Virtualization and File System Management

Module 6

Dr. Naveenkumar J

Associate Professor,

PRP- 217 - 4

Directory

- ❑ A directory is essentially a container that holds information about a collection of files.

- ❑ In more advanced structures, it can also contain other directories.

Directory Structures Need

Logical Organization

- It allows for the logical grouping of related files. For example, you can create a directory to hold all your work-related documents, keeping them separate from personal files.

Efficiency

- By organizing files into groups, it becomes faster and easier to find a specific file when you need it.

Convenient Naming

- Simple directory structures require every file in the system to have a unique name, which is impractical with many files or users. More advanced structures, like the two-level or tree structure, solve this by allowing different users or different directories to contain files with the same name.

Sharing

- A directory structure enables files to be shared across different locations or between users without making multiple copies. This is achieved by creating links to the original file in other directories

Directory Structures Type

- **Single-Level Directory**
  - This is the most straightforward directory structure where all files are stored in a single, common directory. Think of it as one large folder for everything.
  - The operating system maintains a single list of all files. When a new file is created, it is added to this directory.
  - To access a file, the system searches this one directory.

Directory Structures Type

- **Single-Level Directory** - Merits:
  - **Simplicity** It is the easiest structure to understand and implement.
  - **Easy Access** Finding and accessing files is straightforward since everything is in one place.
  - **Simple File Operations** Tasks like creating, deleting, and renaming files are very easy.

Directory Structures Type

- **Single-Level Directory –** De-merits:

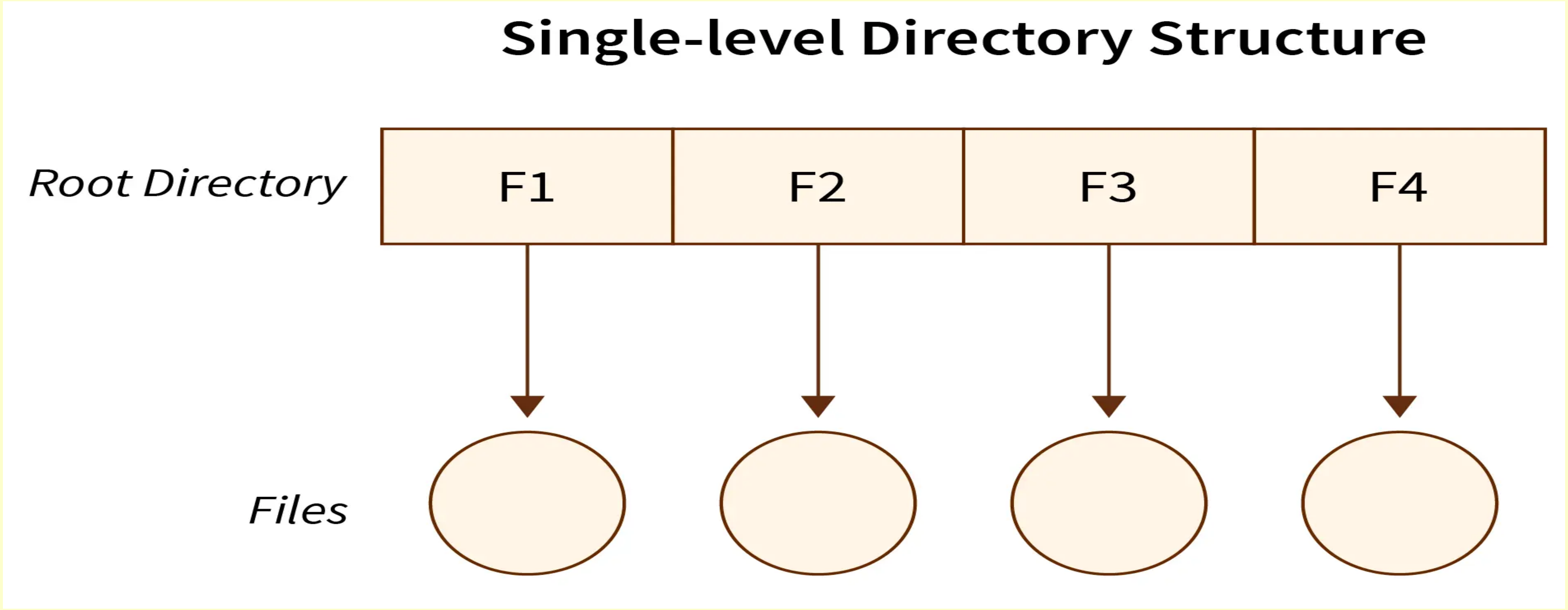
  - **Naming Conflicts** No two files can have the same name. This becomes a significant problem as the number of files increases or when multiple users are on the system.

  - **Lack of Organization** It is difficult to group related files, making the system cluttered and hard to manage as it grows.

  - **Security Issues** There is no way to restrict access to files, as all files are in a shared directory.

Directory Structures Type

- **Single-Level Directory**



Directory Structures Type

❑

```
Root Directory
 ├── file1.txt
 ├── file2.doc
 ├── program.exe
 ├── data.csv
└── report.pdf
```

Directory Structures Type

- **Two-Level Directory**
  - To overcome the limitations of the single-level structure, the two-level directory creates a separate directory for each user.
  - There is a Master File Directory (MFD) that contains pointers to each user's individual User File Directory (UFD).
  - When a user logs in, the system accesses their specific UFD.
  - Any file operations are then confined to that user's directory. This means users can have files with the same name as other users because they exist in different directories.

Directory Structures Type

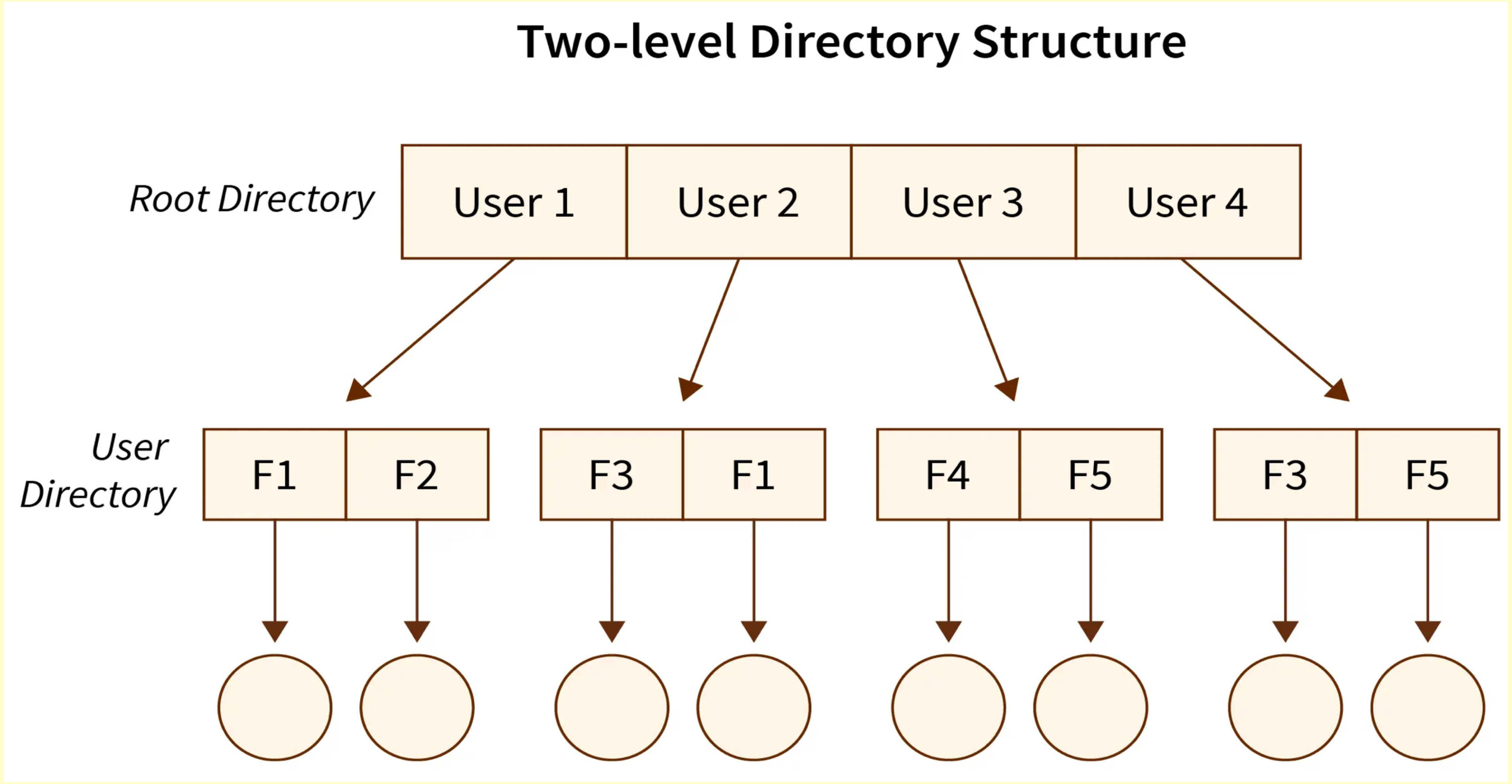
- ❑ **Two-Level Directory - Merits**
  - ❑ **No Naming Conflicts** Since each user has their own directory, naming collisions between users are eliminated.
  - ❑ **Improved Organization** Files are organized on a per-user basis, which makes searching and management easier for individual users.
  - ❑ **Enhanced Security** Users are isolated from each other, preventing them from accessing each other's files by default.

Directory Structures Type

- **Two-Level Directory - Demerits**
  - **No Collaboration** The isolation between users makes it difficult for them to cooperate on tasks or share files.
  - **Limited Organization for Users**: While it separates users, it doesn't help a single user organize their own files into subgroups. A user with many files will still have them all in one large list.

Directory Structures Type

- **Two-Level Directory**



Directory Structures Type

```
Master Directory (MFD)
 ├── User1 Directory
│     ├── file1.txt
│     ├── program.exe
│    └── data.csv
 ├── User2 Directory
│     ├── file1.txt (allowed - different directory)
│     ├── report.doc
│    └── notes.txt
└── User3 Directory
    ├── project.py
   └── results.csv
```

Directory Structures Type

- **Tree Directory Structure**
  - This structure extends the two-level directory into a hierarchy, where directories can contain not only files but also other directories (subdirectories).
  - This creates a tree-like organization.
  - Each user has a home directory and can create a complex tree of subdirectories to organize their files.
  - Files are accessed using a path that specifies the route from the root of the tree down to the file.

Directory Structures Type

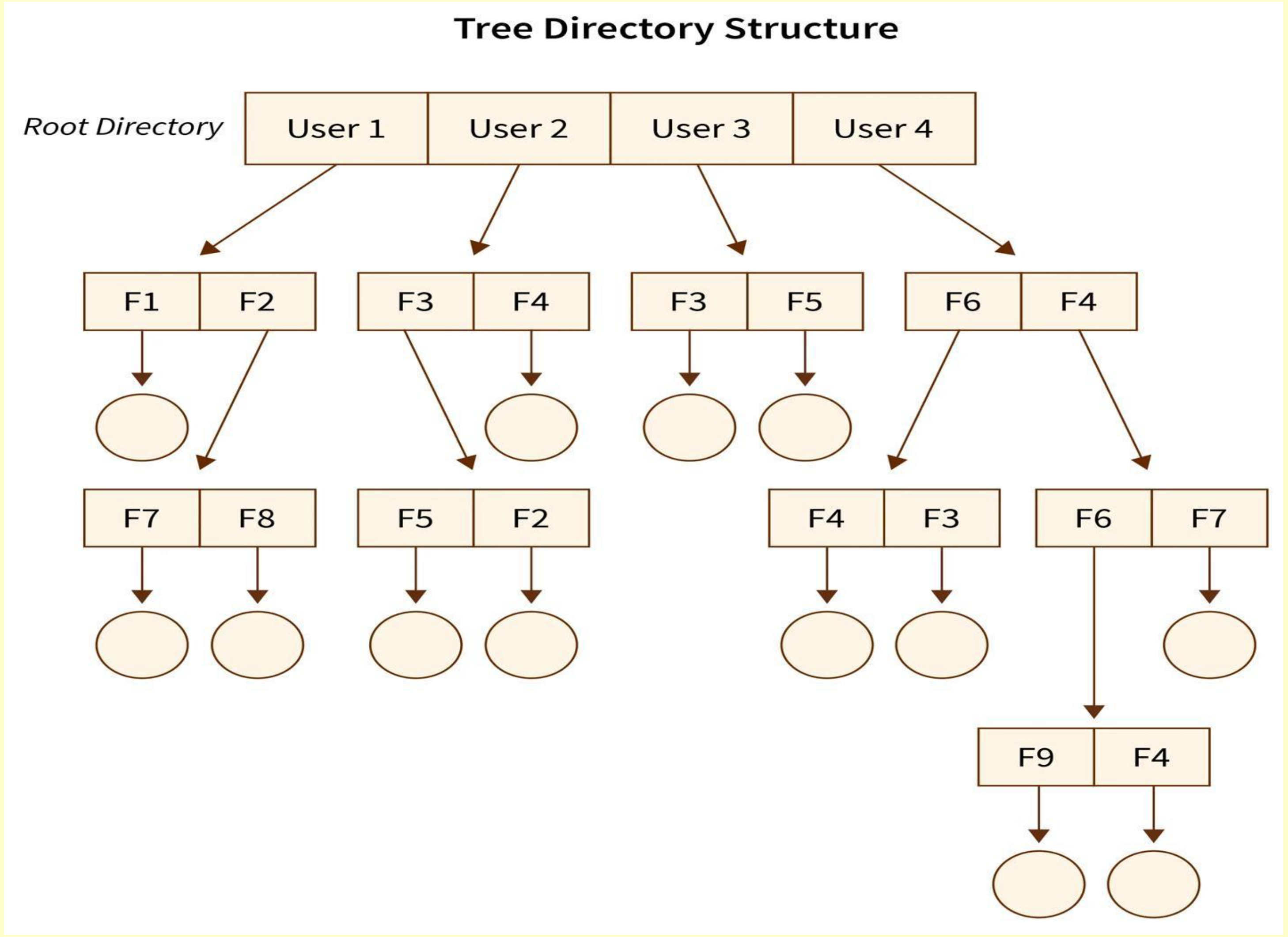
- **Tree Directory Structure - Merits**
  - **Highly Scalable and Organized** It is very effective for organizing a large number of files by grouping them into logical subdirectories.
  - **Flexible Searching** Files can be located using either an absolute path (from the root directory) or a relative path (from the current directory), making navigation efficient.
  - **General and Intuitive**: This structure is widely used in modern operating systems and is easy for users to understand.

Directory Structures Type

- **Tree Directory Structure – Demerits**
- **No Direct File Sharing** In its pure form, this structure does not allow for files or directories to be shared between different branches of the tree, which can lead to file duplication.
- **Inefficiency in Access** Accessing a file deep within the directory tree can be inefficient as it requires traversing multiple directory levels.

Directory Structures Type

❑ Tree Directory Structure



Directory Structures Type

```
/ (Root)
├── home
│   ├── user1
│   │   ├── documents
│   │   │   ├── report.pdf
│   │   │   └── notes.txt
│   │   └── projects
│   │       └── code.py
│   └── user2
│       └── data.csv
├── usr
│   ├── bin
│   └── lib
└── var
    └── log
```

Directory Structures Type

- **Acyclic Graph Directory Structure**
  - This structure is an enhancement of the tree structure that allows for sharing.
  - A file or directory can have multiple parent directories, enabling it to appear in different locations without being duplicated. This is achieved using links or pointers.
  - When users need to share a file, instead of creating a copy, the system creates a link to the original file in the other user's directory.
  - Any changes made to the file are reflected everywhere it is linked.

Directory Structures Type

- **Acyclic Graph Directory Structure – Merits**

  - **Enables File Sharing** It allows for easy collaboration and sharing of files and directories among users, avoiding redundant copies.

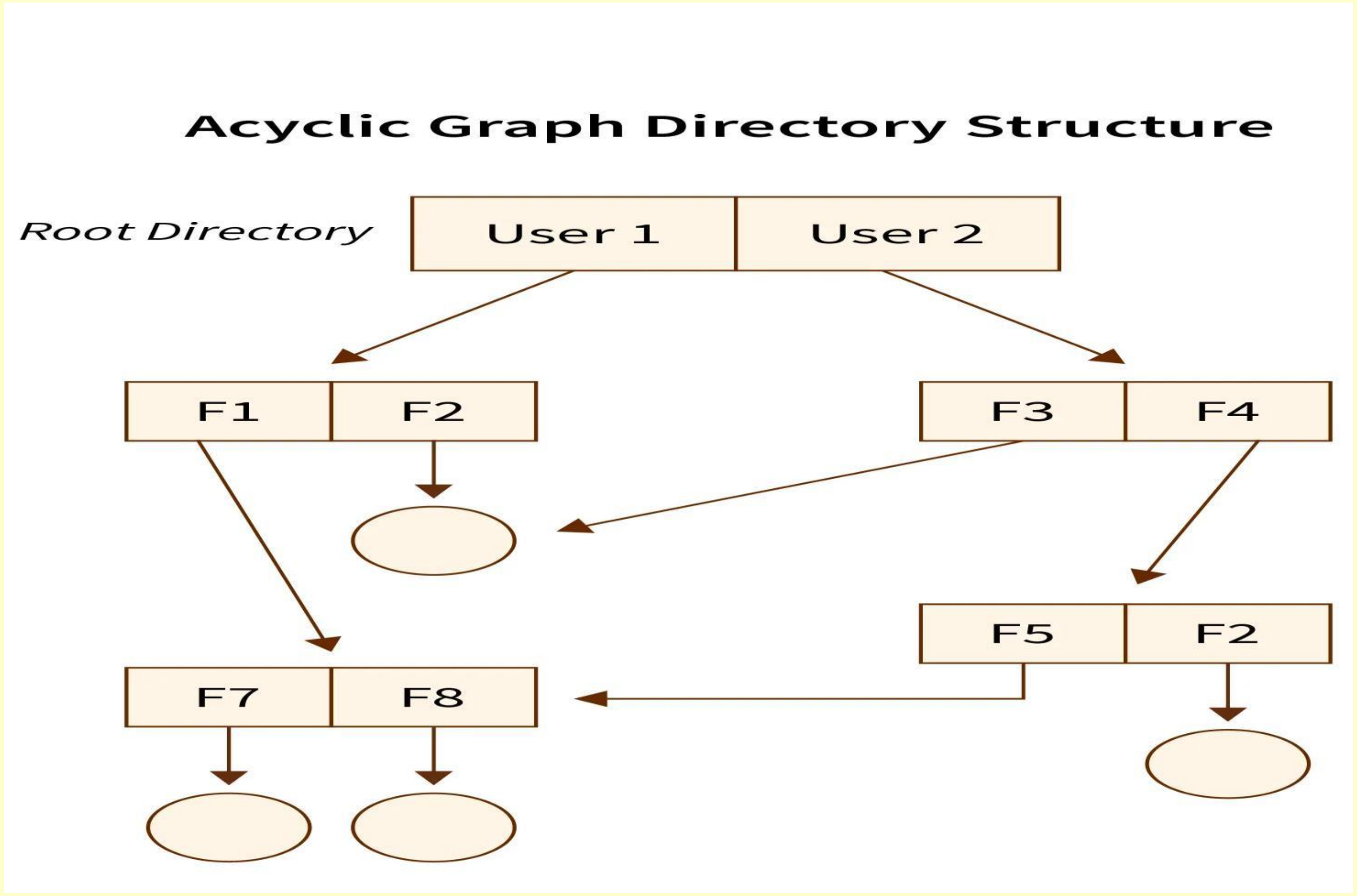
  - **Efficient Searching** The presence of multiple paths to the same file can make searching more flexible

Directory Structures Type

- **Acyclic Graph Directory Structure – Demerits**
  - **Deletion Complexity**: Deleting a file becomes complicated. If a file is deleted, it can leave behind "dangling pointers" in the directories that were linked to it.
  - The system must have a mechanism to handle this, such as only deleting the file when all links to it are removed.
  - **Increased Complexity**: This structure is more complex to manage than a simple tree.

Directory Structures Type

❑ Acyclic Graph Directory Structure



Directory Structures Type

-

```
Root
 ├── UserA
 │    ├── project_file.txt
 │    └── shared_data → (link to /UserB/data)
 └── UserB
      ├── data
      │    └── dataset.csv
      └── project_file.txt → (link to
/UserA/project_file.txt)
```

Directory Structures Type

- **General Graph Directory Structure**
  - This is the most flexible but also the most complex directory structure. It is like the acyclic-graph but allows for cycles. This means a directory can contain a link to one of its parent directories or even to itself.
  - The system allows the creation of links without the restriction of avoiding cycles. This offers maximum flexibility for linking files and directories.

Directory Structures Type

- **General Graph Directory Structure - Merits**
  - **Maximum Flexibility** It provides the greatest freedom in how files and directories can be interlinked.

Directory Structures Type

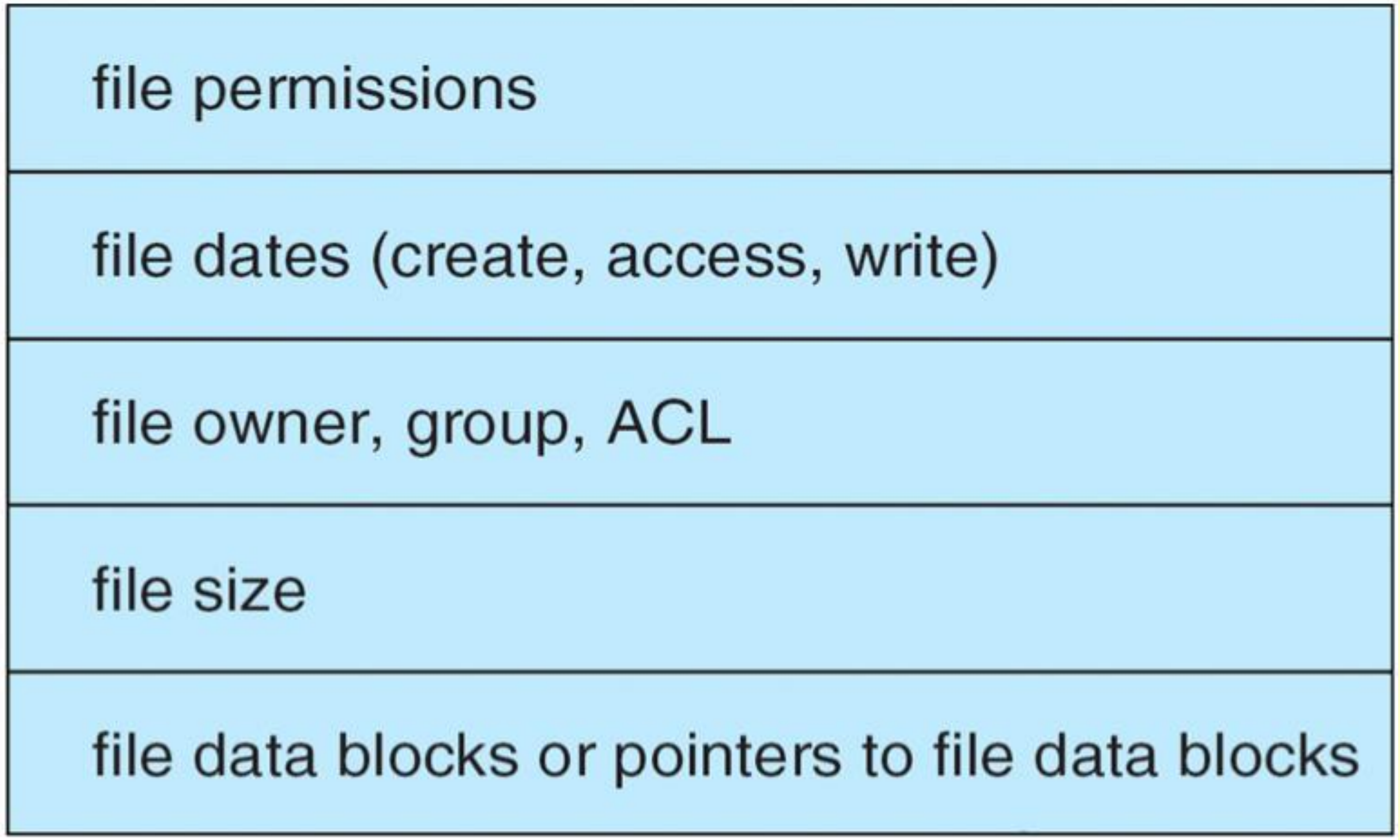
- **General Graph Directory Structure - Demerits**
  - **Risk of Infinite Loops** The presence of cycles can cause algorithms that traverse the directory (like search or file-cleanup utilities) to enter infinite loops. The system must implement cycle detection or garbage collection to manage this.
  - **High Complexity** The complexity of ensuring correctness and preventing problems makes this structure difficult to implement and manage.

File System Implementation

- ❑ File-system needs to maintain **on-disk and in-memory structures**
  - ❑ on-disk for **data storage**,
  - ❑ in-memory for **data access**
- ❑ On-disk structure has several control blocks
  - ❑ Boot control block contains info to boot OS from that volume
    - ❑ only needed if volume contains OS image, usually first block of volume
  - ❑ Volume control block (e.g., superblock) contains volume details
    - ❑ total # of blocks, # of free blocks, block size, free block pointers or array
  - ❑ Directory structure organizes the directories and files
    - ❑ file names and layout
  - ❑ per-file file control block contains many details about the file
    - ❑ inode number, permissions, size, dates

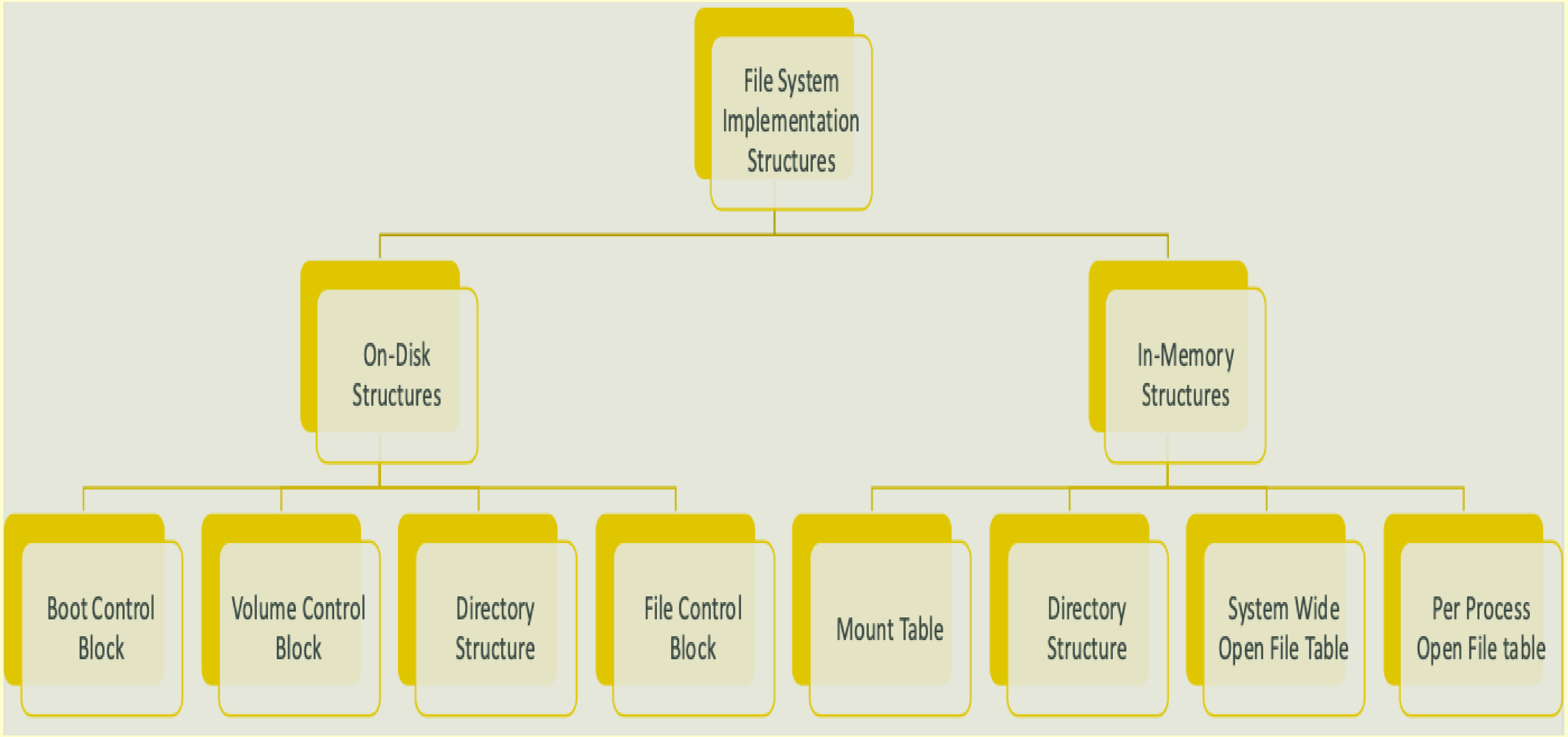
File System Implementation

- File Control Block

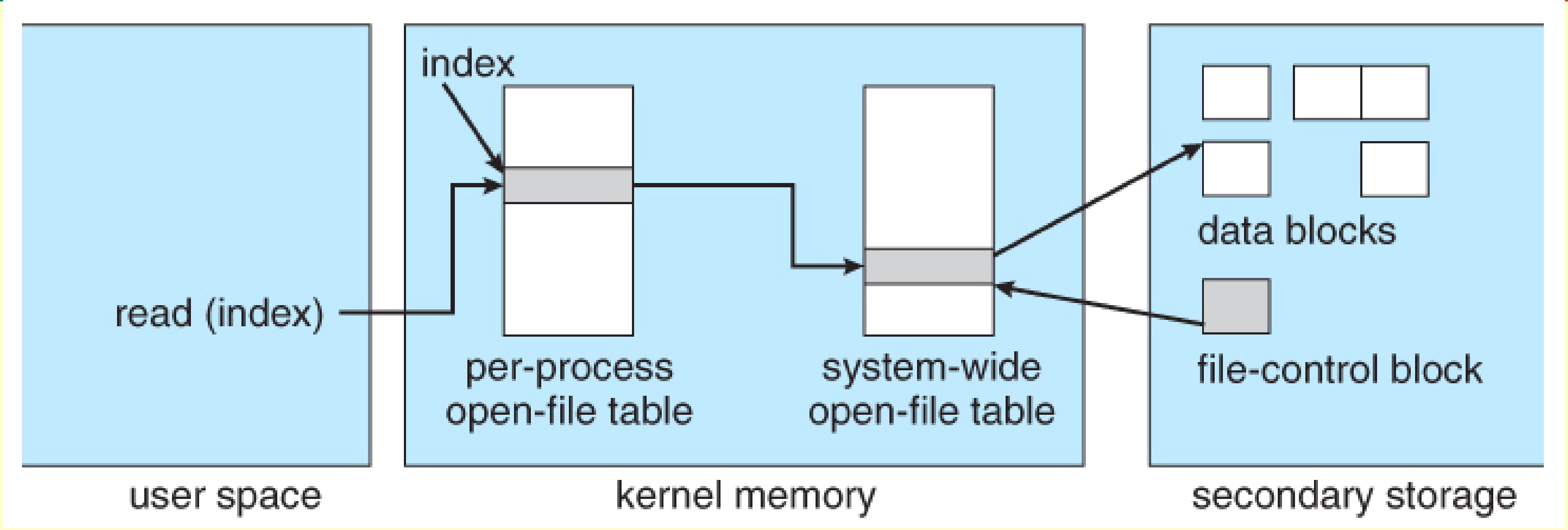


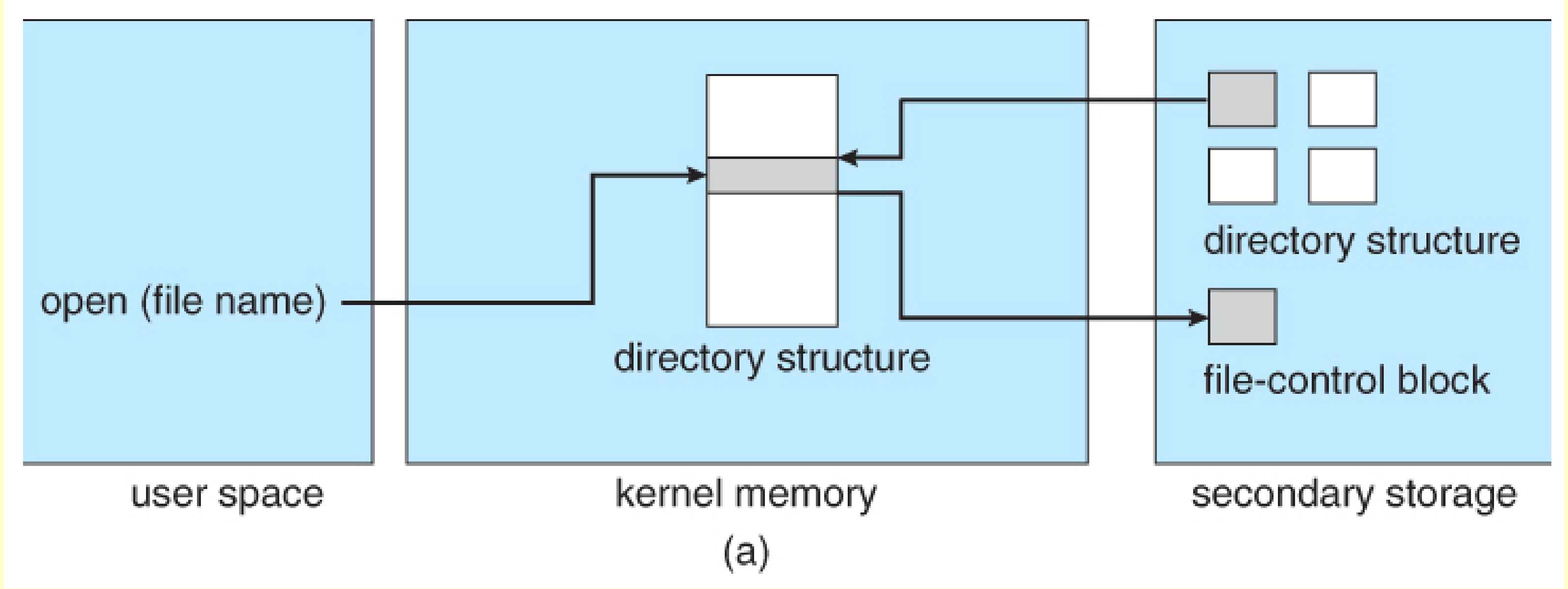
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

File System Implementation

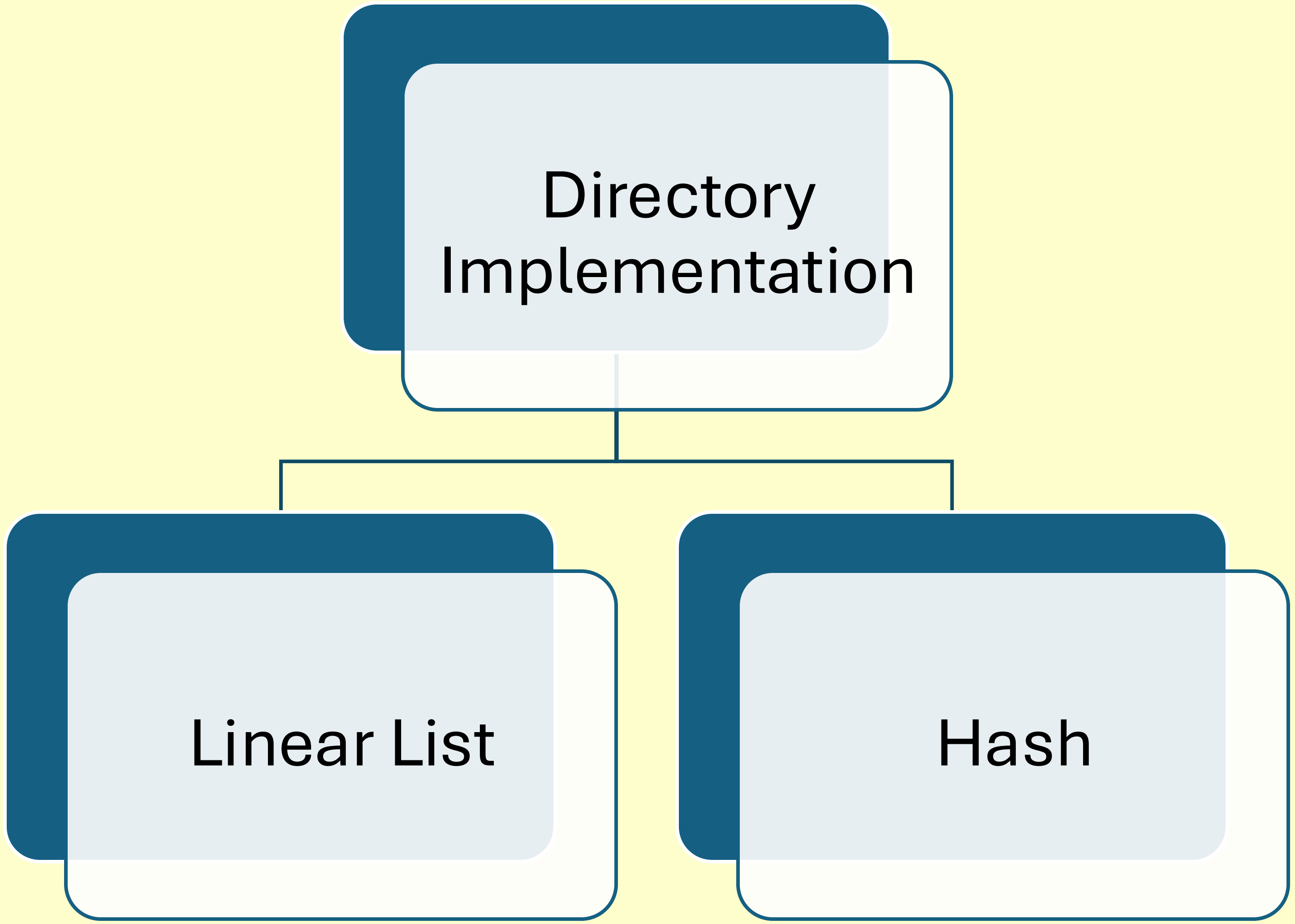


Directory Implementations





Directory Implementations

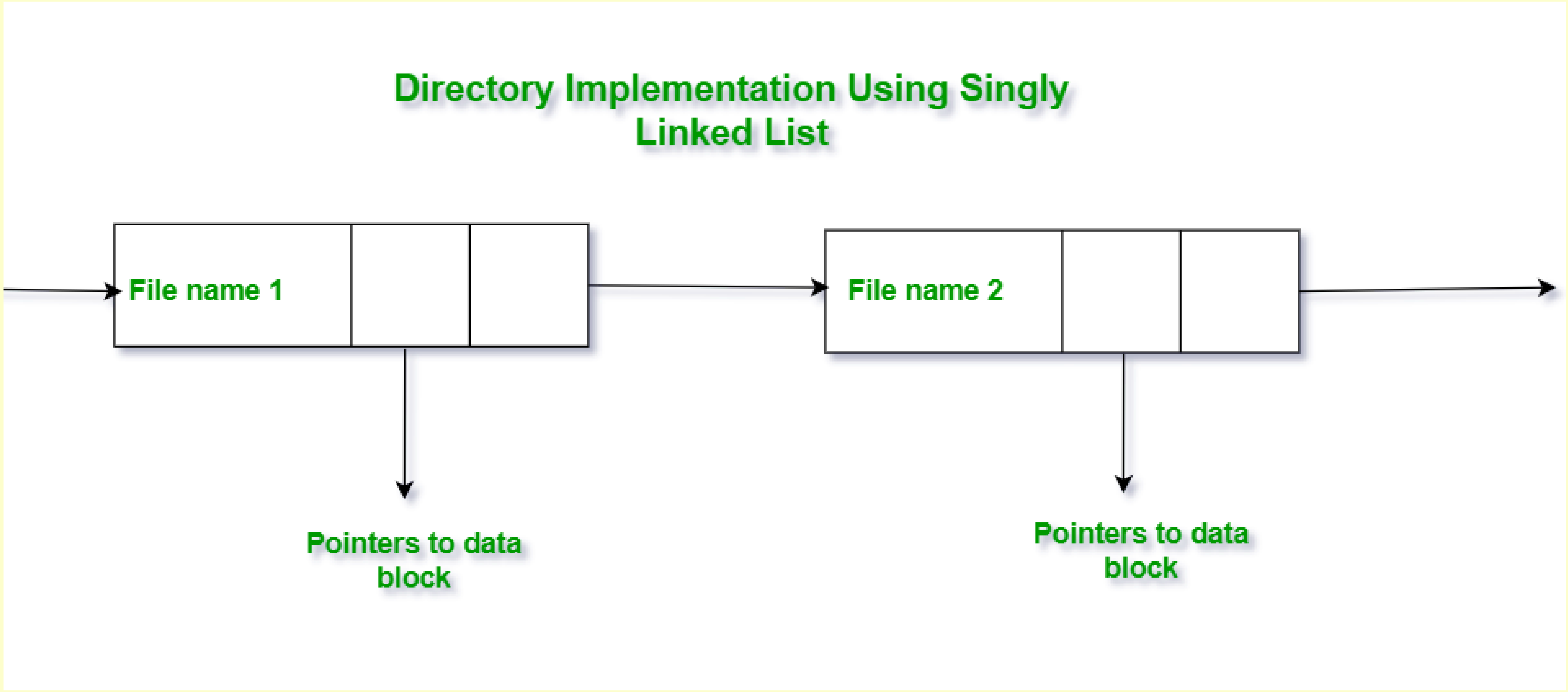


how the logical structure of a directory (as a list of files and subdirectories) is physically stored.

Directory Implementations

- **Linear List**
  - This is the simplest way to implement a directory.
  - It involves creating a list of file names, with each entry in the list pointing to the file's data blocks on the disk.
- When a new file is created, it is added to the end of the list.
- To find a file, the system performs a linear search, checking each entry in the directory one by one.
- To delete a file, the system searches for it and then releases its allocated space.

Directory Implementations



Directory Implementations

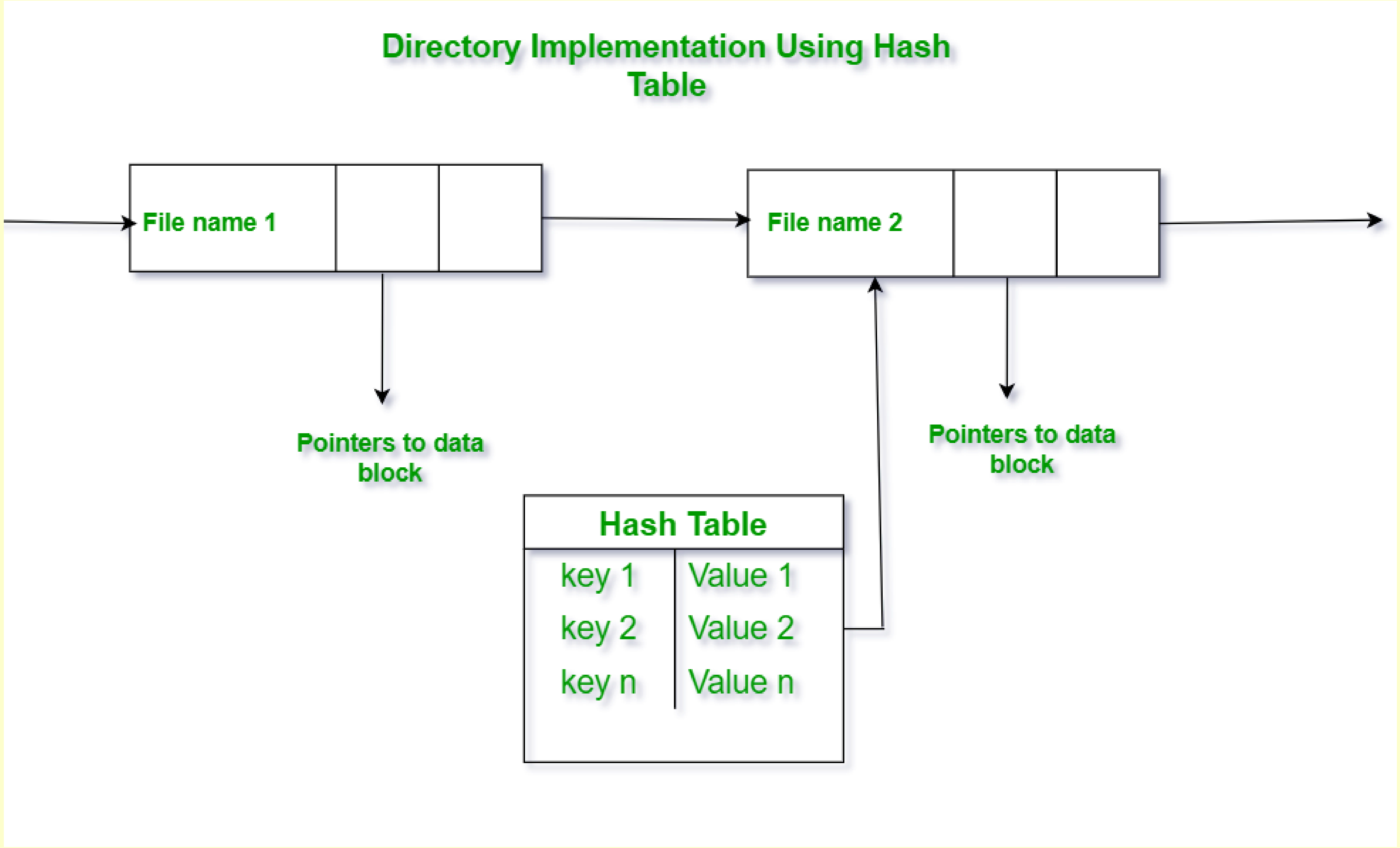
- **Merits**
  - It is simple to program and understand.

- **Demerits**
  - It is time-consuming to execute operations because it requires a linear search. This becomes very slow as the directory grows.
  - To improve performance, the list can be kept sorted alphabetically, which allows for a binary search, but this makes file creation and deletion more complex.

Directory Implementations

- **Hash Table**
  - This method uses a hash table in conjunction with a linear list to speed up file searching.
- A hash function takes the file name and computes a hash value.
- This value is used as an index into the hash table, which then points to the file's entry in the linear list.
- This avoids a lengthy linear search.

Directory Implementations

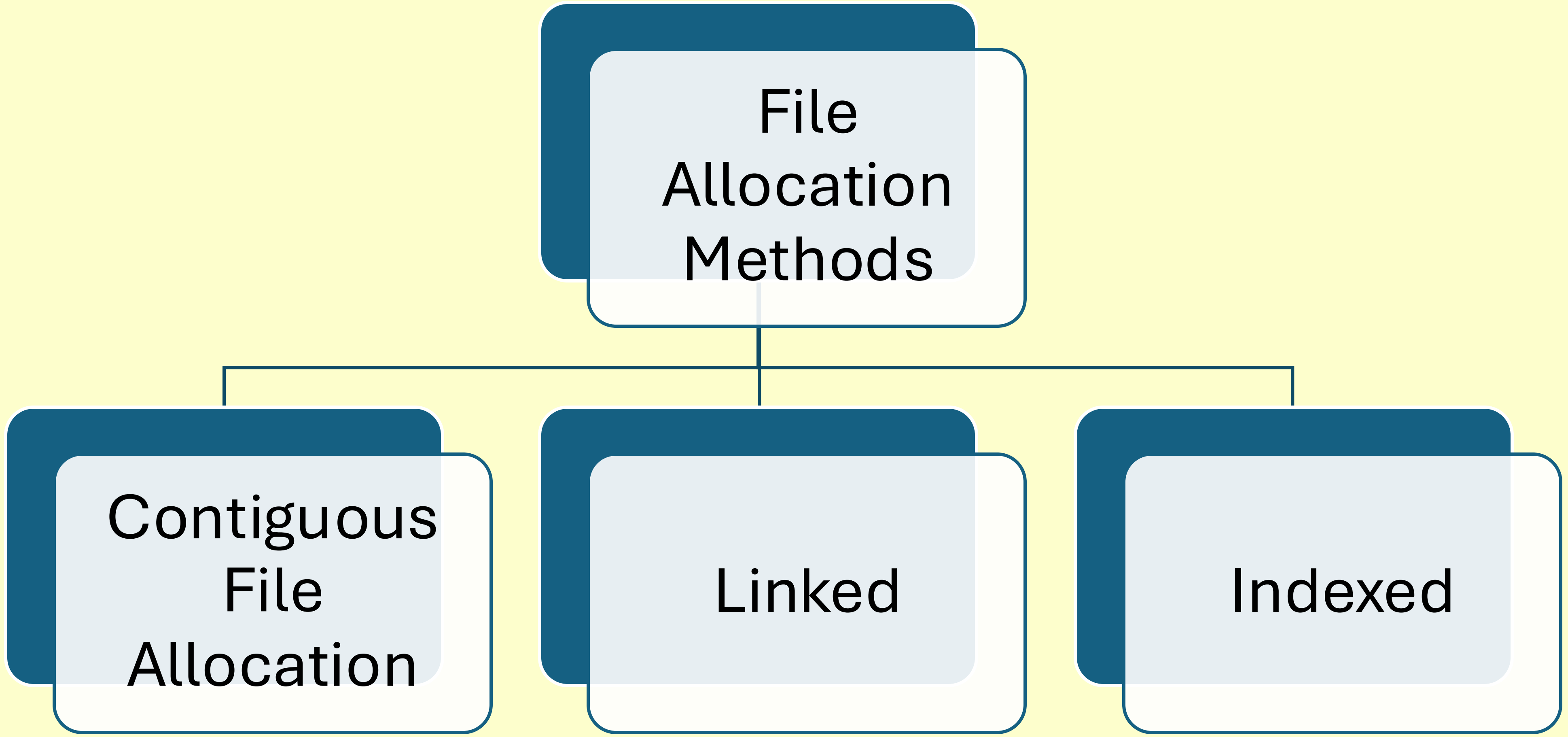


Directory Implementations

- ❑ **Merits**
  - ❑ It significantly decreases directory search time, making it much faster to locate a file.

- ❑**Demerits**
  - ❑ **Collisions** It's possible for two different file names to hash to the same location. This must be managed using collision resolution techniques, such as chaining.
  - ❑ **Fixed Size:** Hash tables are typically of a fixed size. If the number of files grows too large, the performance degrades, and the table may need to be resized and reorganized, which is a complex operation.

File Allocation Methods



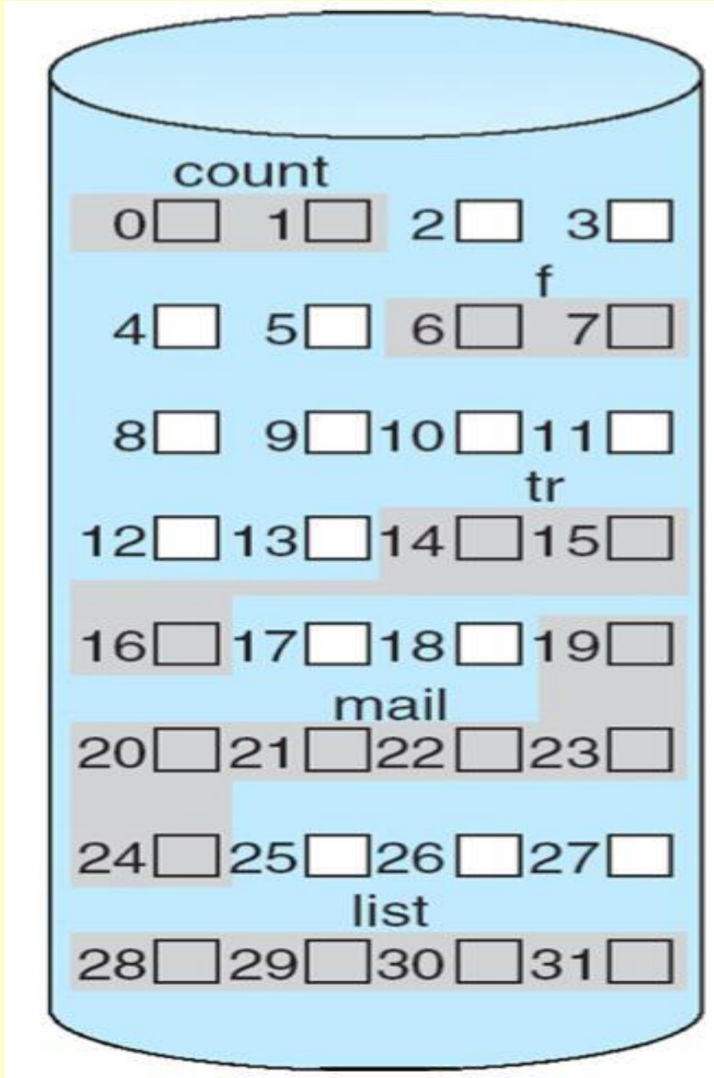
File allocation methods determine how an operating system allocates disk blocks for files. The three main methods are contiguous, linked, and indexed allocation.

File Allocation Methods

- **Contiguous Allocation**
  - In this method, each file occupies a set of adjacent, contiguous blocks on the disk.
- The directory entry for a file needs to store only two pieces of information: the starting block address and the total length of the file (in blocks).
- Accessing any part of the file is straightforward because the system can calculate the exact block address from the starting address and the logical position within the file.

File Allocation Methods

- **Contiguous Allocation**



directory

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

File Allocation Methods

- **Merits**
  - It offers excellent performance for both sequential and random access because the file's blocks are physically close together, minimizing disk head movement.
  - It is simple to implement.

- **Demerits:**
  - **External Fragmentation**: Over time, as files are created and deleted, the free space on the disk becomes broken into small, non-contiguous chunks. This can make it difficult to find a large enough contiguous space for a new file, even if there is enough total free space.

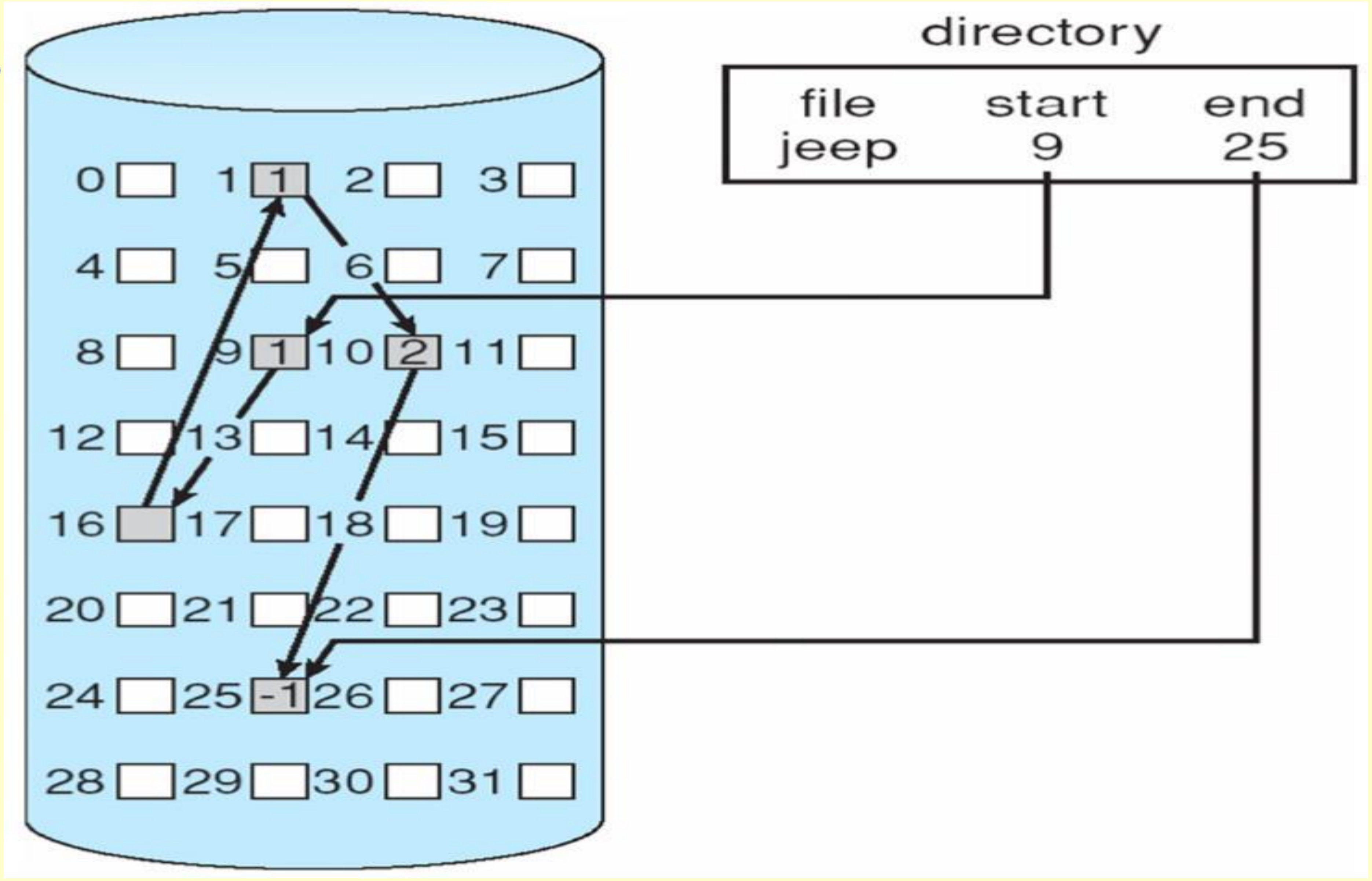
  - **File Growth:** It is difficult for files to grow, as they may run into another file immediately after their allocated space.

File Allocation Methods

- ❑**Linked Allocation**
  - ❑ This method stores each file as a linked list of disk blocks, which can be scattered anywhere on the disk.
- ❑ Each block contains not only the file's data but also a pointer to the next block in the file.
- ❑ The directory entry only needs to store the starting block address.
- ❑ The file is traversed by following the pointers from one block to the next.

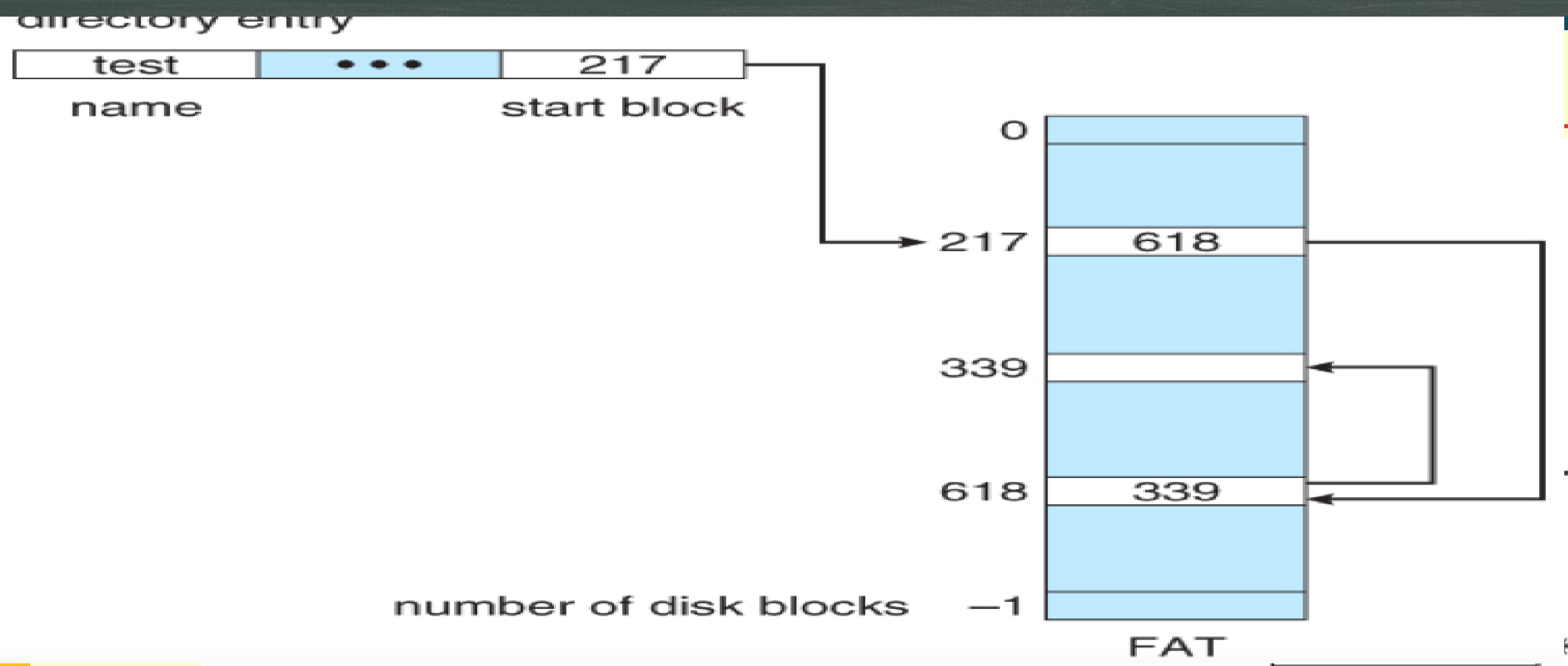
File Allocation Methods

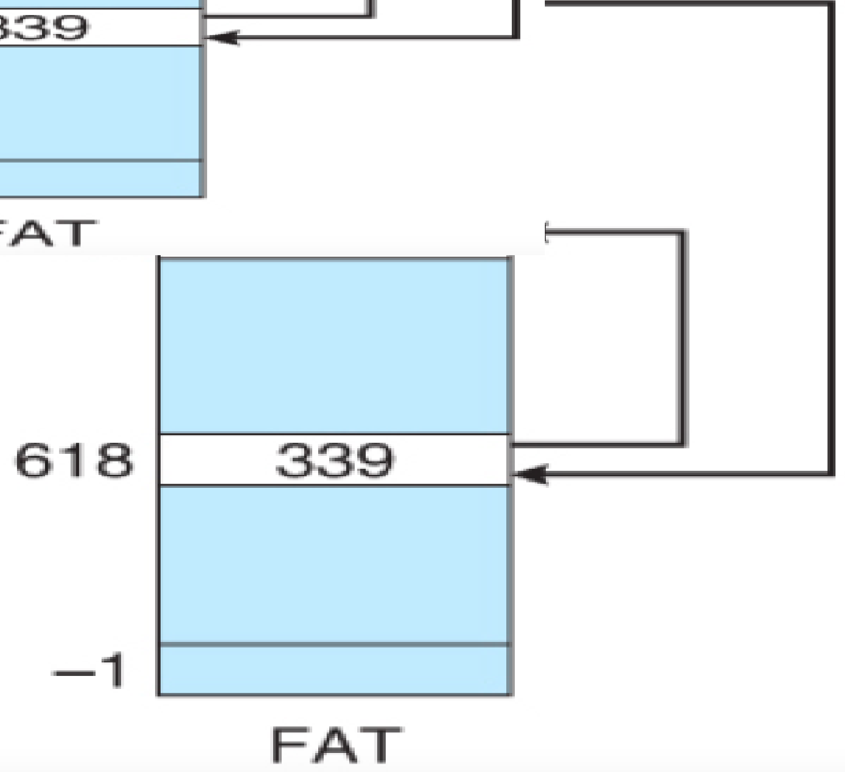
- **Linked Allo**



File Allocation Methods

- **Linked Allocation**
  - **Merits:**
    - It eliminates external fragmentation, as any free block can be used.
    - Files can grow easily and dynamically as long as there are free blocks available.
  - **Demerits:**
    - It is only efficient for sequential access. Random access is very slow because, to get to a specific block, the system must traverse the chain from the beginning.
    - It has a space overhead because each block must reserve space for the pointer.
    - It is not very reliable. If a pointer is damaged or lost, the rest of the file becomes inaccessible.
- FAT (File Allocation Table): A variation of this method, used by MS-DOS and early Windows, takes the pointers from all blocks and stores them in a separate table at the beginning of the disk, called the File Allocation Table (FAT). This improves random access times.

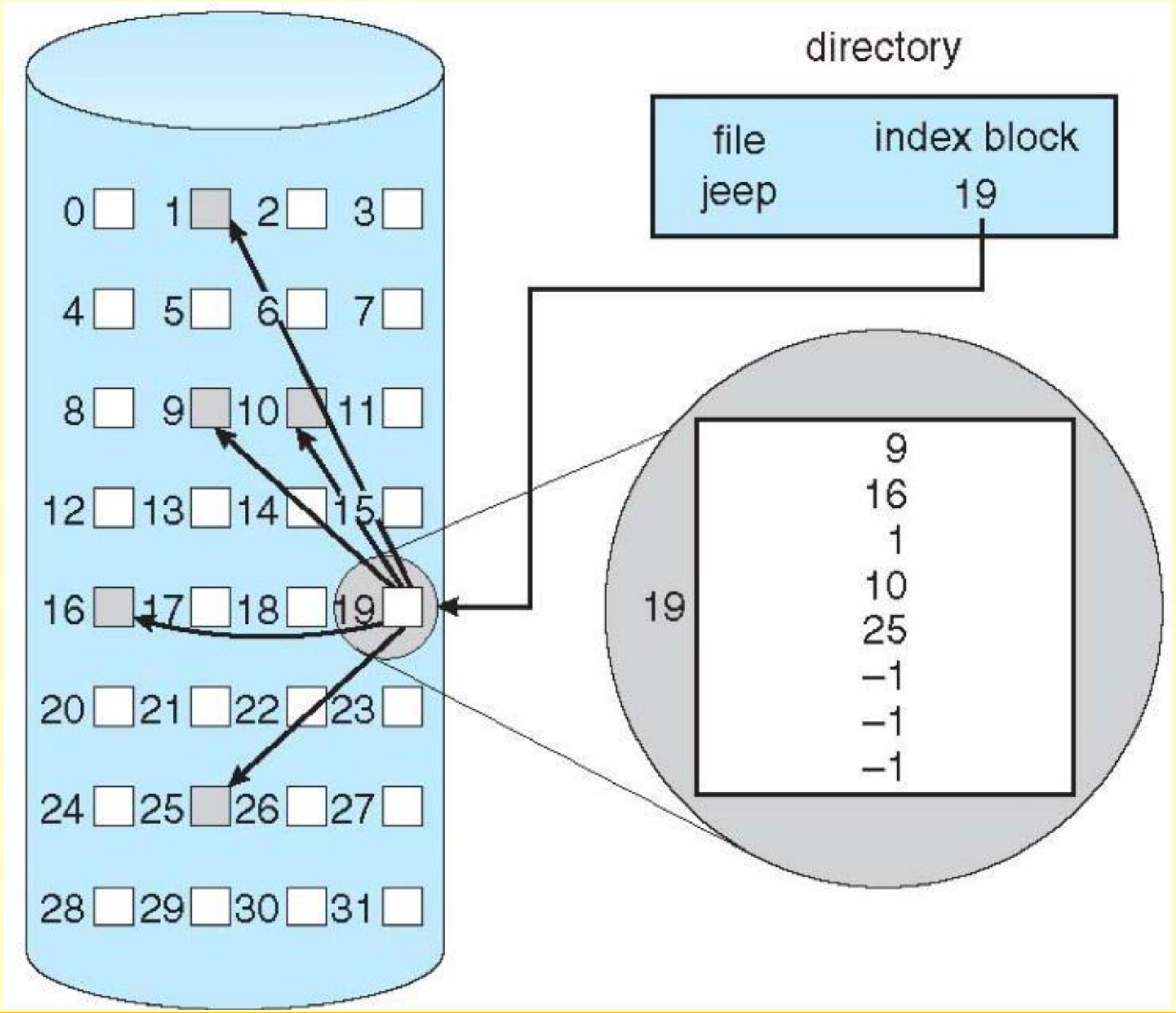

directory entry
test
• • •
217
name
start block
0
217
618
339
339
618
339
number of disk blocks
–1
FAT


618
339
number of disk blocks
–1
FAT

File Allocation Methods

- ❑ **Indexed Allocation**
  - ❑ This method combines the benefits of the previous two by bringing all the pointers for a file's blocks together into a single location called an index block.
- ❑ Each file has its own index block, which is an array of disk block addresses.
- ❑ The i^{th} entry in the index block points to the i^{th} block of the file.
- ❑ The directory entry contains the address of this index block.

File Allocation Methods



File Allocation Methods

- **Merits:**
  - It supports direct, random access to any block of the file without performance loss.
  - It solves the problem of external fragmentation.
- **Demerits:**
  - It has a space overhead due to the need for the index block. For small files, this can be wasteful.
  - For very large files, a single index block may not be enough to hold all the pointers. This is handled by creating multiple, linked index blocks or a multi-level index, which adds complexity.

File Allocation Methods

❑Contiguous Allocation

- Best for: Both sequential and random access.
- Why: All the file’s blocks are together, so reading through (sequential access) or jumping to a specific part (random access) is fast—just calculate the block's position and go straight to it.
- Drawback: Hard for files to grow or shrink; may run into fragmentation.

File Allocation Methods

❑Linked Allocation

- Best for: Sequential access.
- Why: Each file block has a pointer to the next, so you just follow the chain for sequential reads/writes.
- Drawback: For random access, you have to start at the beginning and step through the chain—very slow.

File Allocation Methods

❑Indexed Allocation (including Combined)

- ❑ Best for: Both sequential and random access, but with more overhead.
- ❑ Why: The file has an index block (or multiple for big files) listing addresses of all its blocks. For random access, get the required block's address from the index and read it.
- ❑ Drawback: Sometimes you need to read two index blocks before you can read the data block—adds complexity and overhead.