OPERATING SYSTEMS
MODULE-5
BCS303

MODULE-5	8 Hours
File System, Implementation of File System: File system: File concept; Access methods; Directory and Disk structure; File system mounting; File sharing; **Implementing File system:** File system structure; File system implementation; Directory implementation; Allocation methods; Free space management.	

Secondary Storage Structure, Protection: Mass storage structures; Disk structure; Disk attachment; Disk scheduling; Disk management; **Protection**: Goals of protection, Principles of protection, Domain of protection, Access matrix.

Textbook 1: Chapter – 10 (10.1-10.5) ,11 (11.1-11.5),12 (12.1-12.5), 14 (14.1-14.4)

FILE CONCEPT

FILE:

- *A* file is a named collection of related information that is recorded on secondary storage.
- The information in a file is defined by its creator. Many different types of information may be stored in a file source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

A file has a certain defined which depends on its type.

- *A text* file is a sequence of characters organized into lines.
- *A source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.
- An *object* file is a sequence of bytes organized into blocks understandable by the system's linker.
- An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *example*.c
- When a file is named, it becomes independent of the process, the user, and even the system that created it.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

1. **Creating a file:** Two steps are necessary to create a file,
   a) Space in the file system must be found for the file.
   b) An entry for the new file must be made in the directory.

2. **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

3. **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file-position pointer.

4. **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. This file operation is also known as files seek.

5. **Deleting a file:** To delete a file, search the directory for the named file. Having found the associated directory entry, then release all file space, so that it can be reused by other files, and erase the directory entry.

6. **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged but lets the file be reset to length zero and its file space released.

- Other common operations include appending new information to the end of an existing file and renaming an existing file.
- Most of the file operations mentioned involve searching the directory for the entry associated with the named file.
- To avoid this constant searching, many systems require that an open () system call be made before a file is first used actively.
- The operating system keeps a small table, called the **open file table** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

- The implementation of the open() and close() operations is more complicated in an environment where several processes may open the file simultaneously
- The operating system uses two levels of internal tables:
    1. A per-process table
    2. A system-wide table

The per-process table:

Tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process.

Each entry in the per-process table in turn points to a system-wide open-file table.

The system-wide table

contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system- wide table includes an entry for the file.

Several pieces of information are associated with an open file.

1. **File pointer:** On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
2. **File-open count:** As files are closed, the operating system must reuse its open- file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
3. **Disk location of the file:**Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
4. **Access rights:** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/0 requests.

File Types

- The operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-**a name and an extension**, usually separated by a period character
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- File types also can be used to indicate the internal structure of the file. For instance source and object files have structures that match the expectations of the programs that read them. Certain files must conform to a required structure that is understood by the operating system.

  For example: the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.

- The operating system support multiple file structures: the resulting size of the operating system also increases. If the operating system defines five different file structures, it needs to contain the code to support these file structures.

- It is necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by

the operating system, severe problems may result.

- Example: The Macintosh operating system supports a minimal number of file structures. It expects files to contain two parts: a resource fork and data fork.
  - **The resource fork** contains information of interest to the user.
  - **The data fork** contains program code or data

Internal File Structure

- Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector.

- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record.

- Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

ACCESS METHODS

- Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

- Some of the common methods are:

Sequential access   Direct access   Other access
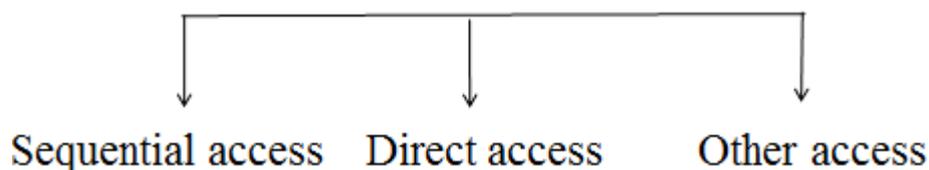
1. Sequential methods

- The simplest access method is sequential methods. Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
- A read operation (next-reads) reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location
- The write operation (write next) appends to the end of the file and advances to the end of the newly written material.
- A file can be reset to the beginning and on some systems, a program may be able to skip forward or backward n records for some integer n-perhaps only for n =1.
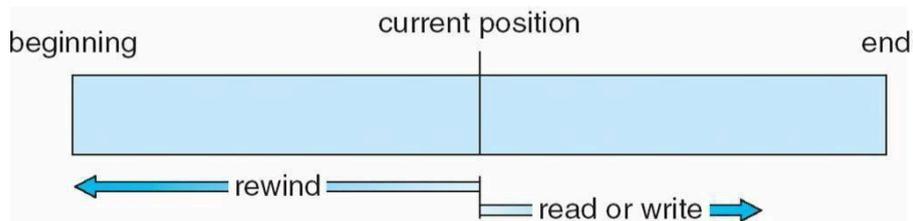
Figure: Sequential-access file.

2. Direct Access

- A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records.
- Example: if we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.
- Direct-access files are of great use for immediate access to large amounts of information such as Databases, where searching becomes easy and fast.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and ·write n rather than write next.
- An alternative approach is to retain read next and write next, as with sequential access, and to add an operation position file to n, where n is the block number. Then, to affect a read n, we would position to n and then read next.

sequential access	implementation for direct access
reset	cp = 0;
read next	read cp; cp = cp + 1;
write next	write cp; cp = cp + 1;

Figure: Simulation of sequential access on a direct-access file.

3. Other Access Methods:

- Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.
- The **Index**, is like an index in the back of a book contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.
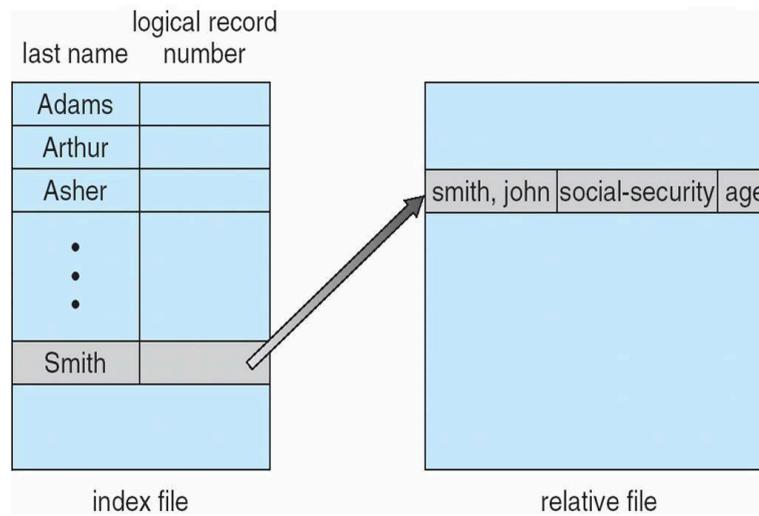
Figure: Example of index and relative files

1. What is a file?
2. Explain the various file attributes.
3. Explain the various file operations.
4. What are the different Access Methods used for accessing information in the file.
5. Differentiate between sequential access and direct access methods.

DIRECTORY AND DISK STRUCTURE

- Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory-based) disks.
- A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control
- Disk can be subdivided into partitions. Each disks or partitions can be RAID protected against failure.
- Partitions also known as minidisks or slices. Entity containing file system known as a volume. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents.**
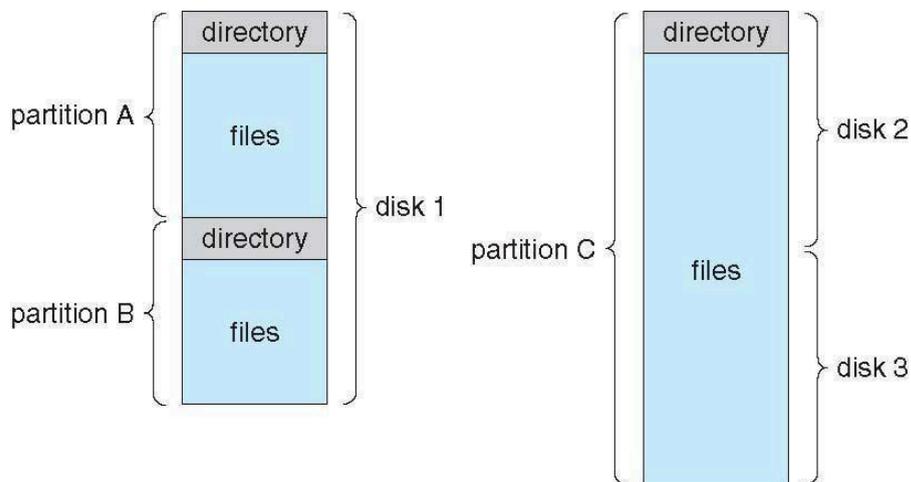

Figure: A Typical File-system Organization

Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries. A directory contains information about the files including attributes location and ownership. To consider a particular directory structure, certain operations on the directory have to be considered:

  - **Search for a file:** Directory structure is searched for a particular file in directory. Files have symbolic names and similar names may indicate a relationship between files. Using this similarity it will be easy to find all whose name matches a particular pattern.

  - **Create a file:** New files needed to be created and added to the directory.

  - **Delete a file:** When a file is no longer needed, then it is able to remove it from the directory.

  - **List a directory:** It is able to list the files in a directory and the contents of the directory entry for each file in the list.

  - **Rename a file:** Because the name of a file represents its contents to its users, It is possible to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

  - **Traverse the file system:** User may wish to access every directory and every file within a directory structure. To provide reliability the contents and structure of the entire file system is saved at regular intervals.

- The most common schemes for defining the logical structure of a directory are described below

    1. Single-level Directory
    2. Two-Level Directory
    3. Tree-Structured Directories
    4. Acyclic-Graph Directories
    5. General Graph Directory

1. Single-level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand
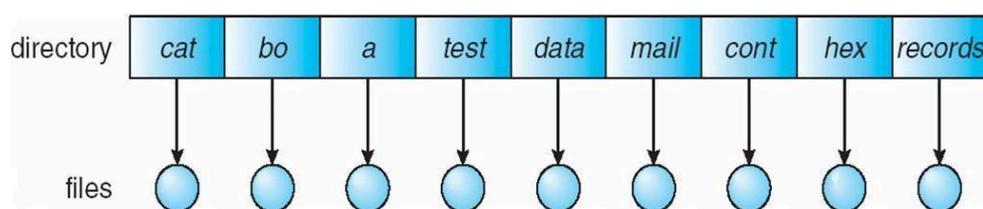
A single-level directory has significant limitations, when the number of files increases or

when the system has more than one user.

As directory structure is single, uniqueness of file name has to be maintained, which is difficult when there are multiple users.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

2. Two-Level Directory

In the two-level directory structure, each user has its own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user.

When a user refers to a particular file, only his own UFD is searched. Different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD thus; it cannot accidentally delete another user's file that has the same name.

When a user job starts or a user logs in, the system's Master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
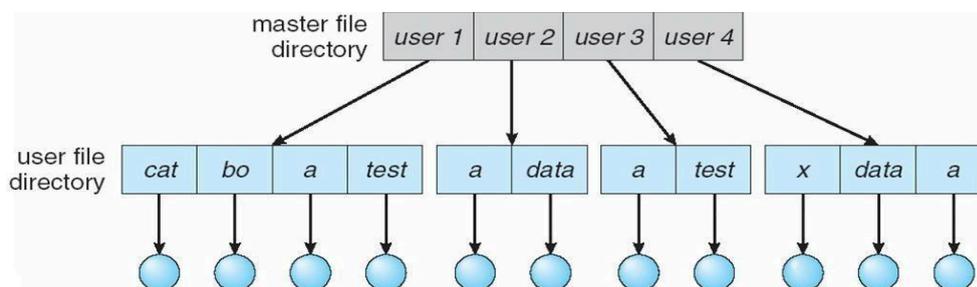
- **Advantage:**
  - No file name-collision among different users.
  - Efficient searching.
- **Disadvantage**
  - Users are isolated from one another and can't cooperate on the same task.

3. Tree Structured Directories

A tree is the most common directory structure.

The tree has a root directory, and every file in the system has a unique path name.

A directory contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.

All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and

delete directories.

Two types of path-names:

1. Absolute path-name: begins at the root.
2. Relative path-name: defines a path from the current directory.
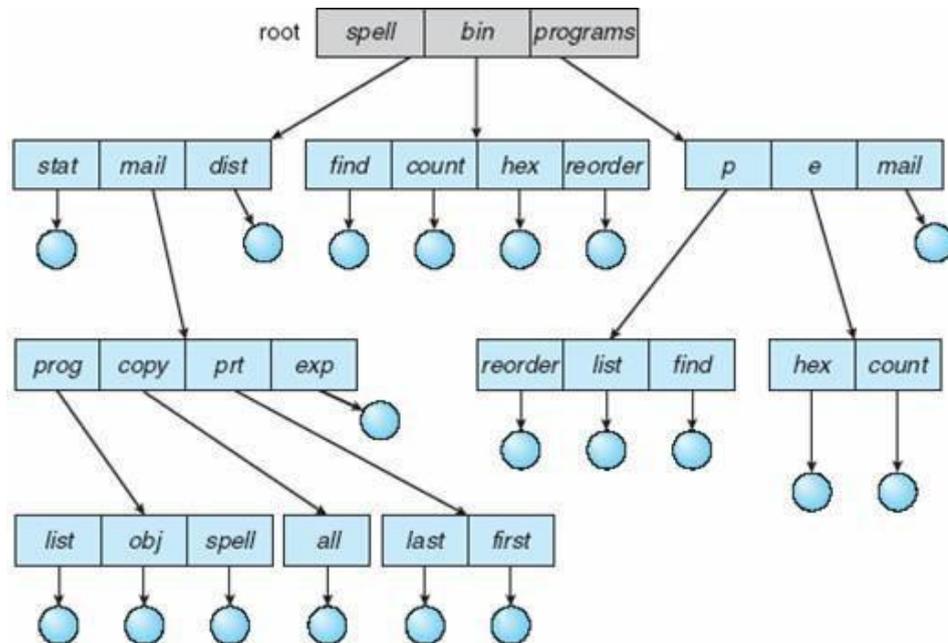
How to delete directory?
1. To delete an empty directory: Just delete the directory.
2. To delete a non-empty directory:
   - First, delete all files in the directory.
   - If any subdirectories exist, this procedure must be applied recursively to them.

Advantage:
- Users can be allowed to access the files of other users.

Disadvantages:
- A path to a file can be longer than a path in a two-level directory.
- Prohibits the sharing of files (or directories).

4. Acyclic Graph Directories

The common subdirectory should be shared. A shared directory or file will exist in the file system in two or more places at once. A tree structure prohibits the sharing of files or directories.

An acyclic graph is a graph with no cycles. It allows directories to share subdirectories and files.
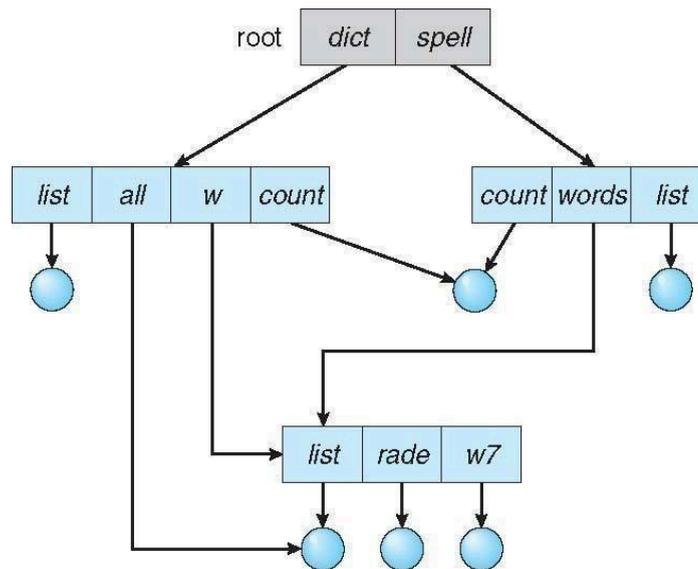
The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

Two methods to implement shared-files (or subdirectories):
1. Create a new directory-entry called a link. A link is a pointer to another file (or subdirectory).
2. Duplicate all information about shared-files in both sharing directories.

Two problems:
1. A file may have multiple absolute path-names.
2. Deletion may leave dangling-pointers to the non-existent file.

Solution to deletion problem:
1. Use back-pointers: Preserve the file until all references to it are deleted.
2. With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

5. General Graph Directory

Problem: If there are cycles, we want to avoid searching components twice.

**Solution:** Limit the no. of directories accessed in a search.

Problem: With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).

Solution: Garbage-collection scheme can be used to determine when the last reference

has been deleted.

Garbage collection involves
- First pass traverses the entire file-system and marks everything that can be accessed.
- A second pass collects everything that is not marked onto a list of free-space
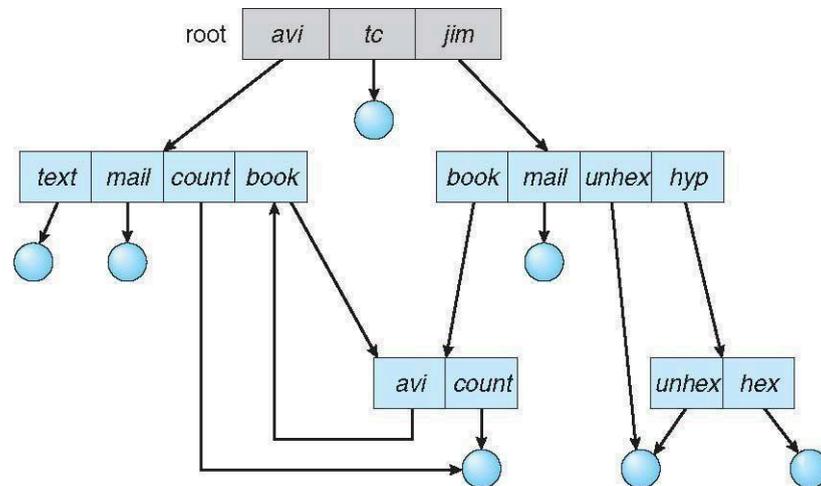
REVIEW QUESTIONS

1. Explain single level directory structure.
2. Explain two level directory structure.
3. Explain various types of directory structures?
4. What are the advantages and disadvantages of single level directory structure.
5. What are the advantages and disadvantages of two-level directory structure.
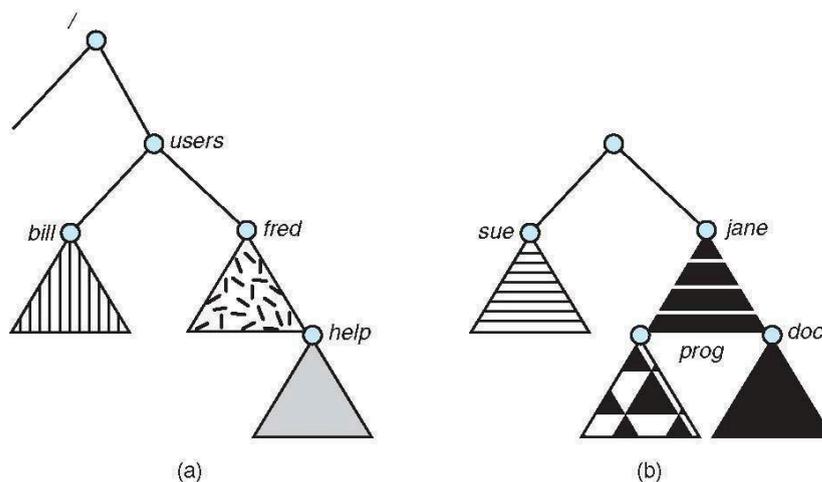
FILE SYSTEM MOUNTING

A file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system

**Mount Point:** The location within the file structure where the file system is to be attached.

The mounting procedure:
- The operating system is given the name of the device and the mount point.
- The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format
- The operating system notes in its directory structure that a file system is mounted at the specified mount point.

To illustrate file mounting, consider the file system shown in figure. The triangles represent sub-trees of directories that are of interest

- Figure (a) shows an existing file system,
- while Figure 1(b) shows an un-mounted volume residing on */device/dsk.* At this point, only the files on the existing file system can be accessed.

Canara Engineering College, Mangaluru
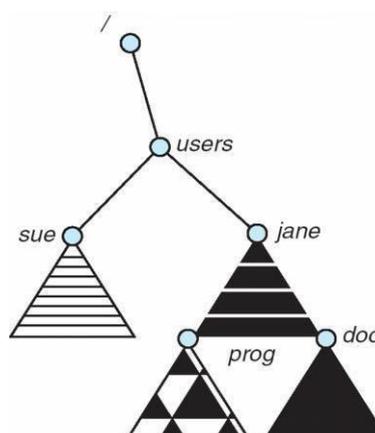
- Above figure shows the effects of mounting the volume residing on *device/dsk* over */users.*
- If the volume is un-mounted, the file system is restored to the situation depicted in first Figure.

FILE SHARING

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through a protection scheme.
- On distributed systems, files may be shared across a network.
- Network File-system (NFS) is a common distributed file-sharing method.

Multiple Users

File-sharing can be done in 2 ways:

1. The system can allow a user to access the files of other users by default or
2. The system may require that a user specifically grant access.

To implement file-sharing, the system must maintain more file- & directory-attributes than on a single-user system.
Most systems use concepts of file owner and group.

1. Owner

- The user who may change attributes & grant access and has the most control over the file (or directory).
- Most systems implement owner attributes by managing a list of user-names and user IDs

2. Group

- The group attribute defines a subset of users who can share access to the file.
- Group functionality can be implemented as a system-wide list of group-names and group IDs.
- Exactly which operations can be executed by group-members and other users is definable by the file's owner.
- The owner and group IDs of files are stored with the other file-attributes and can be used to allow/deny requested operations.

Remote File Systems

It allows a computer to mount one or more file-systems from one or more remote- machines.
There are three methods:

1. Manually transferring files between machines via programs like ftp.
2. Automatically DFS (Distributed file-system): remote directories are visible from a local machine.
3. Semi-automatically via www (World Wide Web): A browser is needed to gain access to

the remote files, and separate operations (a wrapper for ftp) are used to transfer files. ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system.

Client Server Model

- Allows clients to mount remote file-systems from servers.
- The machine containing the files is called the **server**. The machine seeking access to the files is called the **client**.
- A server can serve multiple clients, and a client can use multiple servers.
- The server specifies which resources (files) are available to which clients.
- A client can be specified by a network-name such as an IP address.

Disadvantage:

- Client identification is more difficult.
- In UNIX and its NFS (network file-system), authentication takes place via the client networking information by default.
- Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

Distributed Information Systems

- Provides unified access to the information needed for remote computing.
- The DNS (domain name system) provides hostname-to-network address translations.
- Other distributed info. systems provide username/password space for a distributed facility

Failure Modes

- Local file-systems can fail for a variety of reasons such as failure of disk (containing the file-system), corruption of directory-structure & cable failure.
- Remote file-systems have more failure modes because of the complexity of network-systems.
- The network can be interrupted between 2 hosts. Such interruptions can result from hardware failure, poor hardware configuration or networking implementation issues.
- DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.
- To implement failure-recovery, some kind of state information may be maintained on both the client and the server.

Consistency Semantics

- These represent an important criterion of evaluating file-systems that supports file-sharing. These specify how multiple users of a system are to access a shared-file simultaneously.
- In particular, they specify when modifications of data by one user will be observed by

other users.

- These semantics are typically implemented as code with the file-system.
- These are directly related to the process-synchronization algorithms.
- A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

UNIX Semantics

- UNIX file-system (UFS) uses the following consistency semantics:
    1. Writes to an open-file by a user are visible immediately to other users who have this file opened.
    2. One mode of sharing allows users to share the pointer of current location into a file. Thus, the advancing of the pointer by one user affects all sharing users.
- A file is associated with a single physical image that is accessed as an exclusive resource.
- Contention for the single image causes delays in user processes.

Session Semantics

The AFS uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open.
2. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

- A file may be associated temporarily with several (possibly different) images at the same time.
- consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.
- Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has 2 key properties:
    1. File-name may not be reused
    2. File-contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined

PROTECTION

When information is stored in a computer system, we want to keep it safe from physical

damage (reliability) and improper access (protection).
- Reliability is generally provided by duplicate copies of files.
- For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.
- File owner/creator should be able to control what can be done and by whom.

Types of Access
- Systems that do not permit access to the files of other users do not need protection. This is too extreme, so controlled-access is needed.
- Following operations may be controlled:
    1. *Read:* Read from the file.
    2. *Write:* Write or rewrite the file.
    3. *Execute:* Load the file into memory and execute it.
    4. *Append:* Write new information at the end of the file.
    5. *Delete:* Delete the file and tree its space for possible reuse.
    6. *List:* List the name and attributes of the file.

Access Control
- Common approach to protection problem is to make access dependent on identity of user.
- Files can be associated with an ACL (access-control list) which specifies username and types of access for each user.

Problems:
1. Constructing a list can be tedious.
2. Directory-entry now needs to be of variable-size, resulting in more complicated space management.

Solution:

These problems can be resolved by combining ACLs with an 'owner, group, universe' access control scheme

To reduce the length of the ACL, many systems recognize 3 classifications of users:
1. *Owner:* The user who created the file is the owner.
2. *Group:* A set of users who are sharing the file and need similar access is a group.
3. *Universe:* All other users in the system constitute the universe.

Other Protection Approaches

A password can be associated with each file.

1. The no. of passwords you need to remember may become large.
2. If only one password is used for all the files, then all files are accessible if it is discovered.
3. Commonly, only one password is associated with all of the user's files, so protection is all-or nothing.

In a multilevel directory-structure, we need to provide a mechanism for directory protection.

The directory operations that must be protected are different from the File-operations:

1. Control creation & deletion of files in a directory.
2. Control whether a user can determine the existence of a file in a directory.

IMPLEMENTATION OF FILE SYSTEM
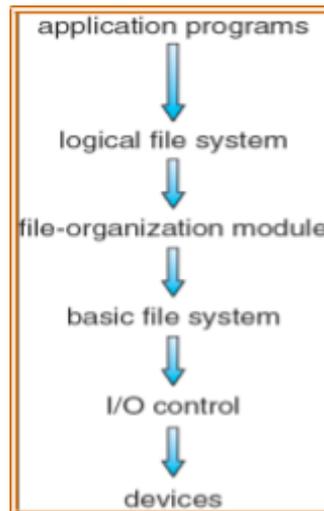
FILE SYSTEM STRUCTURE
- Disks provide the bulk of secondary-storage on which a file-system is maintained.

The disk is a suitable medium for storing multiple files.
- This is because of two characteristics
  1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors. Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes. The usual size is 512 bytes.
- File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
- Design problems of file-systems:
  1. Defining how the file-system should look to the user.
  2. Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

Layered File Systems:
- The file-system itself is generally composed of many different levels. Every level in design uses features of lower levels to create new features for use by higher levels.

- File system provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

A file system poses two quite different design problems.
1. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
2. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

- The lowest level, **the *I/O control*,** consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk system.

A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123."
Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/0 device to the rest of the system.
- The device driver usually writes specific bit patterns to special locations in the I/0 controller's memory to tell the controller which device location to act on and what actions to take.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector10).

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks.

A block in the buffer is allocated before the transfer of a disk block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.

File organization

- Module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file- organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Each file's logical blocks are numbered from 0 (or 1) through N. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.
- The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical file system

- Manages metadata information. Metadata includes all of the file- system structure except the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name. It maintains file structure via **file-control blocks (FCB)**.
- FCB contains information about the file, including ownership, permissions, and location of the file contents.

File System Implementation

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- **Boot Control Block**: On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- **Volume Control Block** : (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.
- **A directory structure** (per file system) is used to organize the files.
- **A per-file FCB** contains many details about the file. It has a unique identifier number to allow association with a directory entry.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system

operations, and discarded at dismount. Several types of structures may be included.
- An in-memory mount table contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories.

- The system wide open file table contains a copy of the FCB of each open file, as well as other information.
- The per -process open file table contains a pointer to the appropriate entry in the system- wide open-file table, as well as other information.
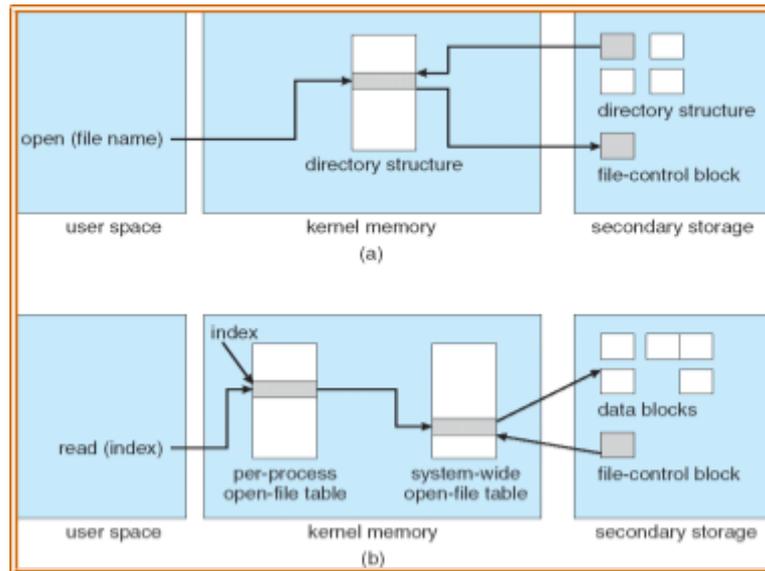
- Buffers hold file-system blocks when they are being read from disk or written to disk.

Steps for creating a file:
1) An application program calls the logical file system, which knows the format of the directory structures
2) The logical file system allocates a new file control block(FCB)
   ▪ If all FCBs are created at file-system creation time, an FCB is allocated from the free list
3) The logical file system then
   ▪ Reads the appropriate directory into memory
   ▪ Updates the directory with the new file name and FCB
   ▪ Writes the directory back to the disk

UNIX treats a directory exactly the same as a file by means of a type field in the i node Windows NT implements separate system calls for files and directories and treats directories as entities separate from files.

Steps for opening a file:
1) The function first searches the system-wide open-file table to see if the file is already in use by another process
   ▪ If it is, a per-process open-file table entry is created pointing to the existing

system-wide open-file table
- This algorithm can have <u>substantial overhead</u>; consequently, parts of the directory structure are usually <u>cached in memory</u> to speed operations

2) Once the file is found, the FCB is copied into a system-wide open-file table in memory
- This table also tracks the number of processes that have the file open

3) Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table

4) The function then returns a pointer/index to the appropriate entry in the per-process file-system table
- All subsequent file operations are then performed via this pointer
- UNIX refers to this pointer as the <u>file descriptor</u>
- Windows refers to it as the <u>file handle</u> Steps for closing a file:

1) The per-process table entry is removed
2) The system-wide entry's open count is decremented
3) When all processes that have opened the file eventually close it

Any updated <u>metadata</u> is copied back to the disk-based directory structure. The system-wide open-file table entry is removed

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

Partitions and Mounting

Each partition can be either "raw," containing no file system, or "cooked," containing a file system.

Raw disk is used where no file system is appropriate.

UNIX swap space can use a raw partition, for example, as it uses its own format on disk and does not use a file system.

Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.

Boot information is a sequential series of blocks, loaded as an image into memory.

Execution of the image starts at a predefined location, such as the first byte. This boot loader in turn knows about the file-system structure to be able to find and load the kernel and start it executing.

The root partition which contains the operating-system kernel and sometimes other system files, is mounted at boot time.

Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.

After successful mount operation, the operating system verifies that the device contains a valid file system.

It is done by asking the device driver to read the device directory and verifying that the directory has the expected format.

If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.

Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system.

The root partition is mounted at boot time
- It contains the operating-system kernel and possibly other system files

Other volumes can be automatically mounted at boot time or manually mounted later

As part of a successful mount operation, the operating system verifies that the storage device contains a valid file system
- It asks the device driver to read the device directory and verify that the directory has the expected format
- If the format is invalid, the partition must have its consistency checked and possibly corrected
- Finally, the operating system notes in its in-memory mount table structure that a file system is mounted along with the type of the file system

Virtual file Systems

The file-system implementation consists of three major layers, as depicted schematically in Figure. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors.

The second layer is called the virtual file system (vfs) layer. The VFS layer serves two important functions:

It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.

It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.

The kernel maintains one vnode structure for each active node.

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
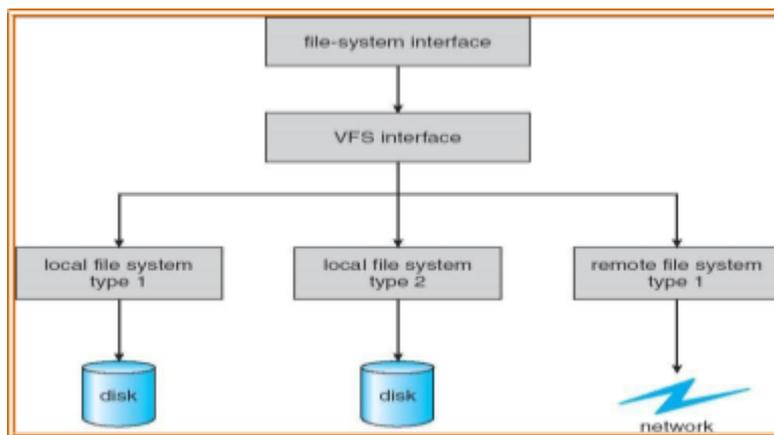
<mark>REVIEW QUESTIONS</mark>

1.Describe the steps involved in mounting a file system.
2. What are the advantages of file sharing in multi-user systems?
3.What is the role of the client-server model in remote file systems, and what are its drawbacks?
4. What are the six types of access that can be controlled in a file protection scheme?
5. Explain the two major design problems in file system implementation.
6. What are the primary functions of the virtual file system (VFS) layer?

Directory Implementation

Selection of directory allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system

One Approach: Direct indexing of a linear list

Consists of a list of file names with pointers to the data blocks

Simple to program

Time-consuming to search because it is a linear search.

Sorting the list allows for a binary search; however, this may complicate creating and deleting files

To create a new file, we must first search the directory to be sure that no existing file has the same name.

Add a new entry at the end of the directory.

To delete a file, we search the directory for the named file and then release the space allocated to it.

To reuse the directory entry, we can do one of several things. Mark the entry as unused.

An alternative is to copy the last entry in the directory into the freed location and to

decrease the length of the directory.

Directory information is used frequently, and users will notice if access to it is slow.

Another Approach: List indexing via a hash function

Takes a value computed from the file name and returns a pointer to the file name in the linear list

Greatly reduces the directory search time

Can result in collisions – situations where two file names hash to the same location

A hash table are its generally fixed size and the dependence of the hash function on that size. (i.e., fixed number of entries).

Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

ALLOCATION METHODS

Allocation methods address the problem of allocating space to files so that disk space is utilized effectively and files can be accessed quickly.

Three methods exist for allocating disk space

Contiguous allocation

Linked allocation

Indexed allocation

Contiguous allocation:

Requires that each file occupy a set of contiguous blocks on the disk

Accessing a file is easy – only need the starting location (block #) and length (number of blocks)

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b, then it occupies blocks b, $b + 1$, $b + 2$, ... ,$b + n$ - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block $b + i$. Thus, both sequential and direct access can be supported

by contiguous allocation.
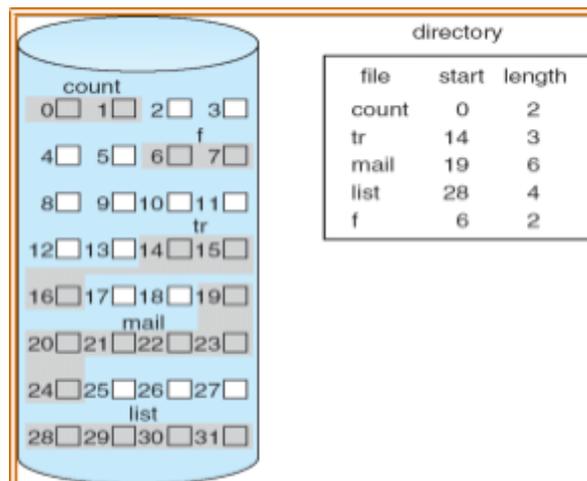
Disadvantages:

1. Finding space for a new file is difficult. The system chosen to manage free space determines how this task is accomplished. Any management system can be used, but some are slower than others.
2. Satisfying a request of size *n* from a list of free holes is a problem. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
3. The above algorithms suffer from the problem of external fragmentation.
   ▪ As files are allocated and deleted, the free disk space is broken into pieces.
   ▪ External fragmentation exists whenever free space is broken into chunks.
   ▪ It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
   ▪ Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.
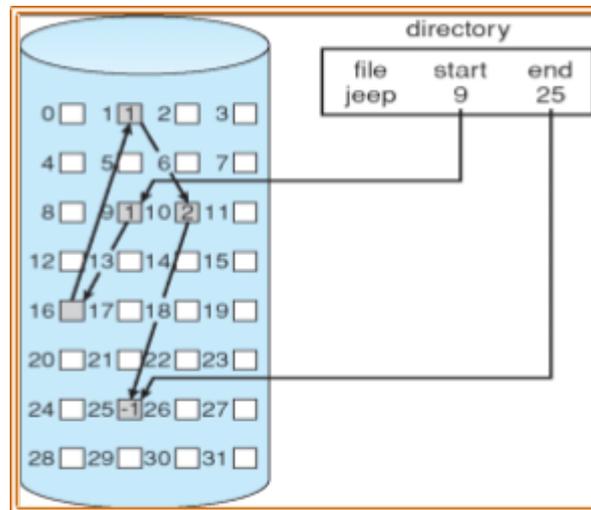
Linked Allocation:

- Solves the problems of contiguous allocation
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of a file
- Creating a new file requires only creation of a new entry in the directory
- Writing to a file causes the free-space management system to find a free block

- This new block is written to and is linked to the end of the file
- Reading from a file requires only reading blocks by following the pointers from block to block.

Advantages

- There is no external fragmentation

- Any free blocks on the free list can be used to satisfy a request for disk space
- The size of a file need not be declared when the file is created
- A file can continue to grow as long as free blocks are available
- It is never necessary to compact disk space for the sake of linked   allocation (however, file access efficiency may require it)

Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. A disk address (the pointer) requires 4 bytes in the disk.

To **create** a new file, we simply create a new entry ile the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.

A **write** to the file causes the free-space management system to filed a free block, and this new block is written to and is linked to the end of the file.

To **read** a file, we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when that file is created.

A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Disadvantages:

1. The major problem is that it can be used effectively only for sequential-access files. To filed the i th block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.
2. Space required for the pointers. Solution is clusters. Collect blocks into multiples and allocate clusters rather than blocks
3. Reliability - the files are linked together by pointers scattered all over the disk and if a pointer were lost or damaged then all the links are lost.

File Allocation Table:

A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.

The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.

The table entry indexed by that block number contains the block number of the next block in the file.

The chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

An unused block is indicated by a table value of 0.

Consider a FAT with a file consisting of disk blocks 217, 618, and 339.
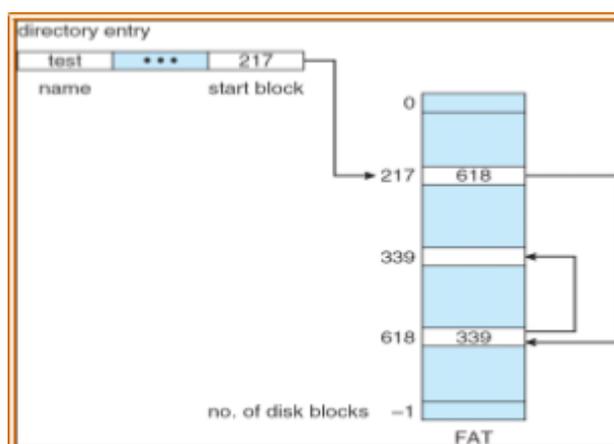
Indexed allocation:

Brings all the pointers together into one location called index block.

Each file has its own index block, which is an array of disk-block addresses.

The *ith* entry in the index block points to the *ith* block of the file. The directory contains the address of the index block. To find and read the *ith* block, we use the pointer in the *ith* index- block entry.

When the file is created, all pointers in the index block are set to *nil.* When the ith block is first written, a block is obtained from the free-space manager and its address is put in the ith index- block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.

Disadvantages :

- Suffers from some of the same performance problems as linked allocation
- Index blocks can be cached in memory; however, data blocks may be spread all over the disk volume.
- Indexed allocation does suffer from wasted space.
- The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
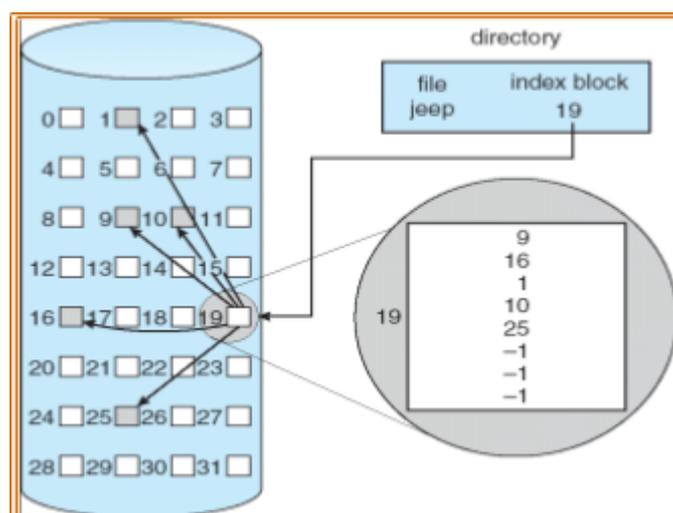
If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

a)   **Linked scheme.** An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

b)   **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size
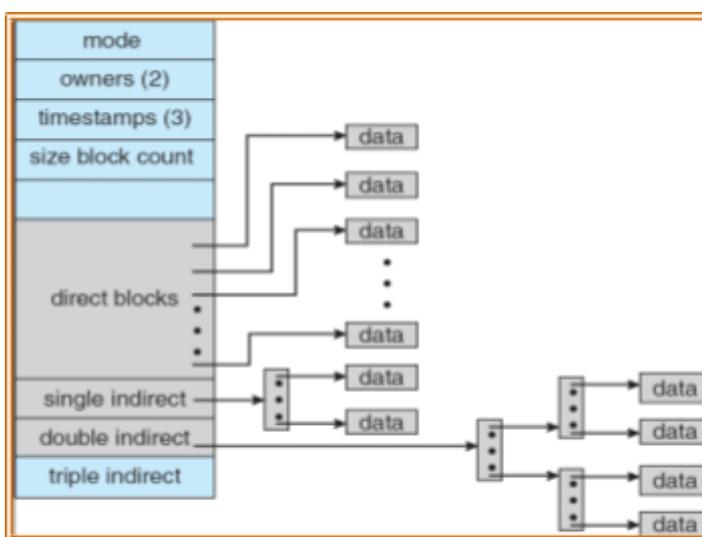
c)   **Combined scheme.** For eg. 15 pointers of the index block is maintained in the file's i node. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**.

Performance

- Contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the *ith* block and read it directly.
- For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access. Linked allocation should

not be used for an application requiring direct access.

- **Indexed allocation** is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block.

Free Space Management

The space created after deleting the files can be reused. Another important aspect of disk management is keeping track of free space in memory. The list which keeps track of free space in memory is called the free-space list. To create a file, search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, is implemented in different ways as explained below.

a) Bit Vector
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

For example, consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17and 18 are free, and the rest of the blocks are allocated. The free-space bit map would be
0011110011111100011

- Easy to implement and also very efficient in finding the first free block or 'n' consecutive free blocks on the disk.
- The down side is that a 40GB disk requires over 5MB just to store the bitmap.

b) Linked List
a. A linked list can also be used to keep track of all free blocks.
b. Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
c. The FAT table keeps track of the free list as just one more linked list on the table.

c) Grouping
a. A variation on linked list free lists. It stores the addresses of n free blocks in the first free block. The first n-1 blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
b. The address of a large number of free blocks can be found quickly.

d) Counting
a. When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.
b. Rather than keeping al list of n free disk addresses, we can keep the address of first free block and the number of free contiguous blocks that follow the first block.

   c. Thus the overall space is shortened. It is similar to the extent method of allocating blocks.

e) Space Maps

   a. Sun's ZFS file system was designed for huge numbers and sizes of files, directories, and even file systems.

   b. The resulting data structures could be inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.

   c. ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) *Meta slabs* of a manageable size, each having their own space map.

   d. Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.

   e. An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.

   f. The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

REVIEW QUESTIONS

1. What are the main steps involved in creating a new file using the linear list approach?
2. What are the limitations of hash tables in file systems?
3. Why is contiguous allocation efficient for sequential and direct access?
4. What are the advantages of linked allocation over contiguous allocation?
5. How does File Allocation Table store the structure of a file?
6. What are the key differences between indexed allocation and linked allocation?
7. What are the challenges of using a linked list for free-space management?
8. Discuss how external fragmentation impacts file system performance in different allocation methods.

<p style="text-align:center"><mark>SESSION</mark></p>

SECONDARY STORAGE STRUCTURES

OVERVIEW OF MASS-STORAGE STRUCTURE

Web link- https://youtu.be/ZjMwUhapSEM

Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems.
- Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches.
- The two surfaces of a platter are covered with a magnetic material. The information stored by recording it magnetically on the platters.
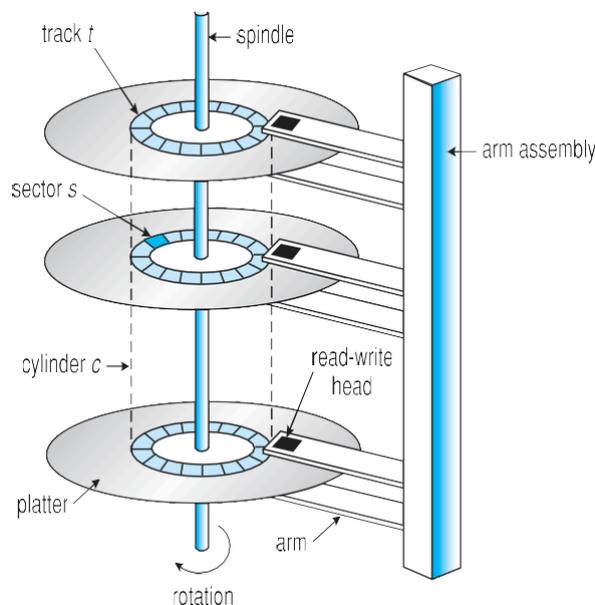

Figure: Moving-head disk mechanism

- The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. Sector is the basic unit of storage. The set of tracks that are at one arm position makes up a cylinder.
- The number of cylinders in the disk drive equals the number of tracks in each platter.
- There may be thousands of concentric cylinders in a disk drive, and each track may

contain hundreds of sectors.

- o **<u>Seek Time:-</u>**Seek time is the time required to move the disk arm to the required track.
- o **<u>Rotational Latency (Rotational Delay):-</u>** Rotational latency is the time taken for the disk to rotate so that the required sector comes under the r/w head.
- o **<u>Positioning time or random access time</u>** is the summation of seek time and rotational delay.
- o **<u>Disk Bandwidth:-</u>** Disk bandwidth is the total number of bytes transferred divided by total time between the first request for service and the completion of last transfer.
- o **<u>Transfer rate</u>** is the rate at which data flow between the drive and the computer.

As the disk head flies on an extremely thin cushion of air, the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **<u>head crash</u>**.

Magnetic Tapes

- Magnetic tape is a secondary-storage medium. It is a permanent memory and can hold large quantities of data.
- The time taken to access data (access time) is large compared with that of magnetic disk, because here data is accessed sequentially.
- When the nth data has to be read, the tape starts moving from first and reaches the nth position and then data is read from n^{th} position. It is not possible to directly move to the nth position. So tapes are used mainly for backup, for storage of infrequently used information.
- A tape is kept in a spool and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.

DISK STRUCTURE

- Modern disk drives are addressed as a large one-dimensional array. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially.
- Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The disk structure (architecture) can be of two types –
1. Constant Linear Velocity (CLV)

2. Constant Angular Velocity (CAV)

1. **CLV** – The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. This architecture is used in CD-ROM and DVD-ROM.

2. **CAV** – There is same number of sectors in each track. The sectors are densely packed in
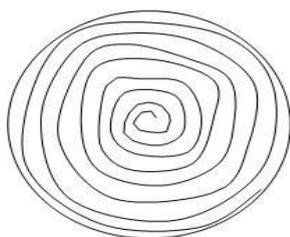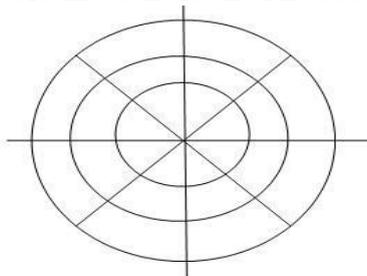
CLV   CAV

the inner tracks. The density of bits decreases from inner tracks to outer tracks to keep the data rate constant.

DISK ATTACHMENT

Computers can access data in two ways.
1. via I/O ports (or host-attached storage)
2. via a remote host in a distributed file system (or network-attached storage)

1. Host-Attached Storage:

● Host-attached storage is storage accessed through local I/O ports.
● Example: the typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus.
● The other cabling systems are – SATA (Serially Attached Technology Attachment), SCSI (Small Computer System Interface) and fiber channel (FC).
● SCSI is a bus architecture. Its physical medium is usually a ribbon cable. FC is a high- speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. An improved version of this architecture is the basis of storage-area networks (SANs).

2. Network-Attached Storage

- A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a network as shown in the figure.
- Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network usually the same local-area network (LAN) carries all data traffic to the clients.
- Network- attached storage provides a convenient way for all the computers on a LAN to share a pool of storage files.
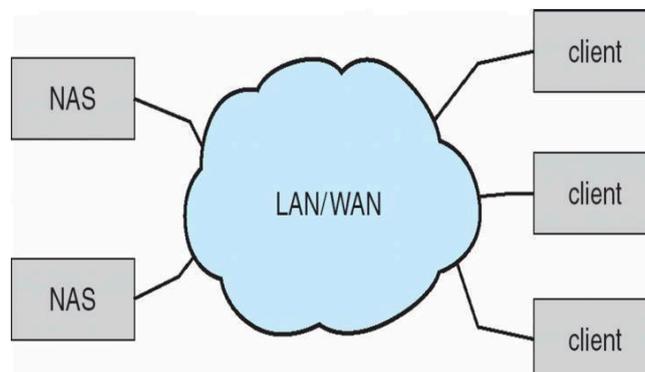
3. Storage Area Network (SAN)

- A storage-area network (SAN) is a private network connecting servers and storage units.
- The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts.
- A SAN switch allows or prohibits access between the hosts and the storage. Fiber Chanel is the most common SAN interconnect.
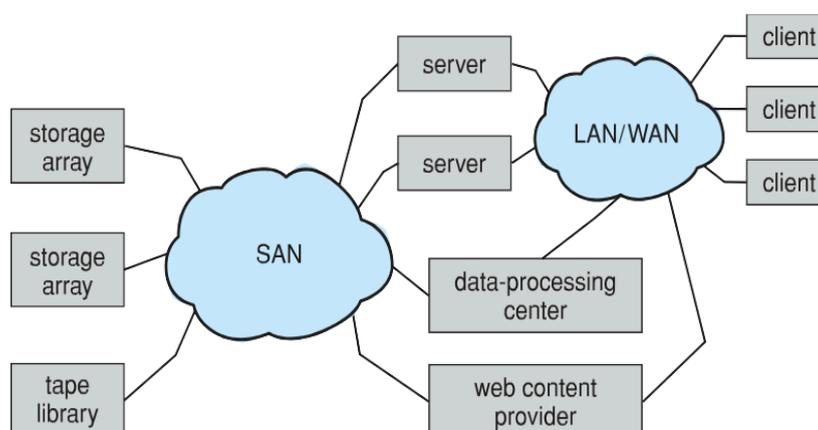
REVIEW QUESTIONS

1. What is the role of a magnetic material on the surface of a disk platter?
2. Define the terms *seek time* and *rotational latency*. How do they contribute to *positioning time*?
3. Differentiate between *Constant Linear Velocity (CLV)* and *Constant Angular Velocity (CAV)* disk structures.
4. How does the structure of a disk affect its performance in terms of storage and access times?
5. Explain the concept of *logical blocks* in modern disk drives and their mapping onto physical sectors.

SESSION

DISK SCHEDULING

Different types of disk scheduling algorithms are as follows:
1. FCFS (First Come First Serve)
2. SSTF (Shortest Seek Time First)
3. SCAN (Elevator)
4. C-SCAN
5. LOOK
6. C-LOOK

1. FCFS scheduling algorithm:

This is the simplest form of disk scheduling algorithm. This services the request in the order they are received. This algorithm is fair but do not provide fastest service. It takes no special care to minimize the overall seek time.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67
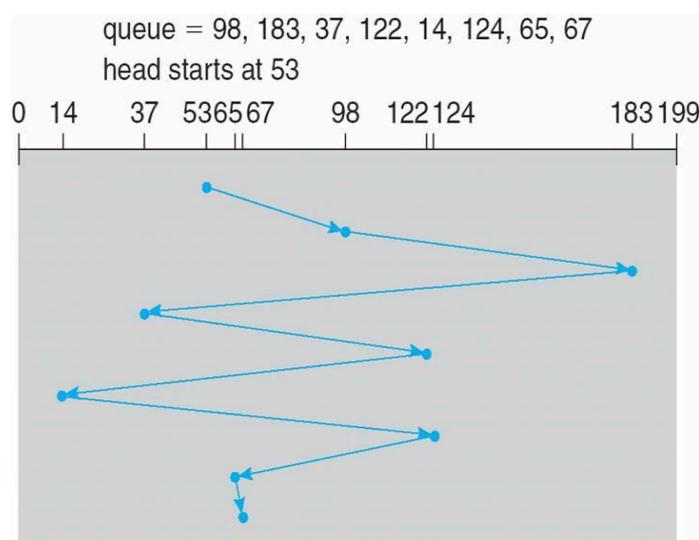
If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 67 for a total head movement of 640 cylinders. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.

Web link: https://youtu.be/hSaPhBtU_BA

2. SSTF (Shortest Seek Time First) algorithm:

This selects the request with minimum seek time from the current head position. SSTF chooses the pending request closest to the current head position.

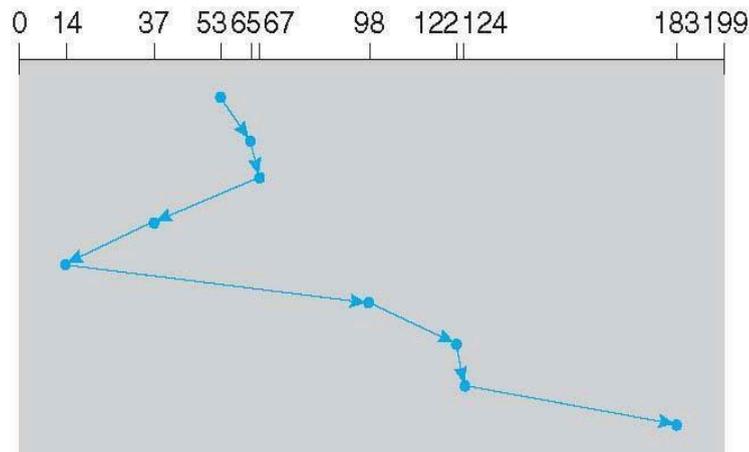Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer than 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183. The total head movement is only 236 cylinders. SSTF is a substantial improvement over FCFS, it is not optimal.

Web link: https://youtu.be/DOnrmOGzooA

3. **SCAN algorithm:**

In this the disk arm starts moving towards one end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues. The initial direction is chosen depending upon the direction of the head.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67
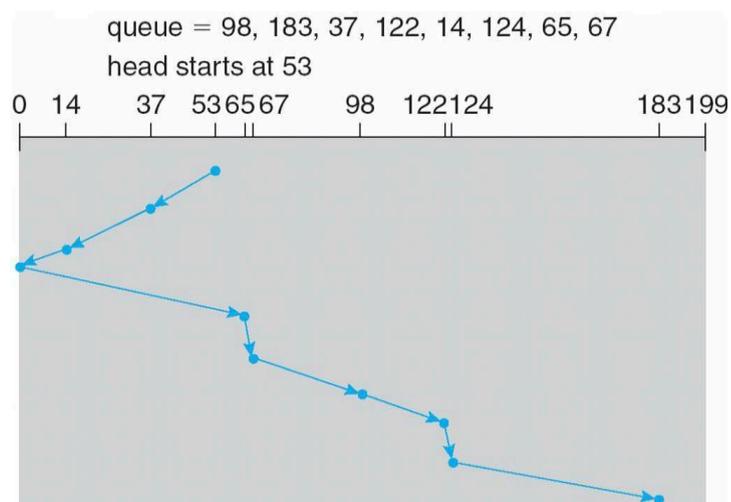
queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move

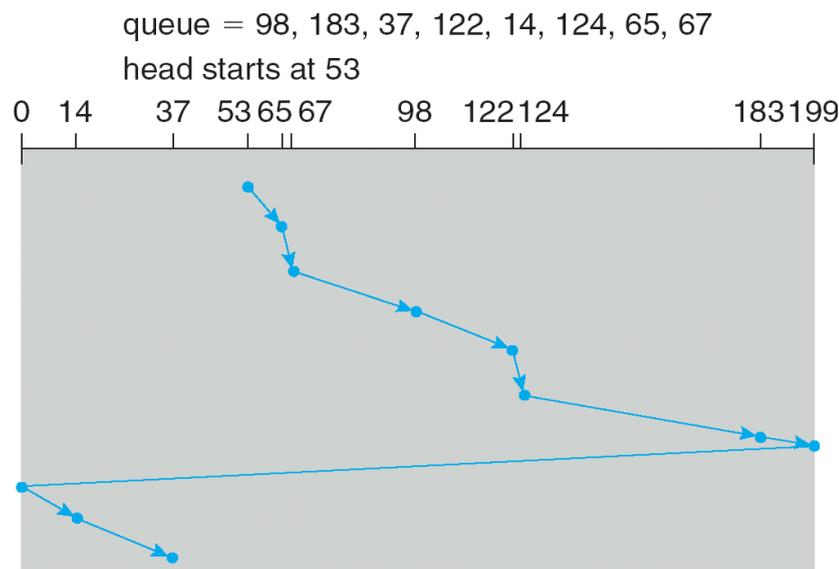towards the other end of the disk servicing 37 and then 14. The SCAN is also called as elevator algorithm

Web link: https://youtu.be/lQAYseBJPuA

4. **C-SCAN (Circular scan) algorithm**:

C-SCAN is a variant of SCAN designed to provide a more uniform wait time.

Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67.

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move immediately towards the other end of the disk, then changes the direction of head and serves 14 and then 37.

Note: If the disk head is initially at 53 and if the head is moving towards track 0, it services 37 and 14 first. At cylinder 0 the arm will reverse and will move immediately towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183.

Web link: https://youtu.be/EgUctpbHUsQ

5. **Look Scheduling algorithm:**

Look and C-Look scheduling are different version of SCAN and C-SCAN respectively. Here the arm goes only as far as the final request in each direction. Then it reverses, without going all the way to the end of the disk. The Look and C-Look scheduling look for a request before continuing to move in a given direction.

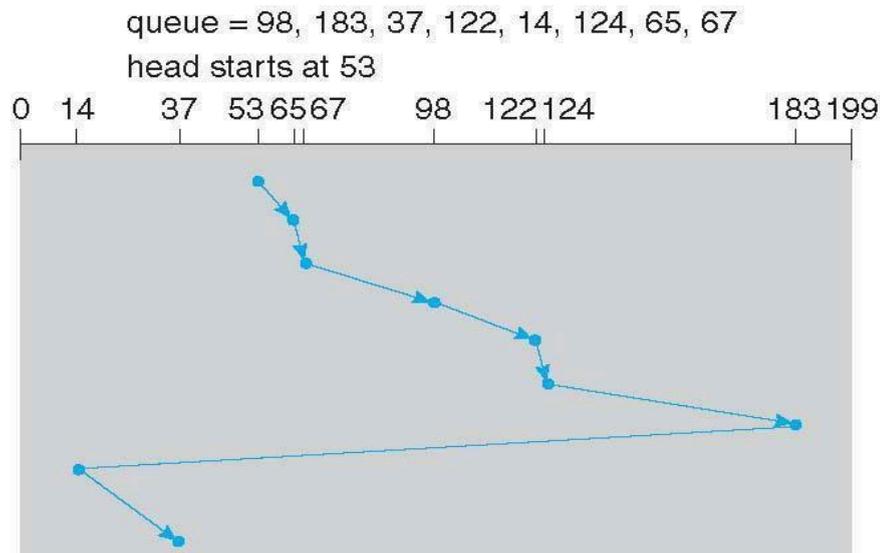Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

Figure: C-LOOK disk scheduling.

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At the final request 183, the arm will reverse and will move towards the first request 14 and then serves 37.

SELECTION OF A DISK-SCHEDULING ALGORITHM

- SSTF is commonly used and it increases performance over FCFS.
- SCAN and C-SCAN algorithm is better for a heavy load on disk. SCAN and C-SCAN have less starvation problem.
- SSTF or Look is a reasonable choice for a default algorithm.
- Selection of disk scheduling algorithm is influenced by the file allocation method, if contiguous file allocation is chosen, then FCFS is best suitable, because the files are stored in contiguous blocks and there will be limited head movements required.
- A linked or indexed file may include blocks that are widely scattered on the disk, resulting in greater head movement.
- The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently.
- Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. The disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

DISK MANAGEMENT

Disk Formatting

- The process of dividing the disk into sectors and filling the disk with a special data structure is called low-level formatting. Sector is the smallest unit of area that is read / written by the disk controller. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size) and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error- correcting code (ECC).
- When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When a sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.
- Most hard disks are low-level- formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk.
- When the disk controller is instructed for low-level-formatting of the disk, the size of data block of all sector sit can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is of sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data.

The operating system needs to record its own data structures on the disk. It does so in two steps i.e., Partition and logical formatting.

1. **Partition** – is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files.
2. **Logical formatting (or creation of a file system)** - Now, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters.**

Boot Block

When a computer is switched on or rebooted, it must have an initial program to run. This is called the bootstrap program.

The bootstrap program –

- Initializes the CPU registers, device controllers, main memory, and then starts the operating system.
- Locates and loads the operating system from the disk
- Jumps to beginning the operating-system execution.

The bootstrap is stored in read-only memory (ROM). Since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips. So most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.
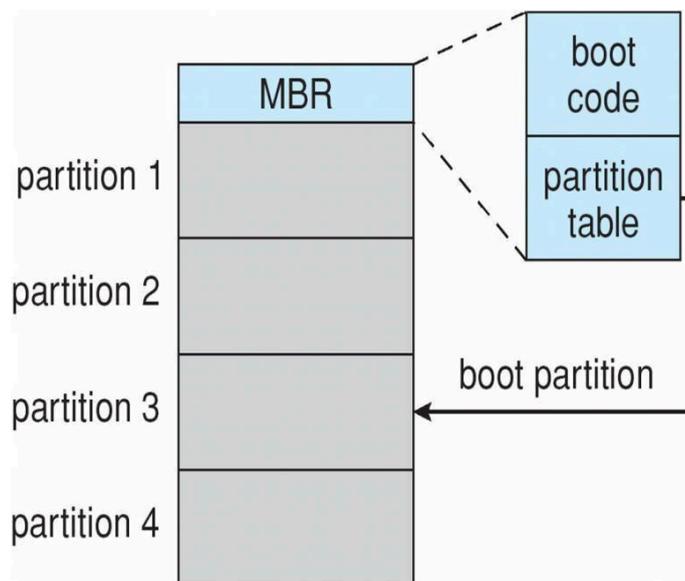

Figure: Booting from disk in Windows 2000.

The Windows 2000 system places its boot code in the first sector on the hard disk (master boot record, or MBR). The code directs the system to read the boot code from, the MBR. In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.

Bad Blocks

Disks are prone to failure of sectors due to the fast movement of r/w head. Sometimes the whole disk will be changed. Such group of sectors that are defective are called as **bad blocks**.

Different ways to overcome bad blocks are -
- Some bad blocks are handled manually, eg. In MS-DOS.
- Some controllers replace each bad sector logically with one of the spare sectors (extra sectors). The schemes used are sector sparing or forwarding and sector slipping.

In MS-DOS format command, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block.
In SCSI disks, bad blocks are found during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing or forwarding**.

A typical bad-sector transaction might be as follows:
- The operating system tries to read logical block 87.
- The controller finds that the sector is bad. It reports this finding to the operating system.
- The next time the system is rebooted, a special, command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's (spare) address by the controller.

Some controllers replace bad blocks by sector slipping.

Example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

SWAP-SPACE MANAGEMENT

- Swap-space management is another low-level task of the operating system.
- Swapping occurs when the amount of physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory.

Swap-Space Use

- The amount of swap space needed on a system can vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.
- The swap space can overestimate or underestimated. It is safer to overestimate than to underestimate the amount of swap space required. If a system runs out of swap space due to underestimation of space, it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

Swap-Space Location

- A swap space can reside in one of two places: It can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.
- External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory.
- Alternatively, swap space can be created in a separate raw partition. A separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

Swap-Space Management: An Example

- Solaris allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created.
- Linux is similar to Solaris in that swap space is only used for anonymous memory or for regions of memory shared by several processes. Linux allows one or more swap areas to be established.
- A swap area may be in either a swap file on a regular file system or a raw swap partition. Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area.

- If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page; for example, a value of 3 indicates that the swapped page is mapped to three different processes.
- The data structures for swapping on Linux systems are shown in below figure.
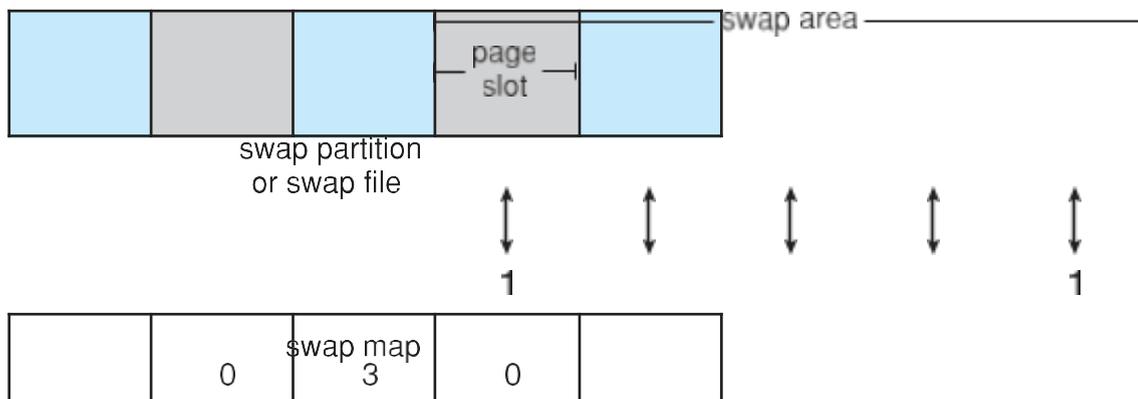
<mark>**REVIEW QUESTIONS**</mark>

1. List the different disk scheduling techniques, explain any two scheduling, consider the following disk queue requests. 98,183,37,22,14,124,65,67.
2. Explain bad-block recovery in disk?
3. Explain the following disk scheduling algorithm in brief with examples.
   a. FCFS scheduling
   b. SSTF scheduling
   c. SCAN scheduling
   d. LOOK scheduling
4.
5.

PROTECTION

Web Link: https://youtu.be/O_WbprDZMDw

GOALS OF PROTECTION

- Protection is a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. Protection ensures that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.
- Protection is required to prevent mischievous, intentional violation of an access restriction by a user.

PRINCIPLES OF PROTECTION

- A key, time-tested guiding principle for protection is the 'principle of least privilege'. It dictates that programs, users, and even systems be given just enough privileges to perform their tasks.
- An operating system provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.
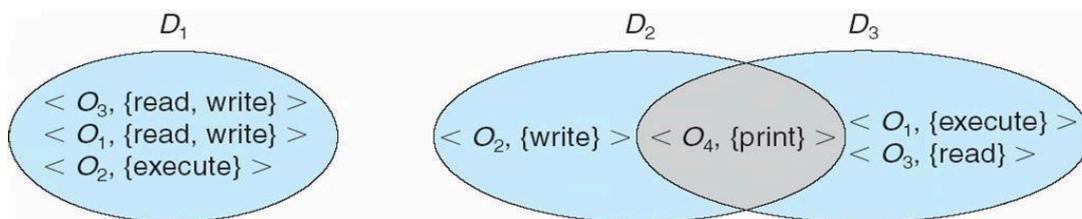
DOMAIN OF PROTECTION

- A computer system is a collection of processes and objects. Objects are both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object (resource) has a unique name that differentiates it from all other objects in the system.
- The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.
- A process should be allowed to access only those resources for which it has authorization and currently requires to complete process

Domain Structure

- A domain is a set of objects and types of access to these objects. Each domain is an ordered pair of <object-name, rights-set>.

- Example, if domain D has the access right <file F,{read,write}>, then all process executing in domain D can both read and write file F, and cannot perform any other operation on that object.
- Domains do not need to be disjoint; they may share access rights. For example, in below figure, we have three domains: D_1 D_2, and D_3. The access right $< O_4$, (print}> is shared by D_2 and D_3,it implies that a process executing in either of these two domains can print object O4.
- A domain can be realized in different ways, it can be a user, process or a procedure. ie. each user as a domain, each process as a domain or each procedure as a domain.

ACCESS MATRIX

- Our model of protection can be viewed as a matrix, called an access matrix. It is a general model of protection that provides a mechanism for protection without imposing a particular protection policy.
- The rows of the access matrix represent domains, and the columns represent objects.
- Each entry in the matrix consists of a set of access rights.
- The entry access(i,j) defines the set of operations that a process executing in domain Di can invoke on object Oj.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- In the above diagram, there are four domains and four objects—three files (F1, F2, F3) and one printer. A process executing in domain D1 can read files F1 and F3. A process executing in domain D4 has the same privileges as one executing in domain D1; but in addition, it can also write onto files F1 and F3.
- When a user creates a new object Oj, the column Oj is added to the access matrix with the appropriate initialization entries, as dictated by the creator.

The process executing in one domain and be switched to another domain. When we switch a

process from one domain to another, we are executing an operation (switch) on an object (the domain).

Domain switching from domain D_i to domain D_j is allowed if and only if the access right switch access(i,j). Thus, in the given figure, a process executing in domain D_2 can switch to domain D_3 or to domain D_4. A process in domain D_4 can switch to D_1, and one in domain D_1 can switch to domain D_2.

object \ domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.

object \ domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object \ domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The copy right allows the copying of the access right only within the column for which the right is defined. In the below figure, a process executing in domain D_2 can copy the read operation into any entry associated with file F_2. Hence, the access matrix of figure (a) can be modified to the access matrix shown in figure (b).

This scheme has two variants:

1. A right is copied from access(i,j) to access(k,j); it is then removed from access(i,j). This action is a transfer of a right, rather than a copy.
2. Propagation of the copy right- limited copy. Here, when the right R* is copied from access(i,j) to access(k,j), only the right R (not R*) is created. A process executing in domain D_k cannot further copy the right R.

We also need a mechanism to allow addition of new rights and removal of some rights. The owner right controls these operations. If access(i,j) includes the owner right, then a process executing in domain Di, can add and remove any right in any entry in column j.

For example, in below figure (a), domain D1 is the owner of F1, and thus can add and delete any valid right in column F1. Similarly, domain D2 is the owner of F2 and F3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of figure(a) can be modified to the access matrix shown in figure(b) as follows.

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

A mechanism is also needed to change the entries in a row. If access(i,j) includes the control right, then a process executing in domain D_i, can remove any access right from row j. For example, in figure, we include the control right in access(D_3, D_4). Then, a process executing in domain D_3 can modify domain D_4.

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

IMPLEMENTATION OF ACCESS MATRIX

Different methods of implementing the access matrix (which is sparse)
- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- Lock-Key Mechanism

1. Global Table

- This is the simplest implementation of access matrix.
- A set of ordered triples <domain, object, rights-set> is maintained in a file. Whenever an operation M is executed on an object O_j, within domain D_i, the table is searched for a triple <D_i, O_j, R_k>. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

Drawbacks -
The table is usually large and thus cannot be kept in main memory. Additional I/O is needed

2. Access Lists for Objects

- Each column in the access matrix can be implemented as an access list for one object. The empty entries are discarded. The resulting list for each object consists of ordered pairs <domain, rights-set>.
- It defines all domains access right for that object. When an operation M is executed on object O_j in D_i, search the access list for object O_j, look for an entry <D_i, R_K > with M ϵ R_k. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

3. Capability Lists for Domains

- A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its name or address, called a capability.
- To execute operation M on object O_j, the process executes the operation M, specifying the capability for object O_j as a parameter. Simple possession of the capability means that access is allowed.

Capabilities are distinguished from other data in one of two ways:
1. Each object has a tag to denote its type either as a capability or as accessible data.
2. The address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system.

4. A Lock-Key Mechanism

- The lock-key scheme is a compromise between access lists and capability lists.
- Each object has a list of unique bit patterns, called <u>locks</u>. Each domain has a list of unique bit patterns, called <u>keys</u>.
- A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

ACCESS CONTROL

- Each file and directory are assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned.
- Solaris 10 advances the protection available in the Sun Microsystems operating system by explicitly adding the principle of least privilege via role-based access control (RBAC). This facility revolves around privileges.
- A privilege is the right to execute a system call or to use an option within that system call.
- Privileges can be assigned to processes, limiting them to exactly the access they need to perform their work. Privileges and programs can also be assigned to roles.
- Users are assigned roles or can take roles based on passwords to the roles. In this way, a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in below figure.
- This implementation of privileges decreases the security risk associated with super users and setuid programs.
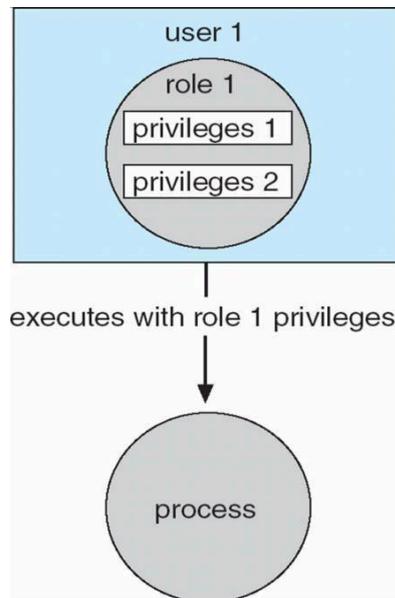
REVOCATION OF ACCESS RIGHTS

The capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

1. **Reacquisition** - Periodically, all capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
2. **Back-pointers** - A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary.
3. **Indirection** - The capabilities point indirectly to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry.
4. **Keys** - A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning the capability. A master key is associated with each object; it can be defined or replaced with the set-key operation.

When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised.

In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users.

REVIEW QUESTIONS

1. What is an access matrix? Explain the different methods of implementing access matrix.
2. Explain the various mechanism of implementing file protection.
3. Explain the access matrix model of implementing protection in operating system.
4. Describe schemes that implement revocation for capabilities of Access Rights?
5. What are the goals and principles of protection?

****