File System Interface and Operations

File System Interface and Operations

A file in computing is defined as a named collection of related information that is stored on secondary storage. It is the smallest logical allotment of secondary storage and can contain various types of information defined by its creator and user.

File Attributes provide the operating system with information about the file and its intended use. These attributes typically include:

1. Name: The human-readable symbolic name of the file.
2. Identifier: A unique tag, usually a number, that identifies the file within the file system.
3. Type: Information about the type of file, especially relevant for systems supporting different file types.
4. Location: Pointer to the device and the file's location on that device.
5. Size: The current size of the file and possibly the maximum allowed size.
6. Protection: Access-control information determining permissions for reading, writing, executing, etc.
7. Time, date, and user identification: Information about creation, last modification, and last use for protection, security, and usage monitoring purposes.

Newer file systems may support extended attributes like character encoding and file checksums for added security and functionality.

File Types can indicate the internal structure and type of the file. Operating systems may recognize file types by extensions in the file name (e.g., .docx for Word documents, .c for C source files). Some OSes support multiple file structures, while others impose a minimal set of structures.

Internal File Structure:

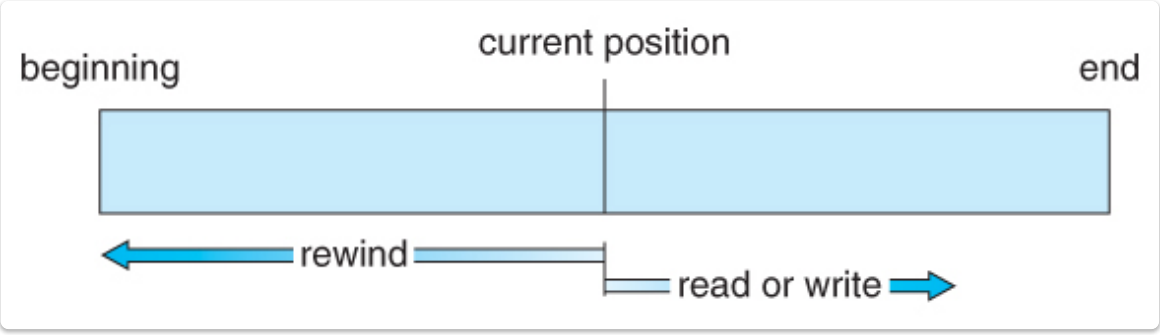
1. Block Structure: Disk systems use a well-defined block size for disk I/O operations, typically determined by the sector size. All disk I/O is performed in units of blocks, which are of uniform size.
2. Record Structure: Files contain a sequence of fixed-length records, although physical records may not always match logical records. Logical records can vary in length.

Access Methods

Access Methods

Access methods define how information stored in a file is accessed and read into computer memory. There are several ways to access information in a file:

1. **Sequential Access**:
   - This is the simplest access method where information in the file is processed in order, one record after the other.
   - It is akin to the tape model of a file and works well with sequential-access devices.
   - Example: Editors and compilers often access files sequentially.
   - Operations: Read operations automatically advance a file pointer, while write operations append to the end of the file.



2. **Direct Access (or Relative Access)**:
   - This method allows rapid reading and writing of fixed-length logical records in no particular order.
   - Files are viewed as numbered sequences of blocks or records, and there are no restrictions on the order of reading or writing.
   - Example: Databases often use direct access for immediate access to large amounts of information.
   - Operations: File operations include the block number as a parameter, enabling reading or writing of specific blocks.
3. **Indexed Access**:
   - This method involves constructing an index for the file, similar to an index in the back of a book.
   - The index contains pointers to various blocks, enabling direct access to desired records.
   - Example: A retail-price file may use UPCs as keys to access prices directly.
   - Operations: Searching the index, followed by direct access to the desired record.

Directory Overview

A file system can be created on different parts of the disk, known as volumes. Each volume containing a file system must also maintain information about the files within it, usually stored in a device directory or volume table of contents.

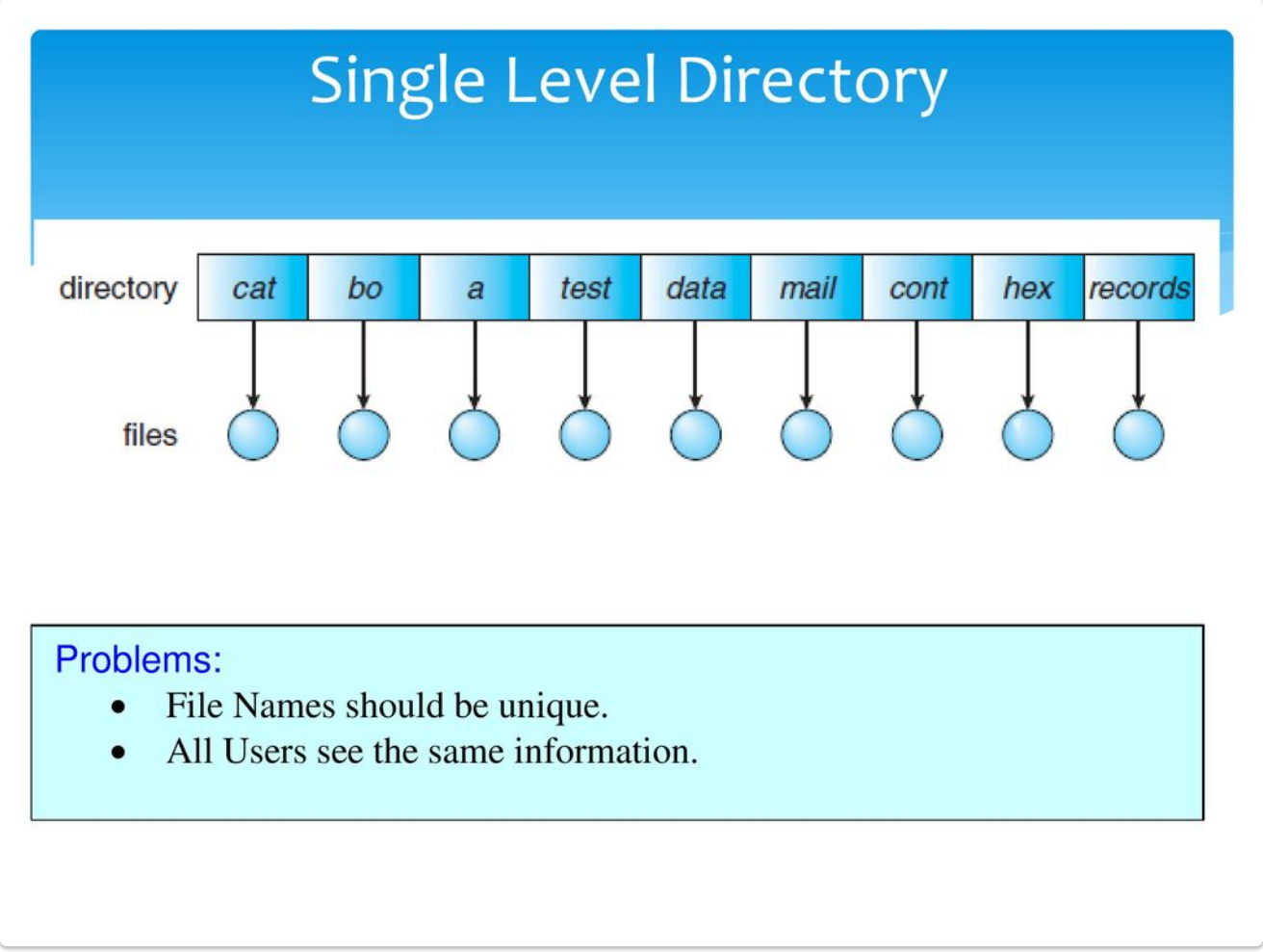
Operations performed on a directory include:

- Searching for a file
- Creating a file
- Deleting a file
- Listing the directory contents
- Renaming a file
- Traversing the file system

Directory Structure

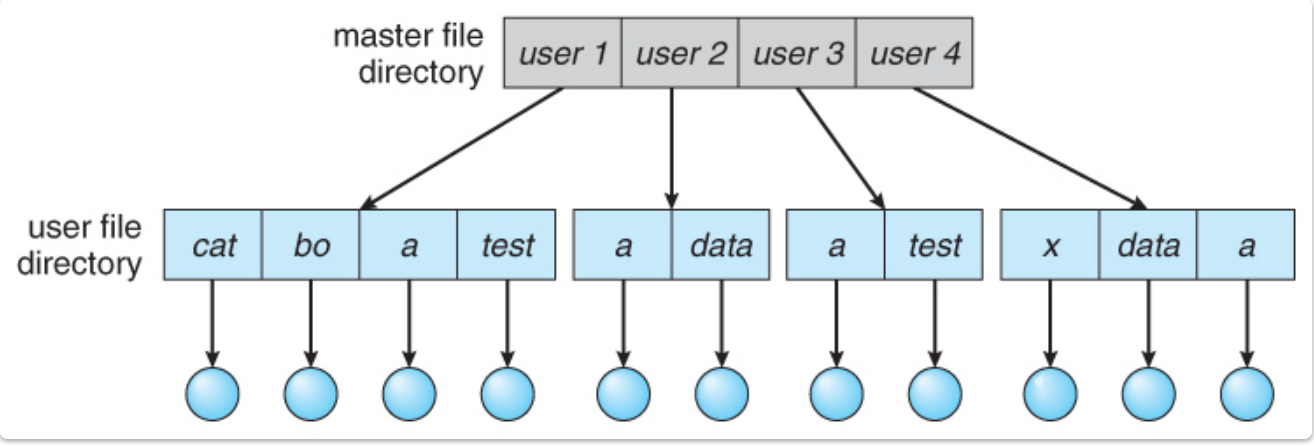
Directory structures define how files are organized within an operating system. Here are the two most common schemes for defining the logical structure of a directory:

1. **Single-Level Directory:**
    - All files are contained within a single directory.
    - **Advantages:**
        - Easy to support and understand.
    - **Limitations:**
        - Each file must have a unique name, which can become cumbersome as the number of files increases.
        - Difficult for users to manage a large number of files.



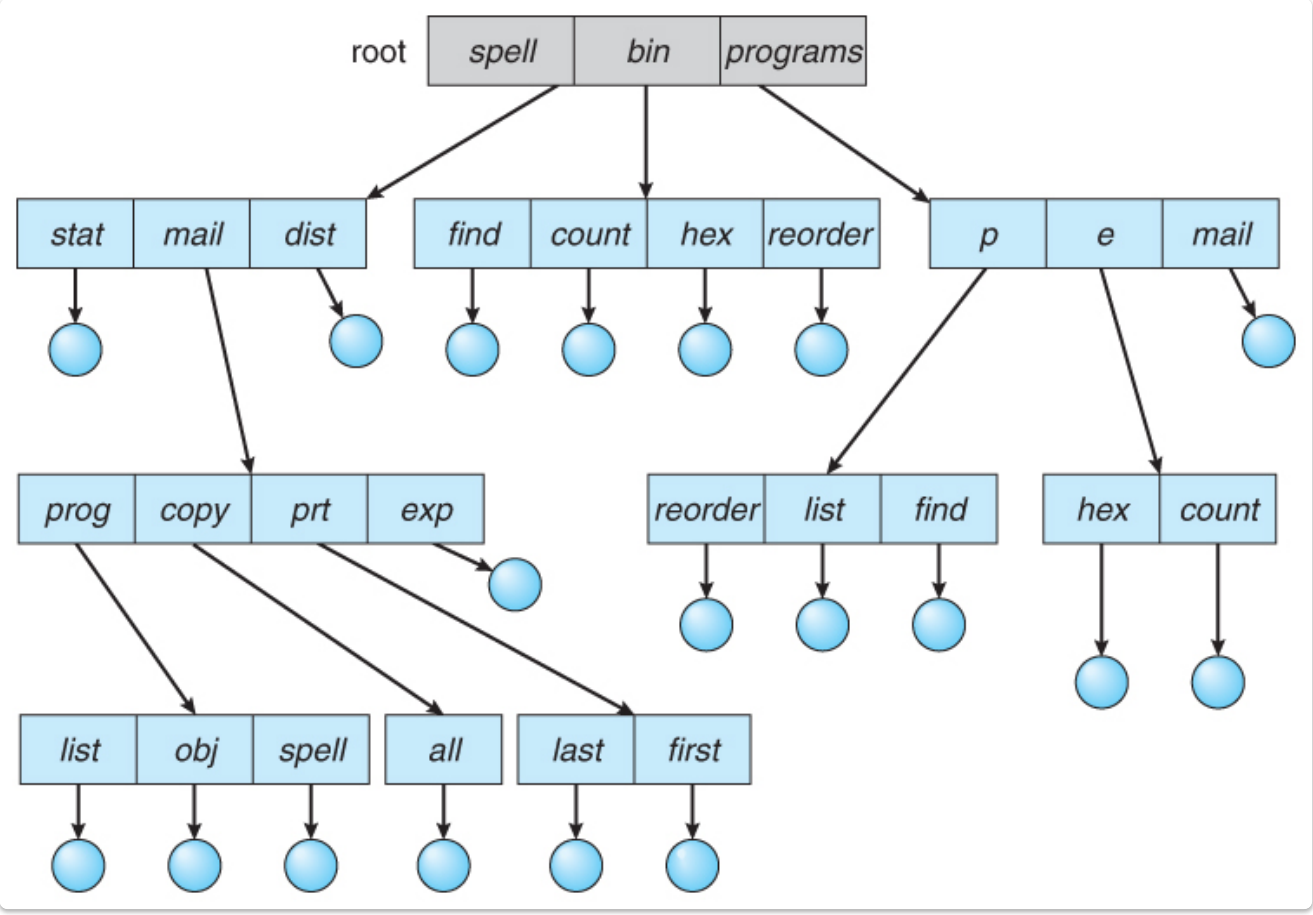
2. Two-Level Directory:

- Consists of two directory levels:
  - Master File Directory (MFD) at the top level.
  - User File Directory (UFD) at the second level, with actual files at the third level.
- Each user has their own UFD, which helps eliminate confusion among different users' file names.
- When a user logs in, the system searches the MFD for their UFD.
- Users can only access files within their own UFD, preventing accidental deletion of other users' files.
- Advantages:
  - Solves the name-collision problem.
  - Ensures user privacy and file isolation.
- Disadvantages:
  - Makes it difficult for users to access files across directories if cooperation is needed.
  - Special mechanisms may be required to allow access to system files.



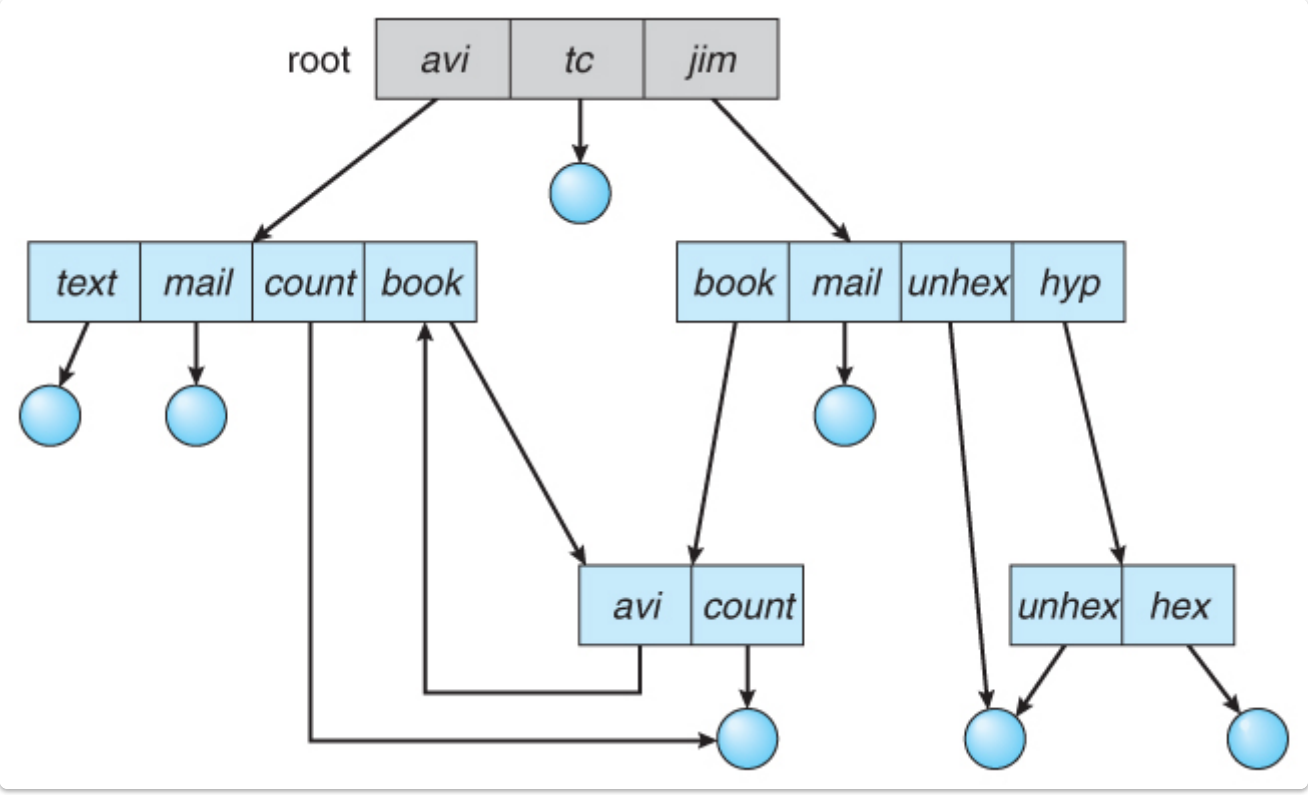
3. Tree-Structured Directories:

- Description: In a tree-structured directory, a root directory serves as the top-level directory, and every file in the system has a unique path name originating from this root. Directories, or subdirectories, contain files or other subdirectories. Each directory entry contains a bit indicating whether it represents a file (0) or a subdirectory (1). Special system calls handle the creation and deletion of directories.
- Current Directory:
  - Every process has a current directory, which usually contains files of immediate interest to that process.
  - When a file reference is made, the operating system searches the current directory. If the file isn't found there, the user may need to specify a path name or change the current directory using system calls like `change directory()`.
- Path Names:
  - Path names describe the route the operating system must follow to reach a particular location.
  - Two types of path names exist:
    - Absolute: Starts from the root and includes all directory names in the path.
    - Relative: Defines a path starting from the current directory.
- Deletion of a Directory:
  - If a directory is empty, its entry in the parent directory can be deleted easily.
  - When a directory contains files or subdirectories, systems may take different approaches:
    - Some systems require directories to be empty before deletion.
    - Others provide options to delete a directory along with its contents, similar to the UNIX `rm` command.



4. Acyclic-Graph Directories:

- Description: In an acyclic-graph directory structure, files and directories can share subdirectories, allowing for greater flexibility than a simple tree structure. This setup permits the same file or subdirectory to exist in multiple directories, creating complexity but offering more options for file organization.
- Implementation:
  - Two common methods are:
    - Linking: Create directory entries called links, acting as pointers to other files or subdirectories.
    - Duplication: Duplicate all information about shared files in both directories where they reside.
- Issues:
  - Multiple absolute path names for files may complicate file traversal.
  - Deletion strategies must address issues like dangling pointers, maintaining consistency, and determining when to deallocate space allocated to shared files.



Protection

When dealing with information stored within a computer system, ensuring its safety from physical damage (reliability) and unauthorized access (protection) becomes paramount. Reliability often involves creating duplicate copies of files to mitigate potential data loss due to system failures or accidental deletions. On the other hand, protection mechanisms aim to restrict access to files and directories based on predefined rules and user identities.

1. Reliability

Reliability is maintained through duplicate copies of files, often facilitated by system programs that automatically back up data at regular intervals onto secondary storage media like tapes. This practice ensures that in the event of file system corruption or accidental deletion, a copy of the data remains intact.

Factors that can compromise reliability include hardware malfunctions (e.g., disk errors), power surges, head crashes, environmental conditions, and malicious activities such as vandalism or deliberate file system manipulation.

2. Protection Mechanisms

Protection mechanisms control access to files and directories by regulating the types of operations users can perform. These operations include:

- Read: Accessing file contents.
- Write: Modifying file contents.

- Execute: Running executable files.
- Append: Adding information to the end of a file.
- Delete: Removing files from the system.
- List: Viewing file attributes.

Additional operations like renaming, copying, and editing files may also be subject to control.

3. Access Control

Access control ties file access to user identities. Access rights are defined in an Access Control List (ACL), which specifies the users allowed to perform specific operations on a file or directory.

When a user requests access to a file, the operating system checks the ACL associated with that file. If the user's identity matches an entry in the ACL, access is granted; otherwise, access is denied.

Access Classification:

- Owner: The creator of the file.
- Group: A set of users with similar access needs.
- Universe: All other users in the system.

4. Permission Structures

Permission structures dictate access rights based on the owner, group, and universe classifications. In systems like UNIX, permissions are represented using a collection of bits, typically three bits each for read (r), write (w), and execute (x) access. This results in a total of 9 bits per file to record protection information.

Example: `19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1`

5. Other Protection Approaches

- Password Protection: Associates a password with each file or directory, restricting access based on authentication. However, managing multiple passwords can be cumbersome, and using a single password for all files poses security risks.
- Subdirectory Passwords: Some systems allow users to associate passwords with subdirectories, offering a more granular level of access control compared to file-based passwords.

In conclusion, effective protection strategies combine reliability measures with access control mechanisms to safeguard information integrity and restrict unauthorized access

in computer systems.

File System Structure

Disks serve as primary secondary storage devices for file systems due to their ability to be rewritten and access any block of information directly. File systems facilitate efficient storage, location, and retrieval of data on disks.

Design Problems in File Systems

1. User Perspective: Designing the file system's appearance to users, including file and directory structures for organizing files.
2. Mapping: Creating algorithms and data structures to map the logical file system onto physical secondary-storage devices.

Layered Design of File Systems

Each level in the design utilizes features from lower levels to create new features for higher levels.

- Application Programs: Contains user code making requests.
- Logical File System: Manages metadata information excluding actual data, including directory structure management.
- File-Organization Module: Knows about files, logical blocks, and physical blocks, translating logical block addresses to physical block addresses for the basic file system.
- Basic File System: Issues generic commands to device drivers to read and write physical blocks on the disk, identifying each physical block by its numeric disk address.
- I/O Control: Consists of device drivers and interrupt handlers facilitating information transfer between main memory and the disk system, translating high-level commands to low-level hardware-specific instructions.
- Devices: Actual hardware devices like disks.

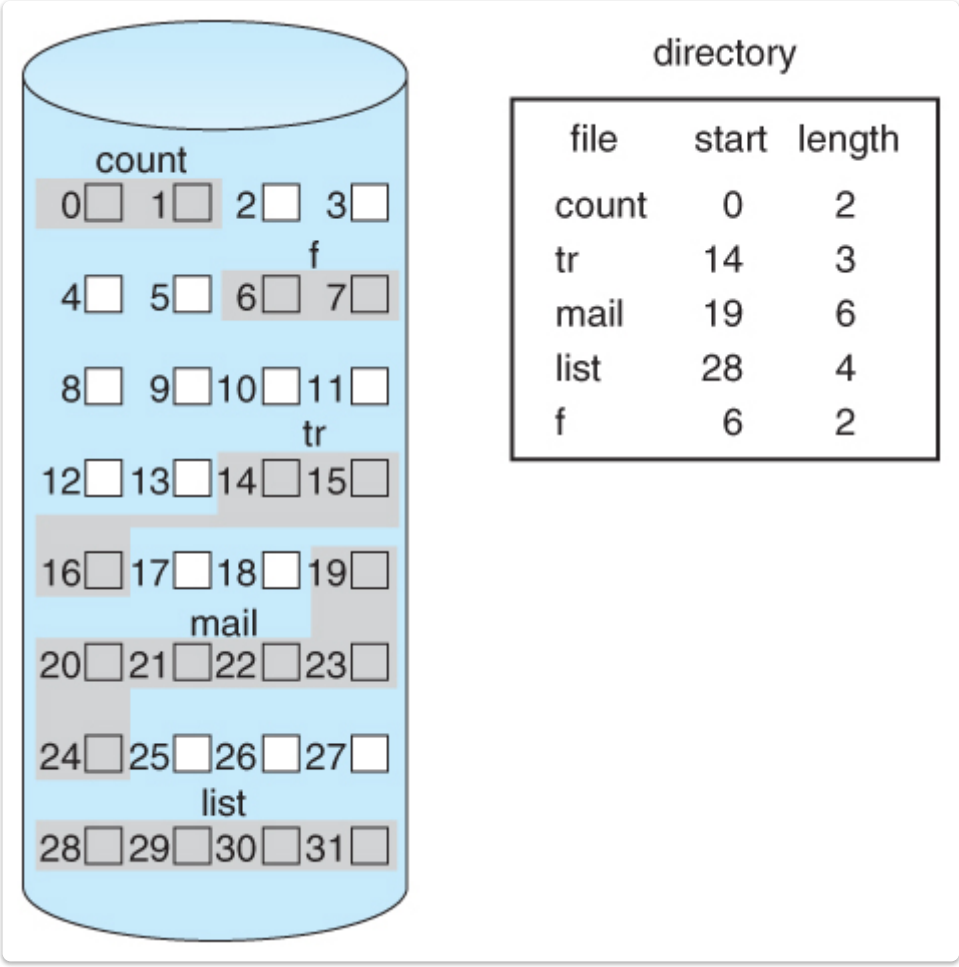
Allocation Methods

Files stored on disks require efficient allocation methods to ensure effective disk space utilization and quick file access. Three major methods of allocating disk space are widely used:

1. Contiguous Allocation

- Requires each file to occupy a set of contiguous blocks on the disk.
- Access to files is straightforward, supporting both sequential and random access.

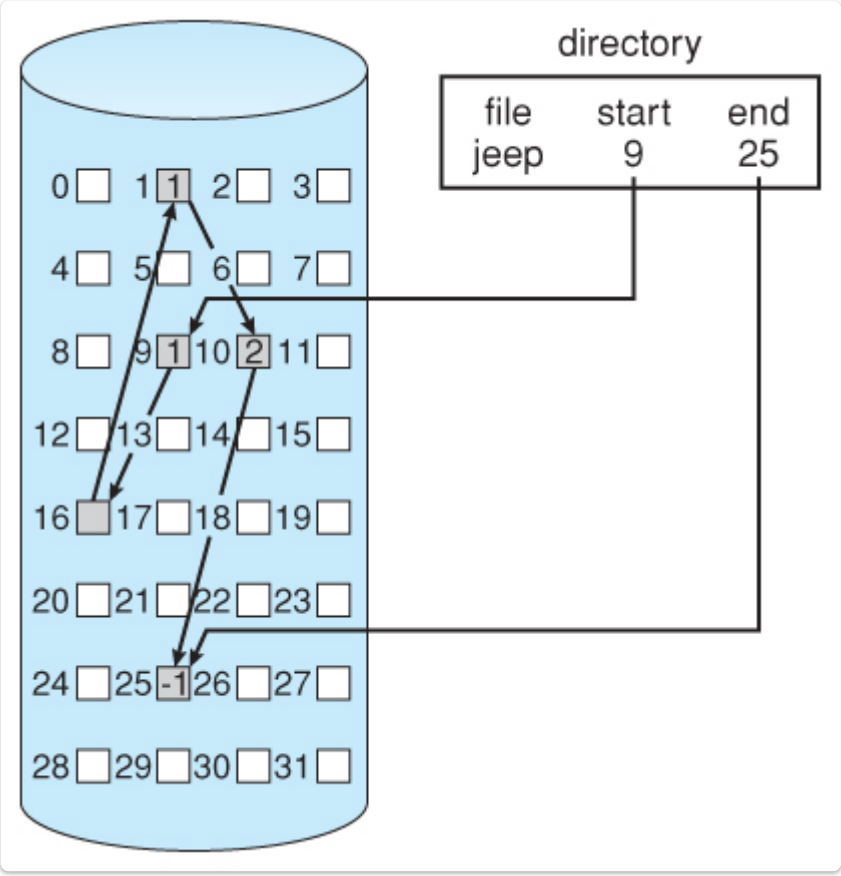
- Drawbacks:
  - Finding space for new files may lead to external fragmentation.
  - External fragmentation occurs as free disk space is broken into small pieces, making it challenging to allocate sufficient contiguous space for large files.
  - Solutions include copying the entire file system onto another disk to compact free space and prevent fragmentation.
  - Determining the required space for a file poses challenges; underestimation may lead to program termination, while overestimation results in wasted space.



2. Linked Allocation

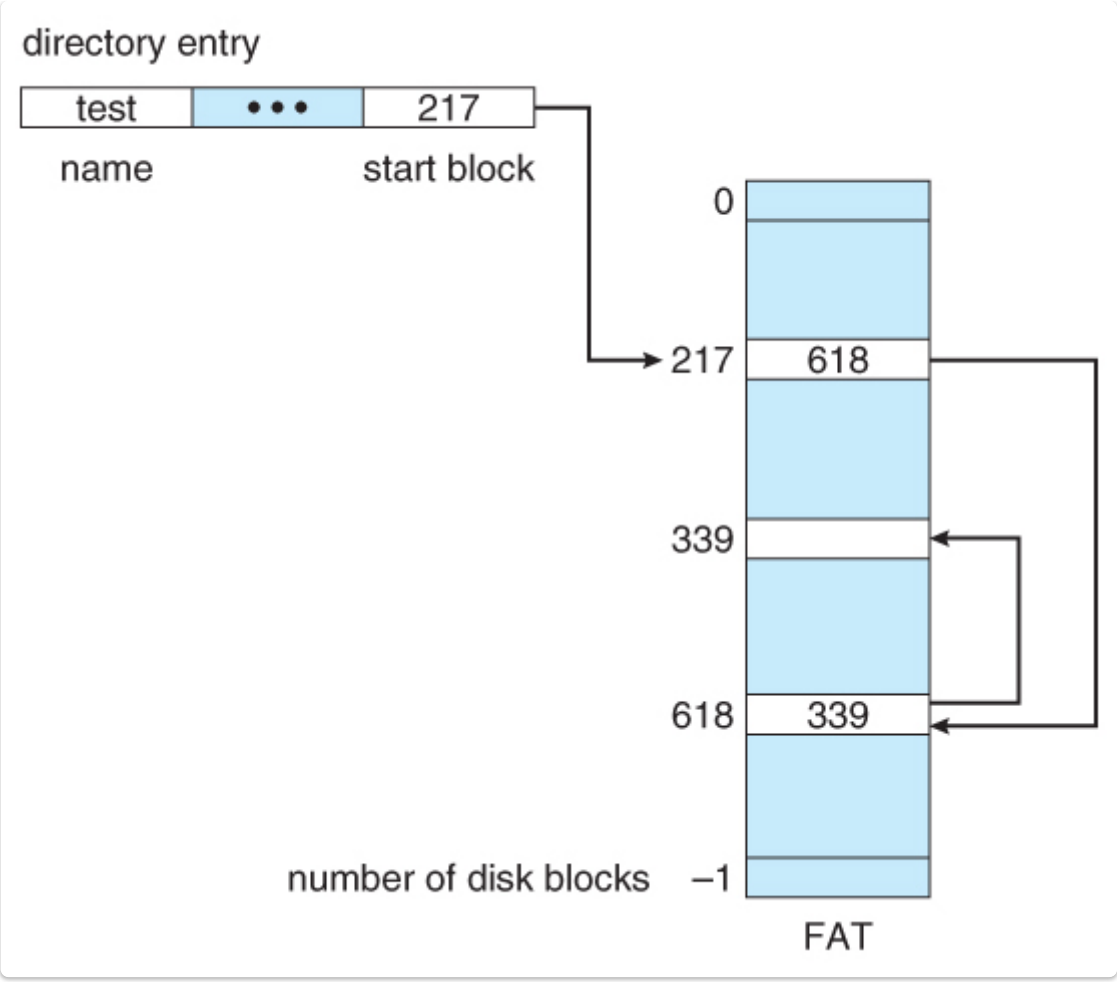
- Files are represented as linked lists of disk blocks, with blocks scattered anywhere on the disk.
- Each file in the directory contains pointers to the first and last blocks of the file.
- Supports dynamic file growth as blocks are allocated on-demand.
- Advantages:
  - Eliminates external fragmentation.
  - Files can grow dynamically without the need for preallocation.
- Disadvantages:
  - Inefficient for direct access, especially for large files.

    - Requires space for pointers, reducing overall disk utilization.
    - Reliability concerns arise from scattered pointers across the disk.
- Variation: File Allocation Table (FAT) used by MS-DOS, where a section of disk contains the table with pointers to disk blocks.



3. Indexed Allocation

- Each file has its index block containing pointers to disk blocks.
- Supports direct access without suffering from external fragmentation.
- Disadvantages:
    - Wasted space due to pointer overhead.
- Implementation Mechanisms:
    - Linked scheme: Index blocks linked together to support large files.
    - Multilevel index: Hierarchical structure of index blocks for large file systems.
    - Combined scheme: Utilizes a combination of direct and indirect blocks, commonly used in UNIX-based file systems.



Free-Space Management

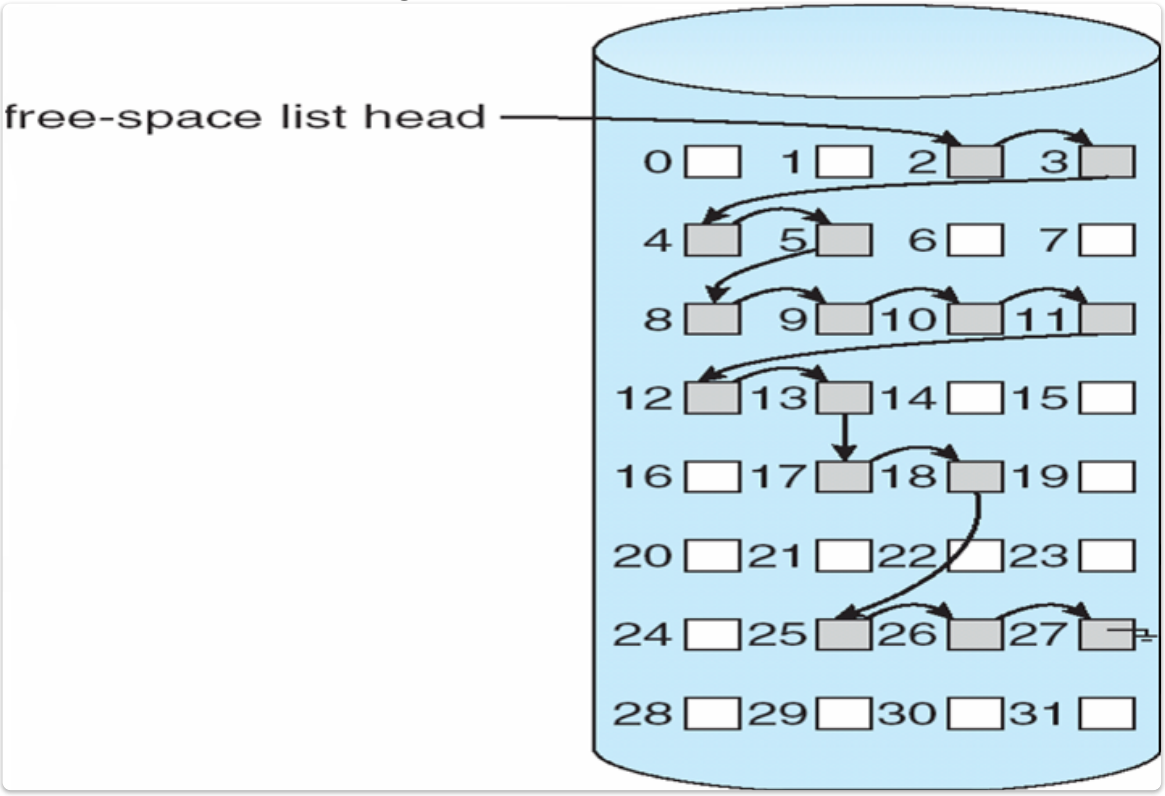
Efficient management of free disk space is crucial for maintaining optimal system performance. Various implementations of free-space management exist, each with its advantages and disadvantages.

1. Bit Vector

- Description: Free-space list represented as a bit map or bit vector, with each block represented by a bit.
- Advantages:
  - Relative simplicity and efficiency in finding the first free block or consecutive free blocks.
- Disadvantages:
  - Inefficient for larger disks unless the entire vector is kept in main memory.
- Example:
  - If blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, the bit map would be `00111100111111000110000011100000...`.

2. Linked List

- Description: Free blocks linked together, with a pointer to the first free block stored on disk.
- Disadvantages:
  - Inefficient traversal; reading each block requires substantial I/O time.



3. Grouping

- Description: Addresses of n free blocks stored in the first free block, with the last block containing addresses of another n free blocks.
- Facilitates quick location of addresses for a large number of free blocks.

4. Counting

- Description: Each entry in the free-space list consists of a disk address and a count of contiguous free blocks.
- Suitable for allocating or freeing contiguous blocks simultaneously.

5. Space Maps

- Description: Utilized by Oracle's ZFS file system to manage huge numbers of files, directories, and file systems.
- Metaslabs divide the space into manageable chunks, each with an associated space map.
- Space map maintains a log of block activity in a balanced-tree structure, enabling efficient allocation and deallocation operations.

Usage of System Calls

Usage of System Calls

File operations in a computer system involve a series of system calls to manage files effectively. These system calls facilitate file creation, reading, writing, closing, deletion, and repositioning within files.

open

2. open()

- Purpose: Opens a file for use.
- Process:
  - Adds the file's entry to the open file table.
  - Maintains an open count associated with each file to track the number of processes with the file open.

create

1. create()

- Purpose: Creates a new file.
- Process:
  - Finds space in the file system for the new file.
  - Adds an entry for the new file in the directory.

read

3. read()

- Purpose: Reads data from a file.
- Process:
  - Specifies the file name and read pointer to determine the read location.
  - Updates the read pointer after reading.

write

4. write()

- Purpose: Writes data to a file.
- Process:
  - Specifies the file name and data to be written.
  - Searches the directory for the file's location.

- Updates the write pointer to indicate the location for the next write.

close

5. close()

- Purpose: Closes a file.
- Process:
  - Decrements the open count associated with the file.
  - Closes the file when the open count reaches zero.

lseek

- Purpose: The `lseek` system call is used to reposition the file offset of an open file descriptor.
- Parameters:
  - File descriptor: Identifies the file whose offset is to be changed.
  - Offset: Specifies the new position in the file.
  - Whence: Indicates the reference point for the offset calculation.
- Functionality:
  - Changes the file offset according to the specified parameters.
  - Allows random access within a file by moving the file pointer to a specified position.
- Usage:

```
off_t lseek(int fd, off_t offset, int whence);
```

stat

- Purpose: The `stat` system call retrieves file attributes for a given file path.
- Parameters:
  - File path: Specifies the path to the file whose attributes are to be retrieved.
  - Stat structure: Stores the file attributes.
- Functionality:
  - Retrieves metadata about a file, including its size, permissions, timestamps, and other details.
- Usage:

```
int stat(const char *pathname, struct stat *statbuf);
```

ioctl

- Purpose: The `ioctl` system call provides input/output control for devices and special files.
- Parameters:
  - File descriptor: Identifies the device or special file.
  - Request code: Specifies the operation to be performed.
  - Additional arguments: Data or parameters required by the operation.
- Functionality:
  - Allows various control operations on devices, such as setting parameters, querying device status, and performing specialized I/O operations.
- Usage:

```
int ioctl(int fd, unsigned long request, ...);
```

Others

6. delete()

- Purpose: Deletes a file from the file system.
- Process:
  - Searches the directory for the named file.
  - Releases all file space associated with the file.
  - Erases the directory entry for the file.

7. truncate()

- Purpose: Resets the contents of a file to length zero while retaining its attributes.
- Process:
  - Allows the file to be reset to length zero and its file space released without altering other attributes.

8. seek() (Reposition)

- Purpose: Repositions the current file position pointer within a file.
- Process:
  - Searches the directory for the appropriate file entry.

- Repositions the current file position pointer to the specified value.

9. unlink()

- **Purpose**: Deletes a name (link) from the file system.
- **Process**:
  - Deletes the file if it was the last link to the file and no processes have the file open.
  - Releases the space occupied by the file for reuse.