each block contains pointer to next free block
  - Only head of list needs to be cached in memory
  - Very fast to search and allocate
  - Contiguous allocation vary difficult

Free Block Management: Bit Vector



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

Free block number calculation

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

Free Block Management: Bit Vector

- Bit map
  - Must be kept both in memory and on disk
  - Copy in memory and disk may differ
  - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk.
  - How about bit[*i*] = 0 in memory and bit[*i*] = 1 on disk? Is it ok?

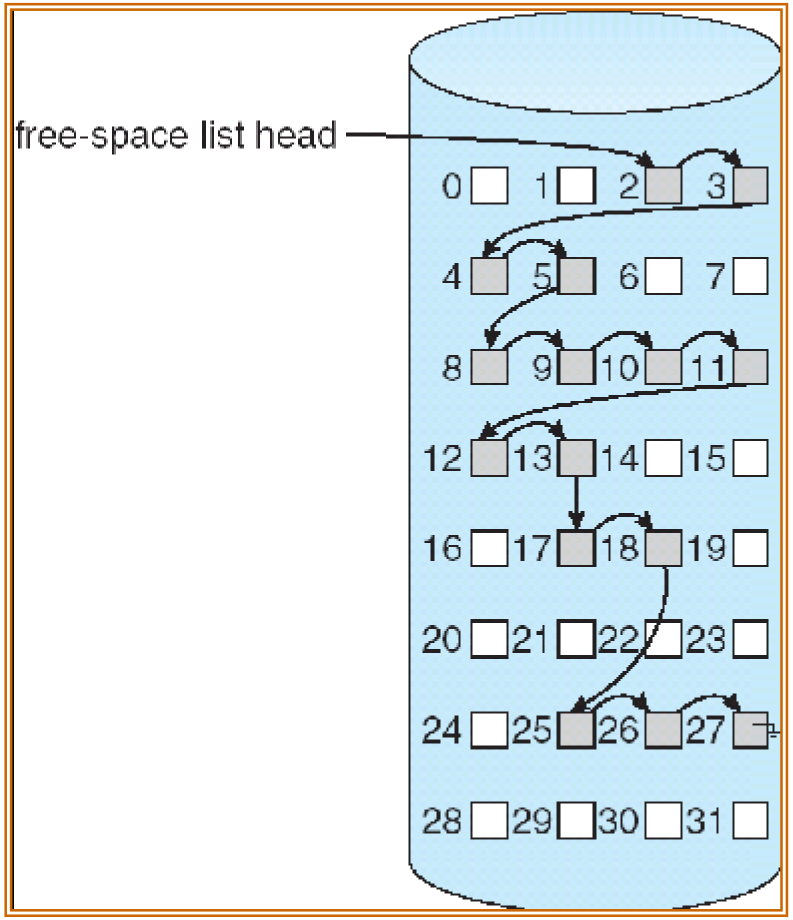
Free Block Management: Bit Vector

- Solution:
  - Set bit[*i*] = 1 on disk
  - Allocate block[*i*]
  - Set bit[*i*] = 1 in memory
  - Similarly for set of contiguous blocks

- Potential for lost blocks in event of crash!
  - Discussion – How do we solve this problem?

Free Block Management: Linked List

- Linked list of free blocks
  - Not in order!
- Cache first few free blocks in memory
- Head of list must be stored both
  - On disk
  - In memory
- Each block must be written to disk when freed
- Potential for losing blocks?



Bad Block Management

- Bad blocks on disks are inevitable
  - Part of manufacturing process (less than 1%)
  - Most are detected during formatting
  - Occasionally, blocks become bad during operation
- Manufacturers typically add extra tracks to disks
  - Physical capacity = (1 + x) * rated_capacity
- Who handles bad blocks?
  - Disk controller: Bad block list maintained internally
    - Automatically substitutes good blocks
  - Formatter: Re-organize track to avoid bad blocks
  - OS: Bad block list maintained by OS, bad blocks never used

Bad Block Management in Contiguous Allocation File Systems

- Bad blocks *must* be concealed
  - Foul up the block-to-sector calculation
- Methods
  - Look-aside list of bad sectors
    - Check each sector request against hash table
    - If present, substitute a replacement sector behind the scenes
  - Spare sectors in each track, remapped by formatting
- Handling
  - Disk controller, invisible to OS
  - Lower levels of OS; invisible to most of file system or application

Bad Block Management in Linked and FAT Systems

- In OS:– format all sectors of disk
  - Don’t reserve any spare sectors
- Allocate bad blocks to a hidden file for the purpose
  - If a block becomes bad, append to the hidden file
- Advantages
  - Very simple
  - No look-aside or sector remapping needed
  - Totally transparent without any hidden mechanism

Implementation of Directories

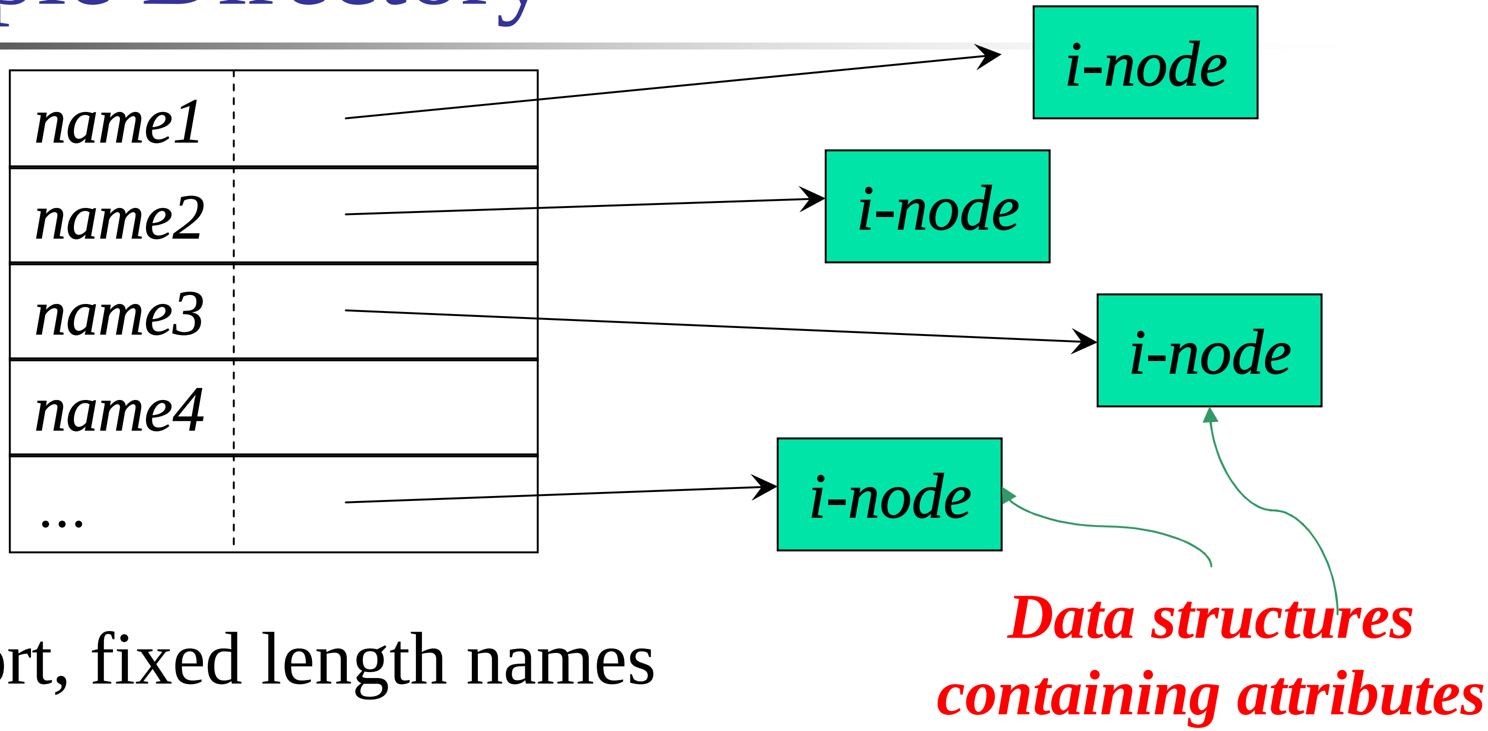
- A list of [name, information] pairs
  - Must be scalable from very few entries to very many
- *Name:*
  - User-friendly, variable length
  - Any language
  - Fast access by name
- *Information:*
  - File metadata (itself)
  - Pointer to file metadata block (or i-node) on disk
  - Pointer to first & last blocks of file
  - Pointer to extent block(s)
  - …

Very Simple Directory

name1	attributes
name2	attributes
name3	attributes
name4	attributes
...	...

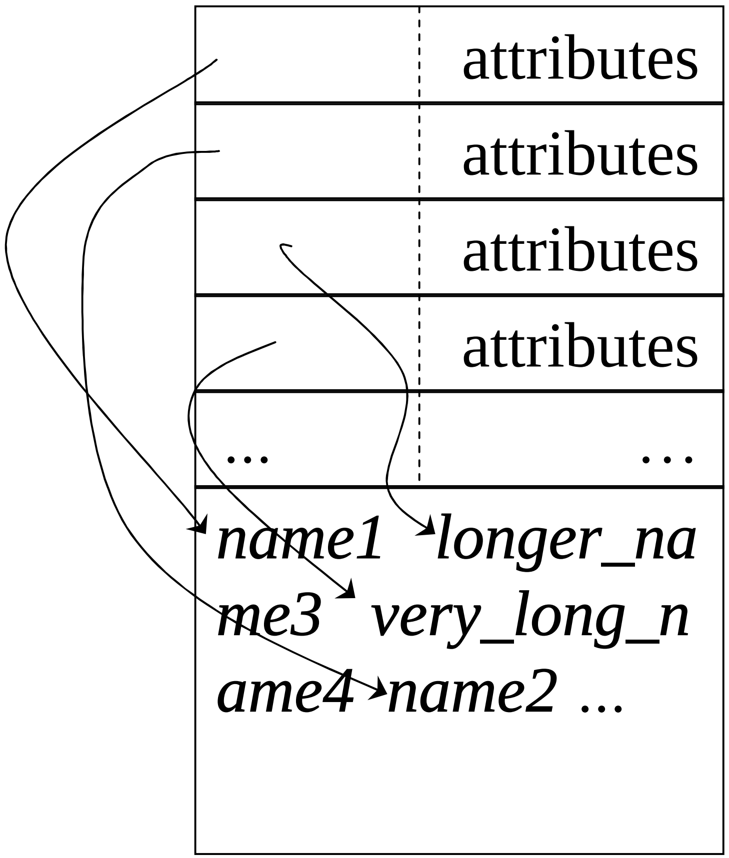
- Short, fixed length names
- Attribute & disk addresses contained *in* directory
- MS-DOS, etc.

Simple Directory



- Short, fixed length names
- Attributes in separate blocks (e.g., *i-nodes*)
  - Attribute pointers are disk addresses (or *i-node* numbers)
- Older Unix versions, MS-DOS, etc.

More Interesting Directory



- Variable length file names
  - Stored in heap at end
- Modern Unix, Windows
- Linear or logarithmic search for name
- Compaction needed after
  - Deletion, Rename

Very Large Directories

- Hash-table implementation
- Each hash chain like a small directory with variable-length names
- Must be sorted for listing

Outline

- Files and Directories
- Implementation Issues
- Example File Systems

Scalability of File Systems

- *Question:* How large can a file be?
- *Answer:* limited by
  - Number of bits in length field in file metadata
  - Size & number of block entries in FAT or *i-node*

- *Question:* How large can file system be?
- *Answer:* limited by
  - Number of bits in length field in file system metadata
  - Size & number of block entries in FAT or *i-node*

MS-DOS & Windows

- *FAT-12* (primarily on floppy disks):
  - 4096 512-byte blocks
  - Only 4086 blocks usable!
- *FAT-16* (early hard drives):
  - 64 K blocks; block sizes up to 32 K bytes
  - 2 GBytes max per partition, 4 partitions per disk
- *FAT-32* (Windows 95)
  - 2^{28} blocks; up to 2 TBytes per disk
  - Max size *FAT* requires 2^{32} bytes in RAM!

MS-DOS File System

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

- Maximum partition for different block sizes
- The empty boxes represent forbidden combinations

System V File System

- The file system resides on a single logical disk or partition
- A partition can be viewed as a linear array of blocks
  - block represents the granularity of space allocation for files
  - a disk block is 512 bytes * some power of 2
  - physical block number identifies a block on a given disk partition
  - physical block number can be translated into physical location on a partition

System V: File System Layout

B	S	inode list	data blocks

- Boot area
  - Code required to bootstrap the operating system
- Superblock
  - Attributes and metadata of the file system itself
- inode list
  - a linear array of inodes
- data blocks
  - data blocks for files and directories, and indirect blocks

System V: Superblock

- One Superblock per file system
- It contains metadata about file system
  - Size in blocks of the file system
  - Size in blocks of the inode list
  - Number of free blocks and inodes
  - Free block list
  - Free inode list
- The kernel reads the superblock and stores it in memory when mounting the file system

System V: Inode

- Each file has an unique inode associated with it.
- Inode contains metadata of the file.
- On-disk inode refers to inode stored in disk within the inode list.
- In-core inode refers to inode stored in memory when a file is open.

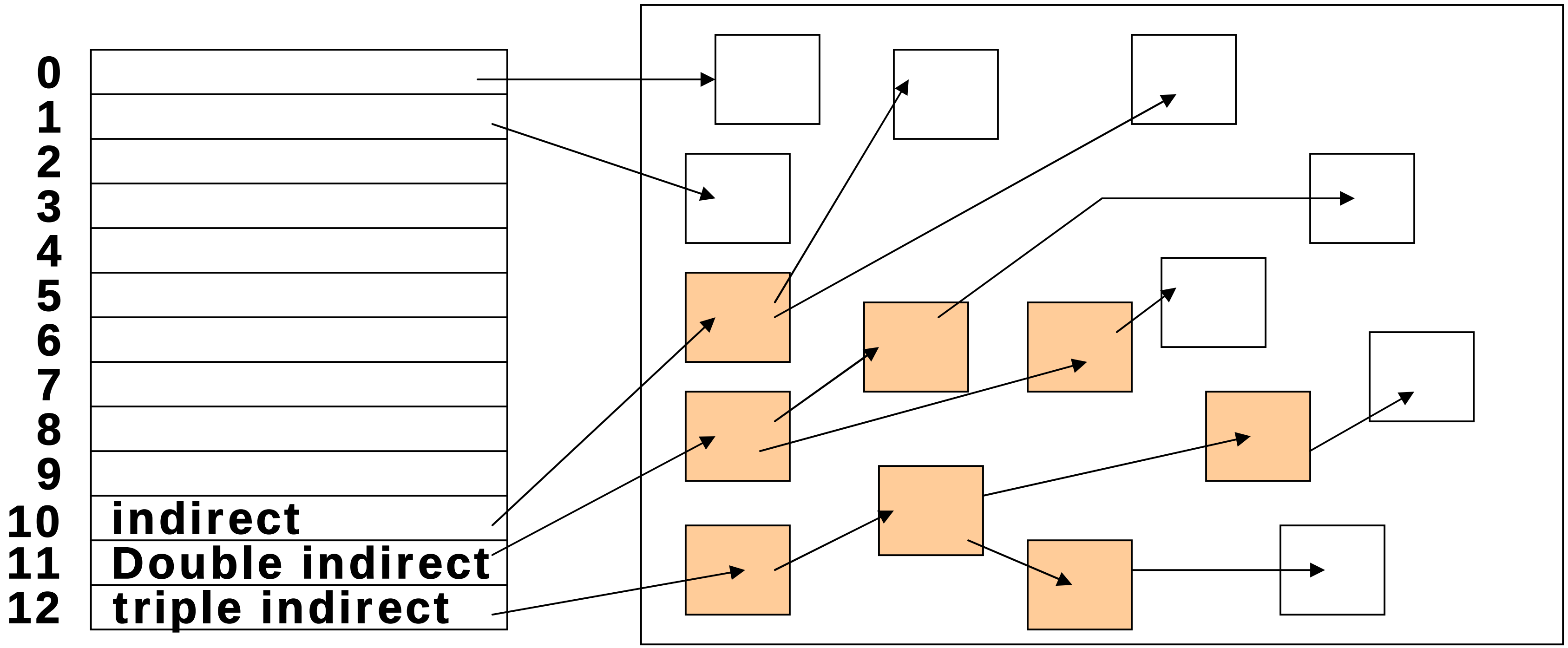
System V: On-disk inode

- The size of on-disk inode is 64 bytes

Field	Size	Description
di_mode	2	File type, permissions
di_uid	2	Owner UID
di_gid	2	Owner GID
di_size	4	Size in bytes
di_addr	39	Array of block addresses
:	:	:

System V: On-disk inode

- Unix files are not stored in contiguous blocks.
- File system need to maintain a map of the disk location of every block of the file.



System V: In-core inode

- It contains all the fields of on-disk inode, and some additional fields, such as
  - The status of the in-core inode (whether the inode is locked, which process is waiting, etc.)
  - The logical device number containing the file
  - The inode number of the file
  - Pointers to keep the inode on a free list
  - Pointers to keep the inode on a hash queue.
  - Block number of last block read.

System V: Inode Operations

- Inode lookup: lookuppn()&s5lookup()
  - translates a pathname and returns a pointer to the vnode of the desired file
- allocate inode: iget()
  - read an inode from disk into memory by inode number or initialize an empty inode if not found
- release inode: iput()
  - kernal writes the inode to disk if the in-core copy differs from the disk copy

System V: File Operations

- Read and write system calls use the following arguments
  - File descriptor, user buffer address, count of number of byte transferred
- Offset is obtained from the opened file object
- Offset is advanced to the number of byte transferred
- For random I/O “lseek” is used to set the offset to desired location
- Kernel verifies the file mode and puts an exclusive lock on the inode for serialized access
- File read: s5read()

System V: Directories

- A file system is organized as a hierarchy of directories.
- It starts from a single directory called root (represented by a /).
- A directory is a file containing list of files and subdirectories.
- It has fixed size records of 16 bytes, each which contains
  - a 14-byte filename
  - a 2-byte inode number ($2^{16} = 65536$ files), acts as a pointer to where the system can find info about the file.

System V: Directories

- 0 inode number means the file no longer exists.
- The directory itself and its parent directory are in the first two entries.

73	.
38	..
9	File1
0	Deleted file
110	Subdirectory1
65	File2

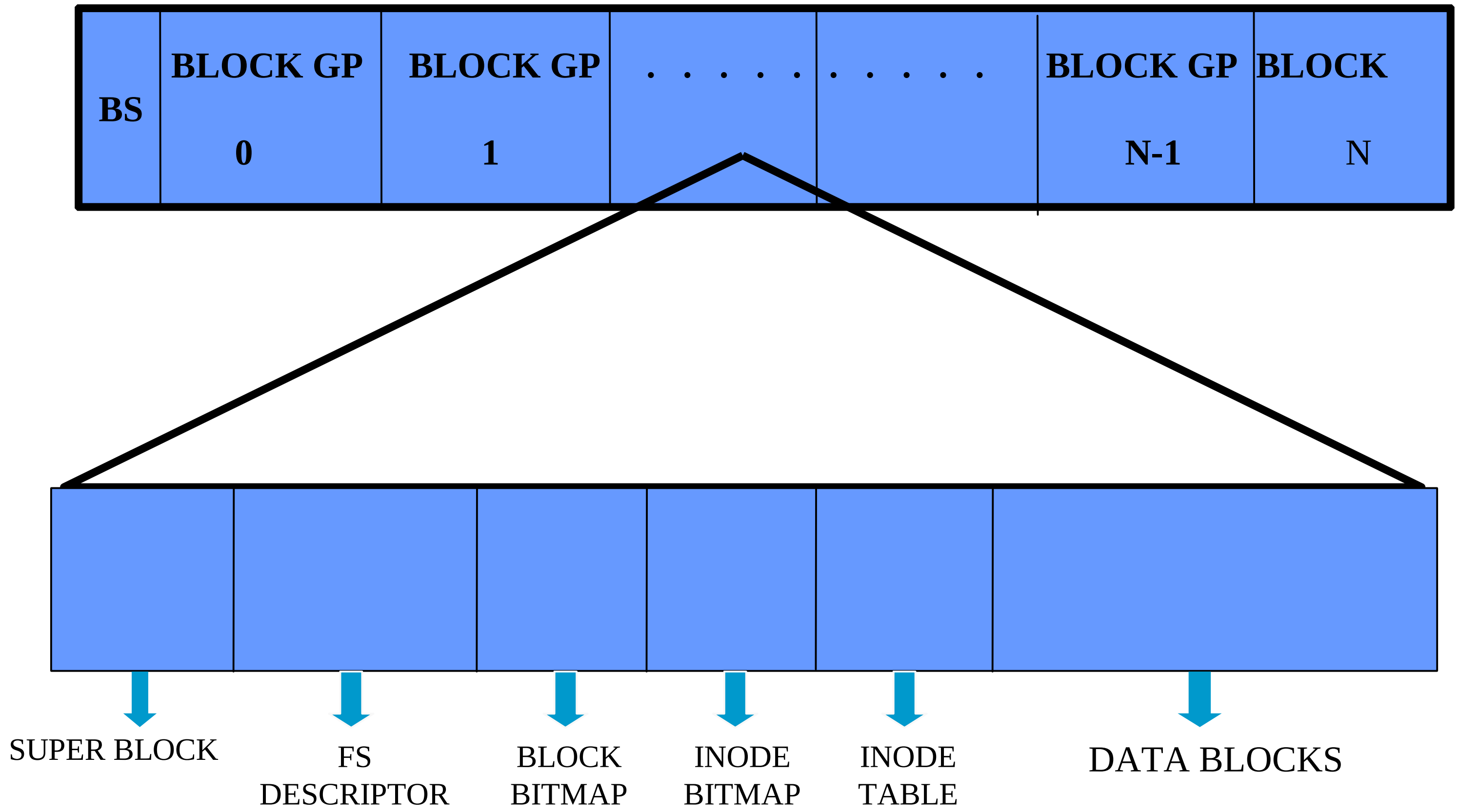
System V: Summary

- Simple design
- Single superblock can be corrupted
- Grouping of inode in the beginning requires long seek time between inode read and file access
- Fixed block size wastes space
- Filename is limited to 14 characters
- Number of inodes are limited to 65535

The ext2 File System

- The Second Extended File system was devised (by Rémy Card) as an extensible and powerful file system for Linux.
- It is also the most successful file system so far in the Linux community and is the basis for all of the currently shipping distributions.
- Due to this, it is extremely well integrated into the kernel, with good performance enhancements.

Ext2: File System Layout



Ext2: File System Layout

- The Boot Sector block is optional, not required if you do not want to make this partition bootable.
- Each block group contains
  - a redundant copy of crucial file system control information (superblock and the file system descriptors)
  - a part of the file system (a block bitmap, an inode bitmap, a piece of the inode table, and data blocks)
- Having multiple block groups helps improves reliability (since backups of the superblock are there) and even speeds up access as the inode table is near the data blocks – reduced seek time for data blocks.

Ext2: Block Group

- Superblock – The file system header, identifies the file system and provides relevant information.
- FS descriptor – Pointers to the bitmaps and table in the block group.
- Block bitmap – Block usage information, tells which blocks in the block group are empty or used
- Inode bitmap – Inode usage information
- Inode table – Table of the inodes. Each inode provides information about a file.
- Data blocks – blocks where the data is stored!

Ext2: Superblock

- The Superblock contains a description of the basic size and shape of this file system.
- System keeps multiple copies of the Superblock in many Block Groups.
- It holds the following information :
  - Magic Number : *0xef53* for the current implementation.
  - Revision Level : for checking compatibility
  - Mount Count and Maximum Mount Count : to ensure that the file system is periodically checked
  - Block Group Number : The Block Group that holds this copy of Superblock.

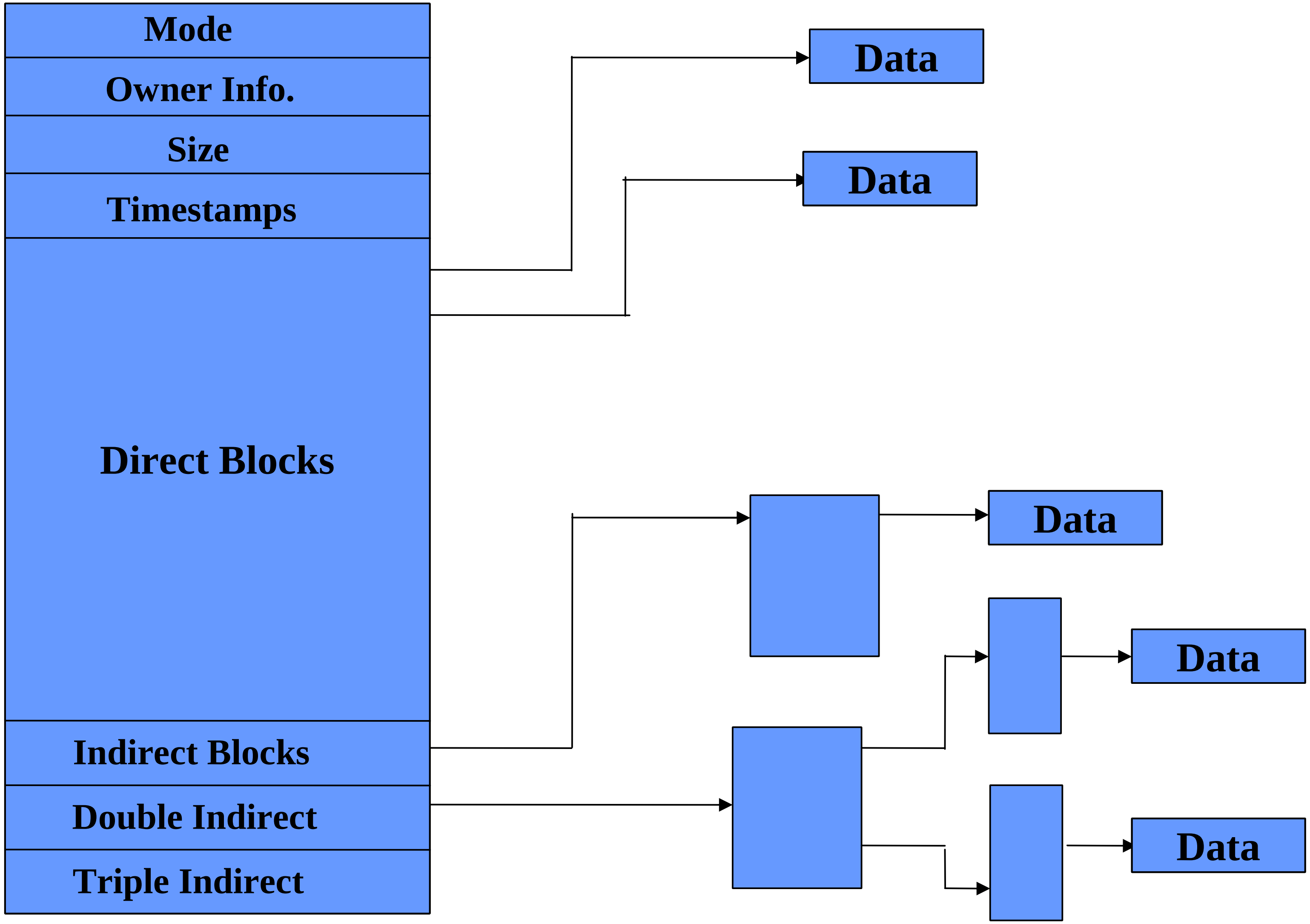
Ext2: Superblock

- Block Size : size of blocks for the file system in bytes.
- Blocks per Group : Number of blocks in a group – fixed when file system is created.
- Free Blocks : Number of free blocks in the system – excludes the blocks reserved for root
- Free Inodes : Number of free Inodes in the system – again excludes inodes reserved for root
- First Inode : The first Inode in an EXT2 root file system would be the directory entry for the '/' directory.

Ext2: FS Descriptor

- The FS Descriptor contains the following:
  - Blocks Bitmap : block number of block allocation bitmap
  - Inode Bitmap : block number of Inode allocation bitmap
  - Inode Table : The block number of the starting block for the Inode table for this Block Group.
  - Free blocks count : number of free data blocks in the Group
  - Free Inodes count : number of free inodes in the Group
  - Used directory count : number of inodes allocated to directories

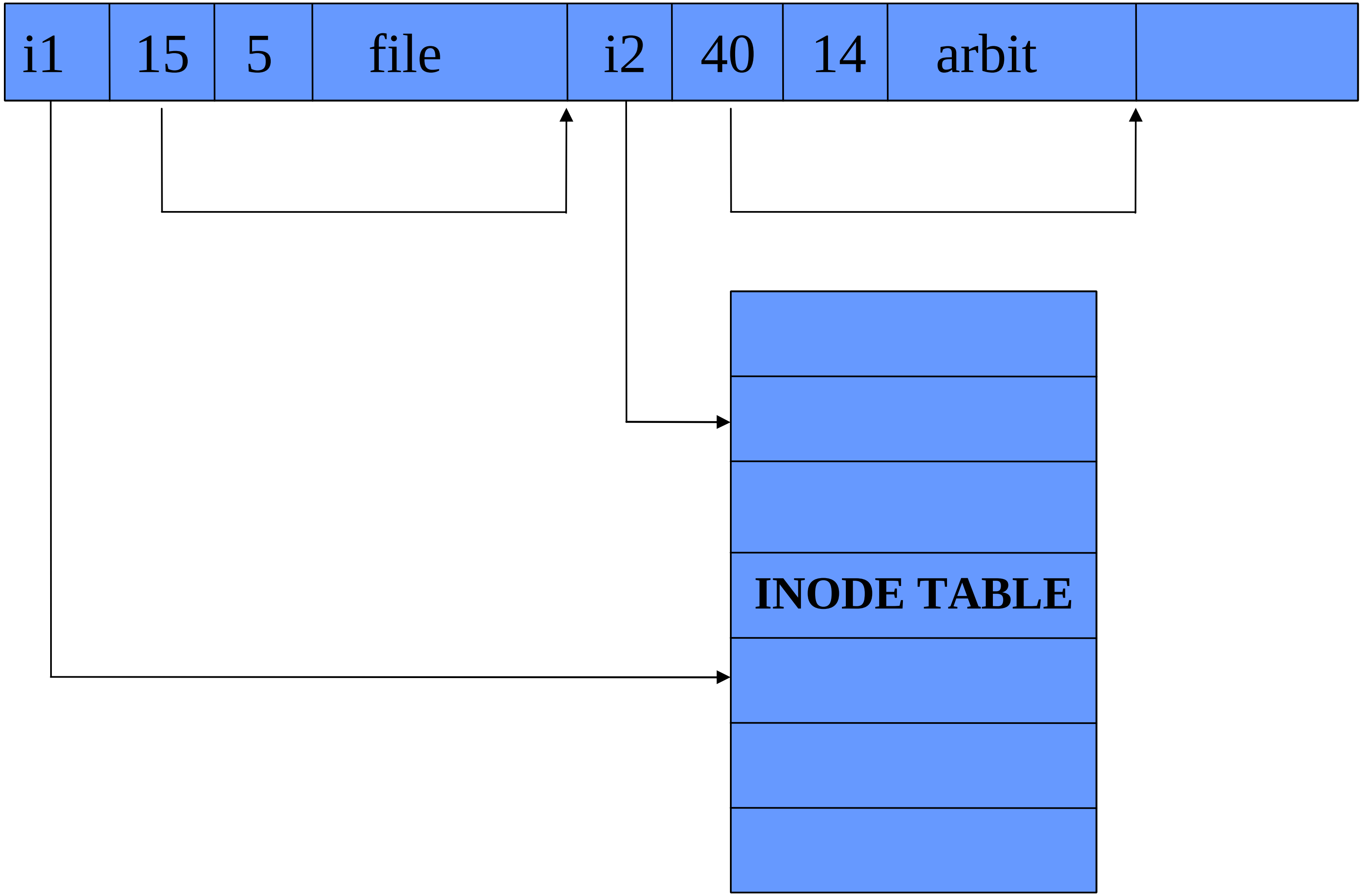
Ext2: Inode


Mode
Owner Info.
Size
Timestamps
Direct Blocks
Indirect Blocks
Double Indirect
Triple Indirect
Data
Data
Data
Data
Data

Ext2: Inode

- Direct/Indirect Blocks : Pointers to the blocks that contain the data that this Inode is describing.
- Timestamp : The time that this Inode was created and the last time that it was modified.
- Size : The size of this file in bytes.
- Owner info : This stores user and group identifiers of the owners of this file or directory
- Mode : This holds two pieces of information; what this inode describes and the permissions that users have on it.

Ext2: Directories



Mounting

```
mount –t type device pathname
```

- Attach *device* (which contains a file system of type *type*) to the directory at *pathname*
  - File system implementation for *type* gets loaded and connected to the device
  - Anything previously below *pathname* becomes hidden until the *device* is un-mounted again
  - The root of the file system on *device* is now accessed as *pathname*
- E.g.,

```
mount –t iso9660 /dev/cdrom /myCD
```

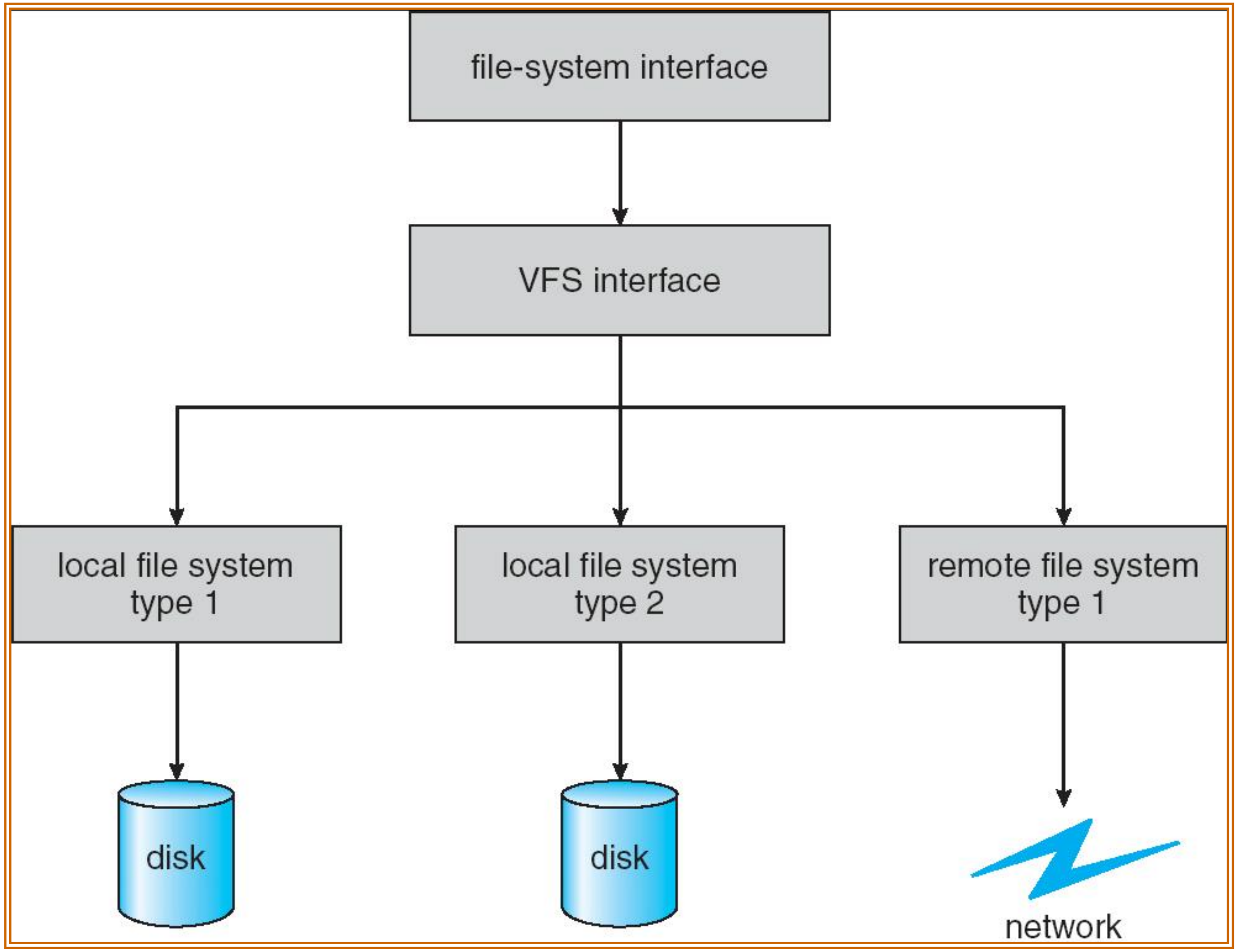
Mounting

- OS automatically mounts devices in mount table at initialization time
  - /etc/fstab in Linux
- Users or applications may mount devices at run time, explicitly or implicitly — e.g.,
  - Insert a floppy disk
  - Plug in a USB flash drive
- Type may be implicit in device
- Windows equivalent
  - Map drive

Virtual File Systems

- Virtual File Systems (VFS) provide object-oriented way of implementing file systems.
- VFS allows same system call interface to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.
- *Mounting:* formal mechanism for attaching a file system to the Virtual File interface.

VFS: Schematic View



Linux Virtual File System

- A generic file system interface provided by the kernel
- Common object framework
  - *superblock:* a specific, mounted file system
  - *i-node object:* a specific file in storage
  - *d-entry object:* a directory entry
  - *file object:* an open file associated with a process

Linux Virtual File System

- VFS operations
  - *super_operations:*
    - *read_inode*, *sync_fs*, etc.
  - *inode_operations:*
    - *create*, *link*, etc.
  - *d_entry_operations:*
    - *d_compare*, *d_delete*, etc.
  - *file_operations:*
    - *read*, *write*, *seek*, etc.

Linux Virtual File System

- Individual file system implementations conform to this architecture.
- May be linked to kernel or loaded as modules
- Linux kernel 2.6 supports over 50 file systems in official version
  - E.g., minix, ext, ext2, ext3, iso9660, msdos, nfs, smb, …