Unit – V

Unit – V
File System Interface

1. File Concept

- Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.
- The physical storage is converted into a logical storage unit by operating system. The logical storage unit is called FILE.
- A file is a collection of similar records. A record is a collection of related fields that can be treated as a unit by some application program. A field is some basic element of data. Any individual field contains a single value. A data base is collection of related data.

Student	Marks	Marks	Fail/Pas
KUMA	85	86	P
LAKSH	93	92	P

Data File

- Student name, Marks in sub1, sub2, File/Pass in fields. The collection of fields is called a RECORD.

LAKSH	93	92	P

- Collection of these records is called a data file

1.1. File Attributes

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 – The unique tag (number) identifies file within file system

Type – This information is needed for systems that support different types

Location – This information is a pointer to file location on device

Size – The current size of the file

Protection – Access-control information determines who can do reading, writing, executing

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.

1.2. 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.
- Six basic operations comprise the minimal set of required file operations.

Creating a file: Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.

Writing a file: To write a file, we have to know two things. One is name of the file name and second is the information or data to be written on the file, the system searches the entire given location for the file. If the file is found, the system must keep a write pointer to the location in the file where the next write is to take place.

Reading a file: To read a file, first of all we searches the directories for the file. If the file is found, 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.

Repositioning within a file: The directory is searched for the appropriate entry and the current-file-position pointer is repositioned to a given value. This operation is also called **file seek.**

Deleting a file: To delete a file, first of all search the directory for the named file, then released the file space and erase the directory entry.

Truncating a file: To truncate a file, remove the file contents only but, attributes are as it is.

- Several pieces of data are needed to manage open files:

Open-file table: contains information about all open files

File pointer: pointer to last read/write location, per process that has the file open

File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last process closes it

Disk location of the file: The information needed to locate the file on disk is kept in memory so that system does not have to read it from disk for each operation.

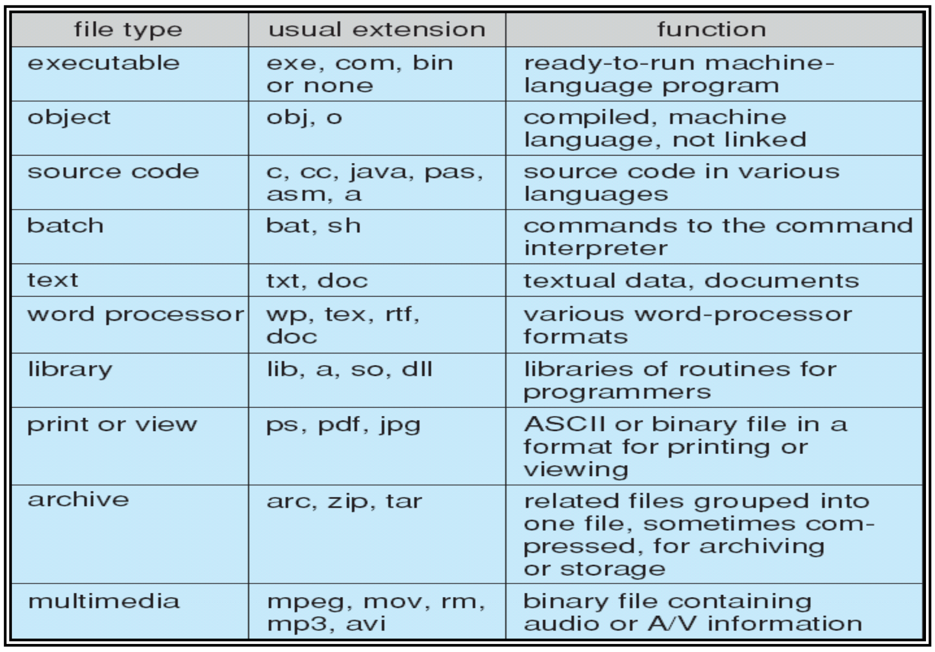
Access rights: Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

File Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

1.3. File Types

- A File name is split into two parts – a name and an extension, usually separated by a period. The file type is depending on extension of the 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 compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

1.4. File Structures

- File types also can be used to indicate the internal structure of the file. 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.
- All operating system support at least one structure that of an executable file so that the system is able to load and run programs.

1.5. Internal File Structure

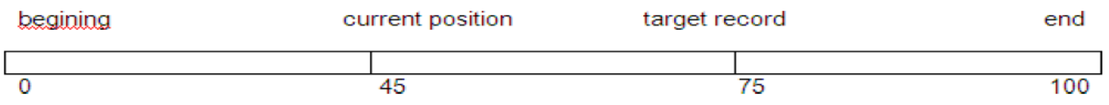
- Internally, locating an offset within a file can be complicated for the operating system. Data systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block, and all blocks are the same size.
- In UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks, say 512 bytes per block
- The logical record size, physical block size, and packing technique determines how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks.
- If each block where 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. It is called internal fragmentation. Al file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

2. 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.

2.1. Sequential Access

- Information in the file is processed in order, one record after the other. This mode of access is by far the most common for editors and compilers usually access files in this fashion.
- Read and writes make up the bulk of the operations on a file. A read operation read_next() reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation write_next() appends to the end of the file and advances to the end of the newly written material.
- Such 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.
- **Example:**
  A file consisting of 100 records, the current position of read/write head is 45th record, suppose we want to read the 75th record then, it access sequentially from 45,46,47..74,75
  So the read/write head traverse all the records between 45 to 75



2.2. Direct Access

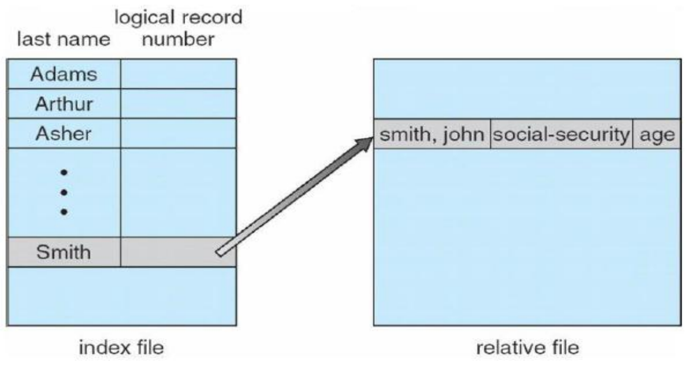
- Direct access is also called relative access. Here records can read/write randomly without any order. The direct access method is based on a disk model of a file, because disks allow random access to any file block.

Example: A disk containing of 256 blocks, the position of read/write head is at 95th block. The block is to be read or write 250th block. Then we can access the 250th block directly without any restrictions.

2.3. Other Access Methods (Indexed Sequential File Access)

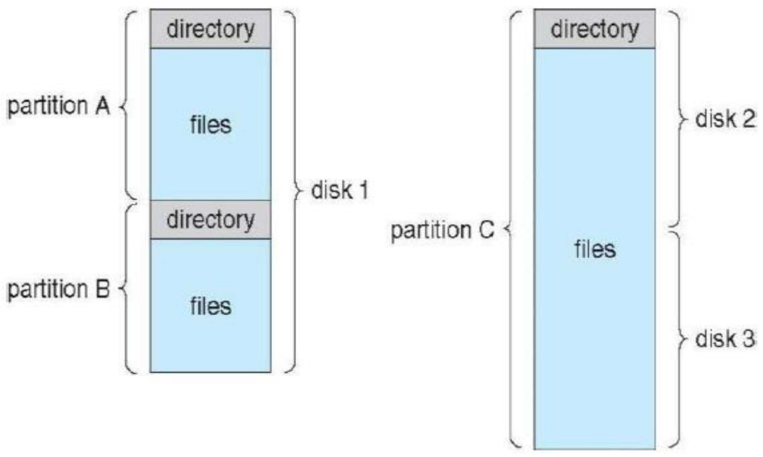
- The main disadvantage in the sequential file is, it takes more time to access a Record. Records are organized in sequence based on a key field.

Example: A File consisting of 6000 records, the master index divide the total records into 6 blocks, each block consisting of a pointer to secondary index. The secondary index divide the 10,000 records into 10 indexes. Each index consisting of a pointer to its original location. Each record is the index file consisting of 2 field. A key field and a pointer field.



3. Directory and Disk Structure

- Millions of files are stored on random-access storage devices, including hard disks, optical disks, and solid-state disks.
- To manage these files, these files are grouped and load one group into one partition.
- Each partition is called a **directory.** A directory provides a mechanism for organizing many files in the file system.



A typical file-system organization.

3.1. Storage Structure

- A general-purpose computer has multiple storage devices and those devices can be sliced up into volumes that hold file systems.
- For example, a typical Solaris system may have dozens of file systems of a dozen different types.
    - tmpfs – memory-based volatile FS for fast, temporary I/O
    - objfs – interface into kernel memory to get kernel symbols for debugging
    - ctfs – contract file system for managing daemons

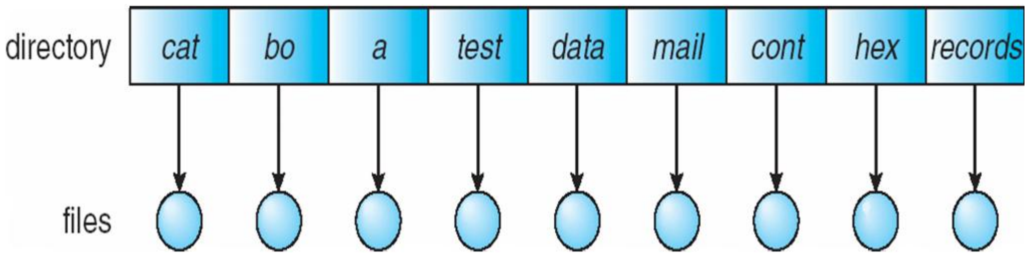
- lofs – loopback file system allows one FS to be accessed in place of another
- procfs – kernel interface to process structures
- ufs, zfs – general purpose file systems

3.2. Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries.
- Operations that are performed on a directory are:
  Search for a file: Search a directory structure for a particular file.
  Create a file: New files need to be created and added to the directory.
  Delete a file: When a file is no longer needed, we want to be able to remove it from the directory.
  List a directory: We need to be able to list the files in a directory.
  Rename a file: Whenever we need to change the name of the file, we can change the name.
  Traverse the file system: We need to access every dictionary and every file within a directory structure.

3.3. Single-Level Directory

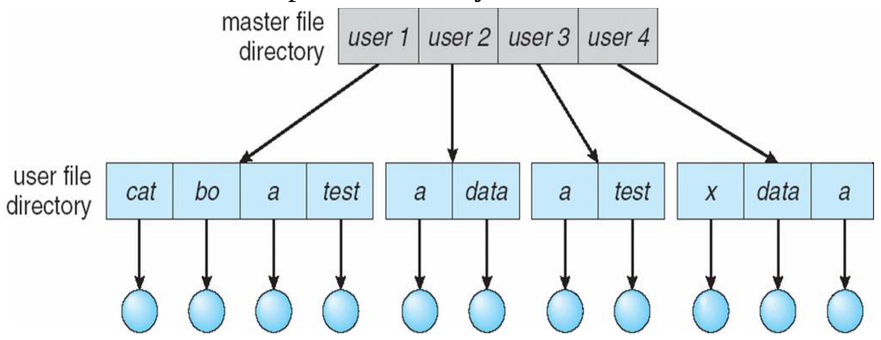
- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand.



- A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data file ***test.txt,*** then the unique-name rule is violated.
- 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.

3.4. Two-Level Directory

- A single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

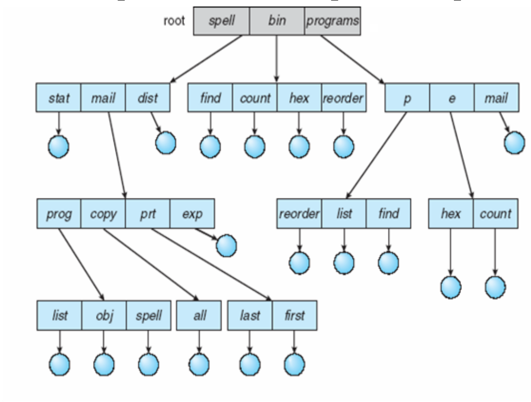


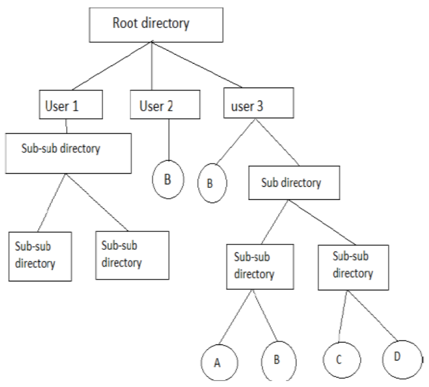
- In two-lever directory structure, each user has his own **user file directory (UFD)**. The **UFD**s have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The **MFD** is indexed by user name or account number, and each entry points to the **UFD** for that user.

- When a user refers to a particular file, only his own **UFD** is searched. Thus, 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 OS searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the OS confines its search to the local UFD; thus it cannot accidentally delete another user's file that has the same name.

3.5. Tree-Structured Directories

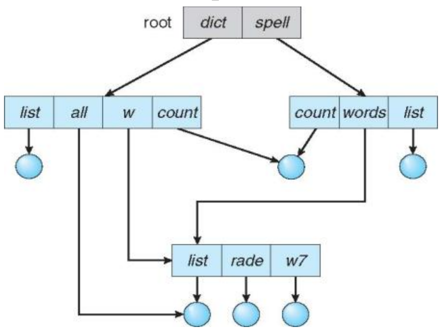
- Two-level directory eliminates name conflicts among users but it is not satisfactory for users with a large number of files. To avoid this create a sub-directory and load the same type of files into the sub-directory. So, here each can have as many directories are needed.
- The tree has a root directory, and every file in the system has a unique path name.
- In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched.
- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- Path names can be of two types: absolute and relative path. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the current directory.
- For example, in the tree-structured file system of figure below, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.





3.6. Acyclic-Graph Directories

- A tree structure prohibits the sharing of files or directories.

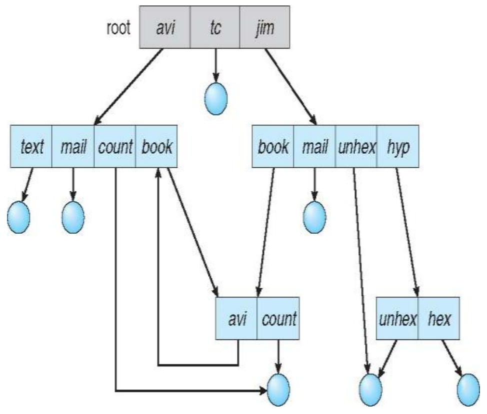


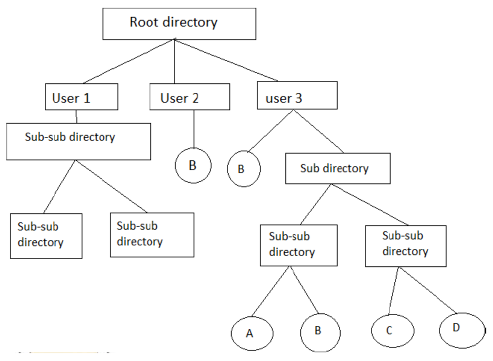
- An acyclic graph is a graph with no cycles – allow directories to share subdirectories and files. The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

- A file may now have multiple absolute paths. When shared directory/file is deleted, all pointers to the directory/files also to be removed.
- Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link.
- A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name.
- We **resolve the link** by using that path name to locate the real file. Links are easily identified by their format in the directory entry and are effectively indirect pointers.

3.7. General Graph Directory

- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results.
- It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure.
- When we add links to an existing tree structured directory, the tree structured is destroyed, resulting in a simple graph structure.

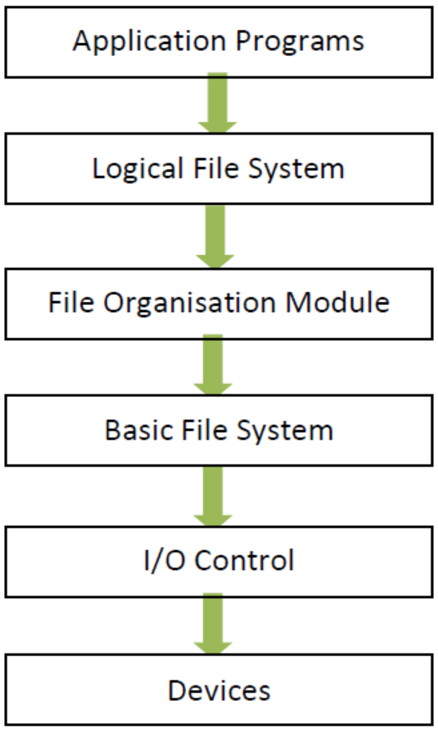




File system Implementation

1. File-system structure

- Disk provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:
  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.
- 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 4,096 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.
- 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.



I/O Control: It consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. The device driver writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.

Basic File System: It 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. (Example, drive 1, cylinder 73, track 2, sector 10).

The file-organization module: It 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 logic block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 through N. So, physical blocks containing the data usually do not match the logical numbers. A translation is needed to locate each block.

Logical File System: It manages all file system structure except the actual data (contents of file). It maintains file structure via file-control blocks. A **file-control block** (inode in UNIX file systems) contains information about the file, ownership, permissions, location of the file contents.

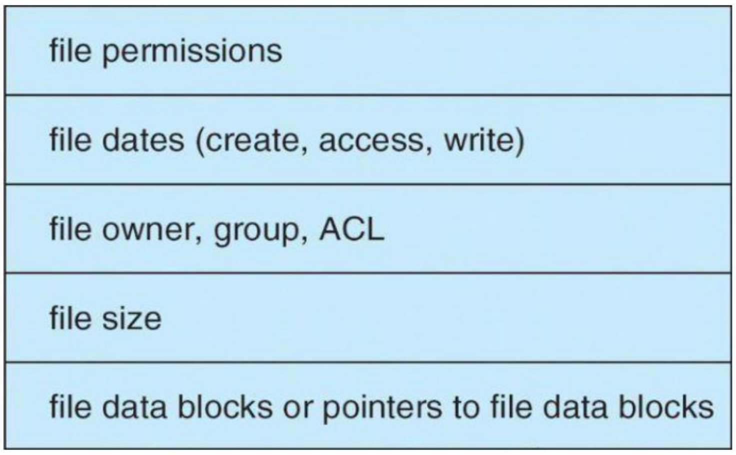
2. File-system Operations/Implementation

Operating systems implement ***open()*** and ***close()*** system calls for processes to request access to file contents.

2.1 Overview

- A **block control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an OS, this block can be empty.
- A **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 Typical File Control Block

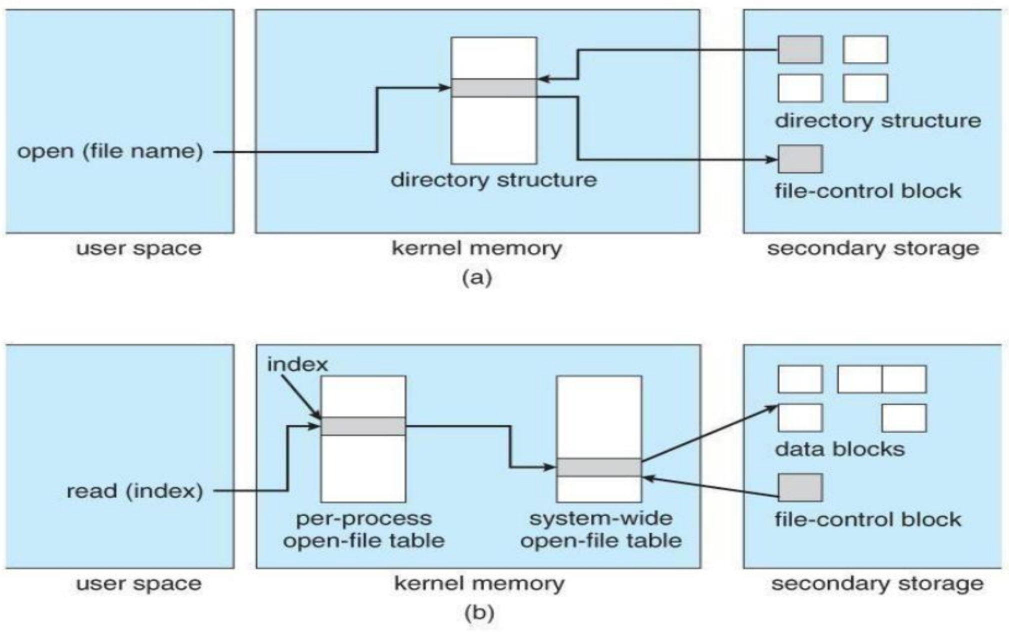


file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

- A directory structure (per file system) is used to organize the files. In per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry.
- To create a new file, an application program calls the logical file system.

2.2 Usage

- A File has been created; it can be used for I/O. The open() call passes a file name to the logical file system. The open() system call 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. If the file is not already open, the directory structure is searched for the given file name. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.
- 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 and some other fields. These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open. The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.
- When a process closes the file, the per-process table entry is removed and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.



In-memory file-system structures. (a) File Open (b) File Read

2.3 Partitioning and Mounting

- The layout of a disk can have many variations, depending on the operating system. A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks.
- 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.
- 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.
- The **root partition**, which contains the operating-system kernel and some-times other systems, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.
- As part of successful mount operation, 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.
- If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.
- Finally, the OS notes in its in-memory mount table that a file system is mounted, along with the type of the file system.
- **Microsoft Windows** – based systems mount each volume in a separate name space, denoted by a letter and a colon. To record a file system is mounted a F: , for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:
- When a process specifies the driver letter, the OS finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory.
- **On UNIX,** file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there.

3. Directory implementation

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system.

Linear List

- Linear list of file names with pointer to the data blocks.
- This method is simple to program but time-consuming to execute.
- To create a new file, we must first search the directory to be that no existing file has the same name. Then, we 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.
- The real disadvantage of a linear list of directory entries is finding a file requires a linear search.

Hash Table

- Here a linear list stores the directory entries, but a hash table data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- It can greatly decrease the directory search time. Insertion and deletion are straightforward.
- Some provisions must be made for collisions – situations in which two file names hash to the same location.
- Hash table is good only if entries are fixed size, or use chained-overflow hash table.

4. Allocation method

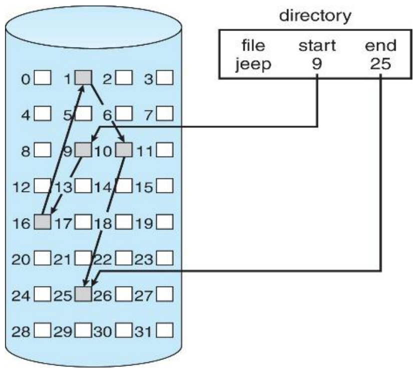
- The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk.
- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- An allocation method refers to how disk blocks are allocated for files:
- Three major methods of allocating disk space are in wide use:

1) Contiguous Allocation

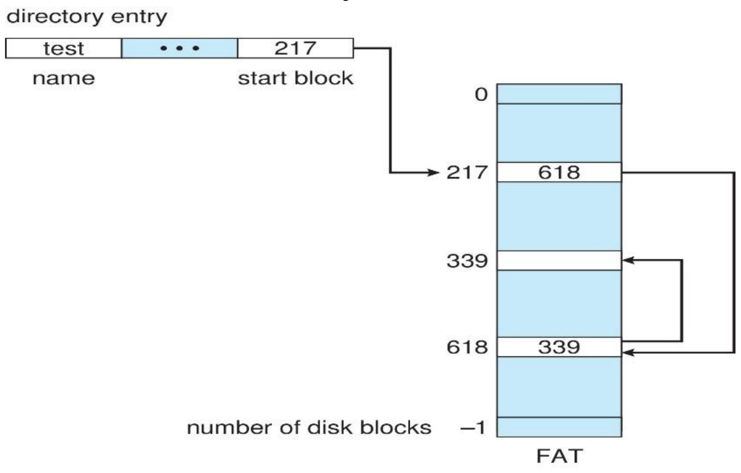
- Contiguous allocation requires that each file occupy a set of contiguous blocks of disks.
- Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file. If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, 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 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.
- Problems with contiguous allocation include
  - Finding space for a new file on disk
  - Determining how much space is needed for a file.
  - Suffers from external fragmentation.

2) Linked Allocation

- Linked allocation solves all problems of contiguous allocation.
- 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.



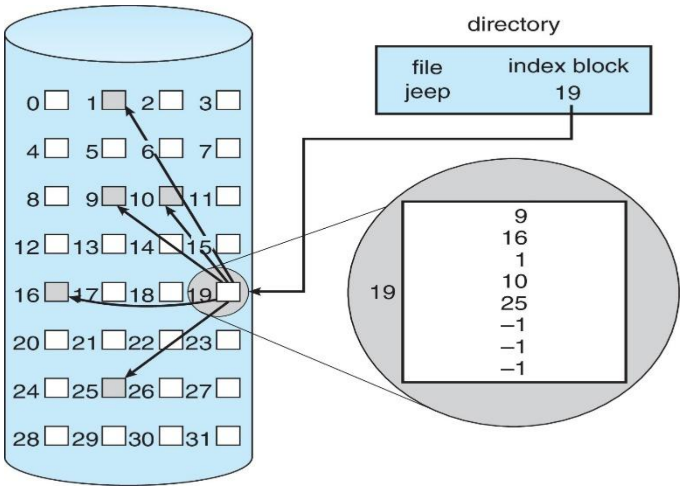
- Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address requires 4 bytes, then the user sees blocks of 508 bytes.
- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to null 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 find 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.
- Linked allocation does have disadvantages
  - It can be used efficiently only for sequential-access files.
  - The space required for the pointers. Each file requires slightly more space than it would be.
  - Reliability: The files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.
- An important variation on linked allocation is the use of a **file-allocation table (FAT)**. 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 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. This 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. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.

3) Indexed Allocation

- Linked allocation solves the problem of external fragmentation and size-declaration problems of contiguous allocation.
- However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
- Indexed allocation brings all the pointers together into one location: the **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 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.

5. Free space management

- To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks.
- To create a file, we 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

5.1 Bit Vector

- Frequently, the free-space list is implemented as a **bit map** or **bit vector.** Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25,26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be
  001111001111110001100000011100000000…

- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or ***n*** consecutive free blocks on the disk.
- The calculation of the block number is
  (number of bits per word) x (number of 0-valued words) + offset of first 1 bit.
- Bit map requires extra space
  Example:
  block size = 4KB = 212 bytes
  disk size = 240 bytes (1 terabyte)
  n = 240/212 = 228 bits (or 32MB)
  if clusters of 4 blocks -> 8MB of memory

5.2 Linked List

- Links together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on.
- For example, consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25,26, and 27 are free and the rest of the blocks are allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5 and so on.
- This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

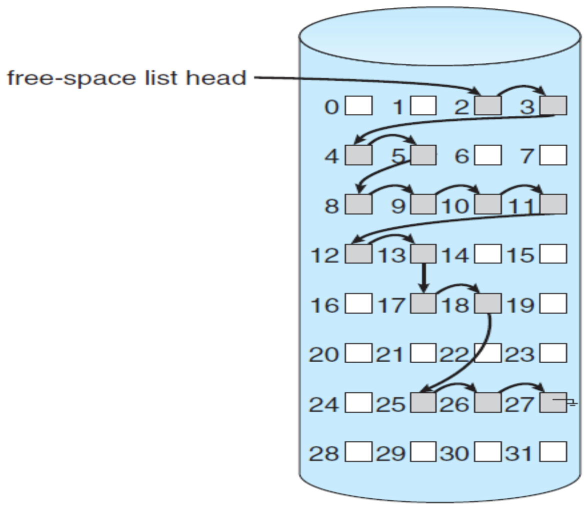


Figure 14.9 Linked free-space list on disk.

File-System Internals

1. **File-System Mounting**

- A File system must be mounted before it can be available to processes on the system.
- The mounting procedure is straightforward. The operating system is given the name of the device and the mount point – the location within the file structure where the file system is to be attached.
- Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as /***home;*** then, to access the directory structure within that file system, we could precede the directory names with ***/home***, as in ***/home/jane.***
- Next 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.
- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.

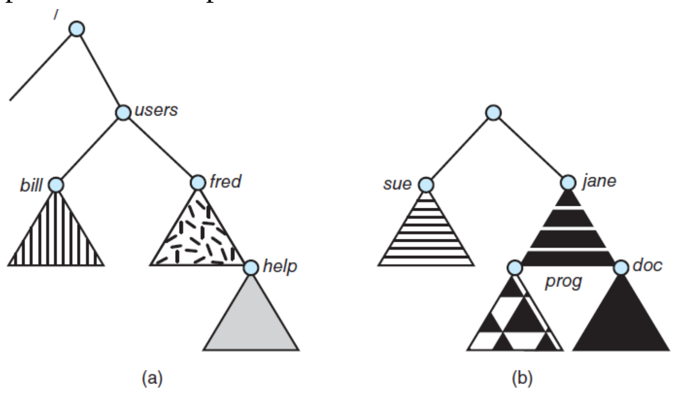


Figure 15.3 File system. (a) Existing system. (b) Unmounted volume.

- To illustrate file mounting, consider the file system depicted in Figure 15.3, where the triangles represent subtrees of directories that are of interest.
- Figure 15.3(a) shows an existing file system, while Figure 15.3(b) shows an unmounted volume residing on /device/dsk. At this point, only the files on the existing file system can be accessed.

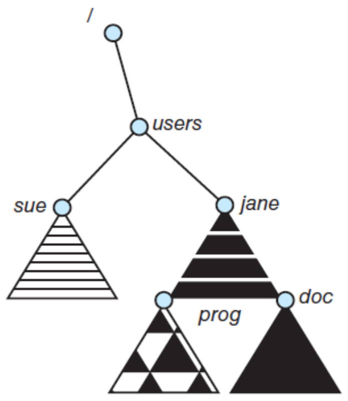


Figure 15.4 Volume mounted at /users.

- Figure 15.4 shows the effects of mounting the volume residing on /device/dsk over /users. If the volume is unmounted, the file system is restored to the situation depicted in Figure 15.3.

2. File Sharing

- Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems.

2.1 Multiple Users

- Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.
- Most systems have evolved to use the concepts of file (or directory) owner (or user) and group. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file.
- The owner and group IDs of a given file are stored with the other file attributes.
- When a user request an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.
- The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

2.2 Remote File Systems

- Through the evolution of network and file technology, remote file-sharing methods have changed.
- The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system (DFS) in which remote directories are visible from a local machine. In some ways, the third method, the World Wide Web, is a revision to the first.

The Client-Server Model

- Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is server, and the machine seeking access to the files is the client.
- Generally the server declares that a resource is available to clients and specifies exactly which resource and exactly which clients.
- A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client – server facility.

Distributed Information Systems

- To make client-server systems easier to manage, distributed information systems, also known as distributed naming services, provide unified access to the information needed for remote computing.
- The domain name system (DNS) provides host-name-to-network-address translations for the entire Internet. Other distributed information systems provide user name/password/user ID/group ID space for a distributed facility.
- Sun Microsystems introduced yellow pages and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like. Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted and identifying host by IP address.
- In the case of Microsoft’s common Internet File System (CIFS), network information is used in conjunction with user authentication to create a network login that the server uses to decide whether to allow or deny access to a requested file system.

Protection

1. Goals of Protection

- We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user.
- Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.
- A protection-oriented system provides means to distinguish between authorized and unauthorized usage.
- The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system.
- A protection system must have the flexibility to enforce a variety of policies.
- Policies for resource use may vary by application, and they many change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system.
- The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse.

2. Principles of Protection

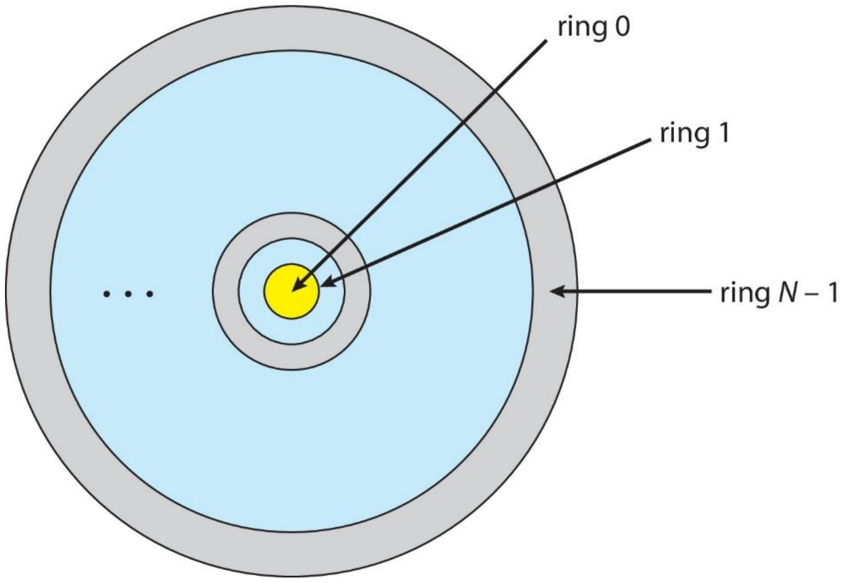
- Guiding principle – **principle of least privilege**
  - ▪ Programs, users and systems should be given just enough privileges to perform their tasks
  - ▪ Properly set permissions can limit damage if entity has a bug, gets abused
  - ▪ Can be static (during life of system, during life of process)
  - ▪ Or dynamic (changed by process as needed) – domain switching, privilege escalation
  - ▪ Compartmentalization a derivative concept regarding access to data
    - ✓ Process of protecting each individual system component through the use of specific permissions and access restrictions
- Must consider “grain” aspect
  - ▪ Rough-grained privilege management easier, simpler, but least privilege now done in large chunks
    - ✓ For example, traditional Unix processes either have abilities of the associated user, or of root
  - ▪ Fine-grained management more complex, more overhead, but more protective
    - ✓ File ACL lists, RBAC
- Domain can be user, process, procedure
- **Audit trail** – recording all protection-orientated activities, important to understanding what happened, why, and catching things that shouldn’t

3. Protection Rings

- Components ordered by amount of privilege and protected from each other
  - ▪ For example, the kernel is in one ring and user applications in another
  - ▪ This privilege separation requires hardware support
  - ▪ Gates used to transfer between levels, for example the syscall Intel instruction

- Also traps and interrupts
- **Hypervisors** introduced the need for yet another ring
- ARMv7 processors added TrustZone(TZ) ring to protect crypto functions with access via new Secure Monitor Call (SMC) instruction
  - ✓ Protecting NFC secure element and crypto keys from even the kernel

- Let D_i and D_j be any two domain rings
- If $j < I \Rightarrow D_i \subseteq D_j$



4. Domain of protection

- A Computer system is a collection of processes and objects. By ***objects*** we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores).
- The operations that are possible may depend on the object. For example, on a CPU, we can only execute. Memory segments can be read and written whereas a CD-ROM or DVD-ROM can only be read.
- A Process should be allowed to access only those resources for which is has authorization. A Process should be able to access only those resources that it currently requires to complete its task is referred to as need-to-know principle.
- Implementation can be via process operating in a protection domain
  - Specifies resources process may access
  - Each domain specifies set of objects and types of operations on them
  - Ability to execute an operation on an object is an access right

    <object-name, rights-set>

4.1 Domain Structure

- A process operates within a **protection domain**, which specifies the resources that the process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an **access right**.
- A domain is a collection of access rights, each of which is an ordered pair

    <object-name, rights-set>
- For example, if domain D has the access right <file F, {read,write}>, then a process executing in domain D can both read and write file F.

- Domain may share access rights. For example, figure below, 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, implying that a process executing in either of these two domains can print object O_4.

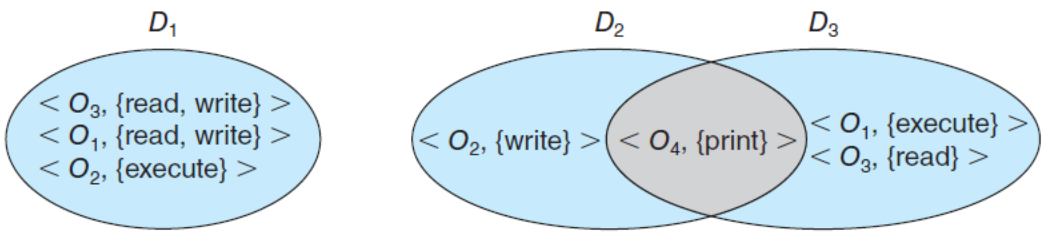
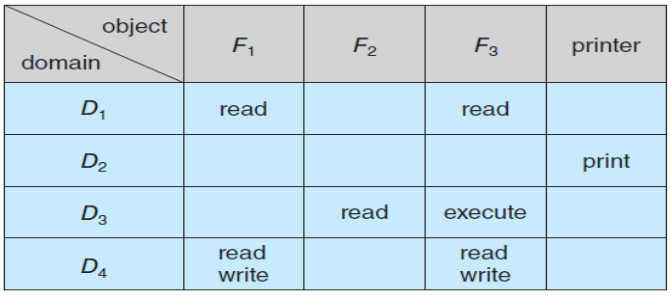


Figure 17.4 System with three protection domains.

- The association between a process and a domain may be either static, if the set of resources available to the process is fixed throughout the process's lifetime, or dynamic.
- If the association is dynamic, a mechanism is available to allow domain switching, enabling the process to switch from one domain to another.
- We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.
- A domain can be realized in a variety of ways:
  - Each ***user*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.
  - Each ***process*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
  - Each ***procedure*** may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

5. Access matrix

- Protection can be viewed abstractly as a matrix, called an **access matrix.** The rows of the access matrix represent domains, and the columns represent objects.
- The entry access(i,j) defines the set of operations that a process executing in domain D_i can invoke on object O_j.



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	

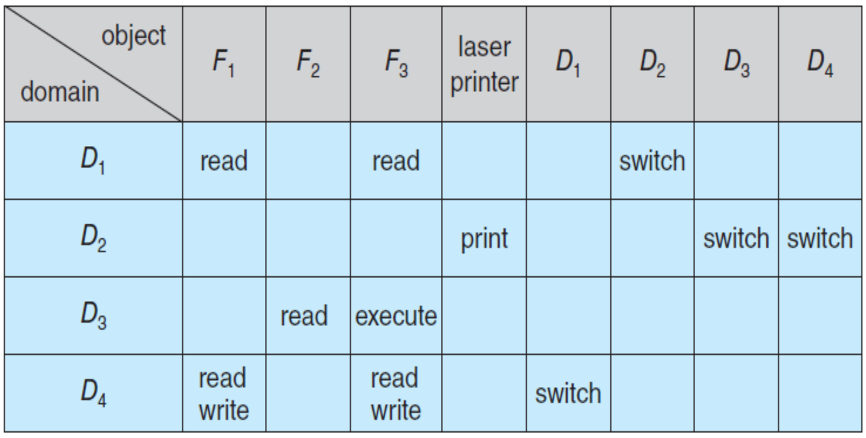
Figure 17.5 Access matrix.

- To illustrate these concepts, we consider the access matrix shown in Figure 17.5. There are four domains and four objects—three files (F_1, F_2, F_3) and one laser printer.

- A process executing in domain D_1 can read files F_1 and F_3.
- A process executing in domain D_4 has the same privileges as one executing in domain D_1; but in addition, it can also write onto files F_1 and F_3.
- The laser printer can be accessed only by a process executing in domain D_2.

Use of Access Matrix

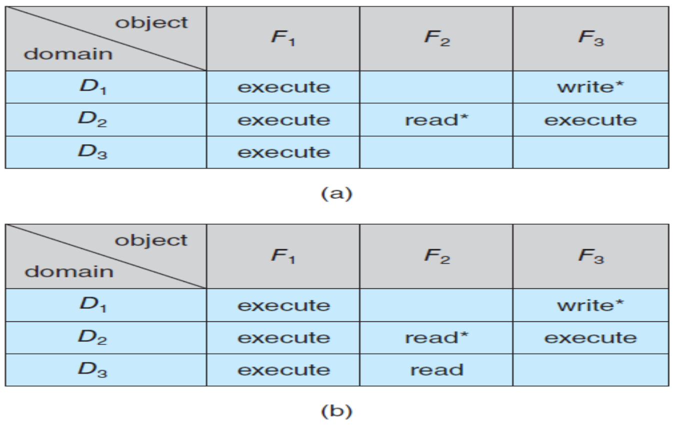
- If a process in Domain Di tries to do "op" on object Oj, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - ✓ owner of O_i
    - ✓ copy op from O_i to O_j (denoted by "*")
    - ✓ control – D_i can modify D_j access rights
    - ✓ transfer – switch from domain D_i to D_j
  - Copy and Owner applicable to an object
  - Control applicable to domain object
- Access matrix design separates mechanism from policy
  - Mechanism
    - ✓ Operating system provides access-matrix + rules
    - ✓ If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
  - Policy
    - ✓ User dictates policy
    - ✓ Who can access what object and in what mode
- But doesn't solve the general confinement problem



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			

Figure 17.6 Access matrix of Figure 17.5 with domains as objects.

- Process should be able to switch from one domain to another. Switching from domain D_i to domain D_j is allowed if and only if the access right switch ∈ access(i,j). Thus in figure above, a process executing in domain D_2 can switch to domain D_1 or to domain D_4.
- 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		

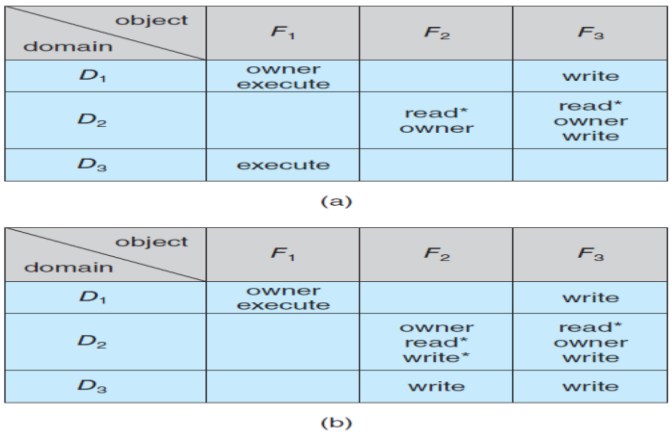
(a)

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

(b)

Figure 17.7 Access matrix with ***copy*** rights.

- 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 access right to be copied only within the column for which the right is defined.



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

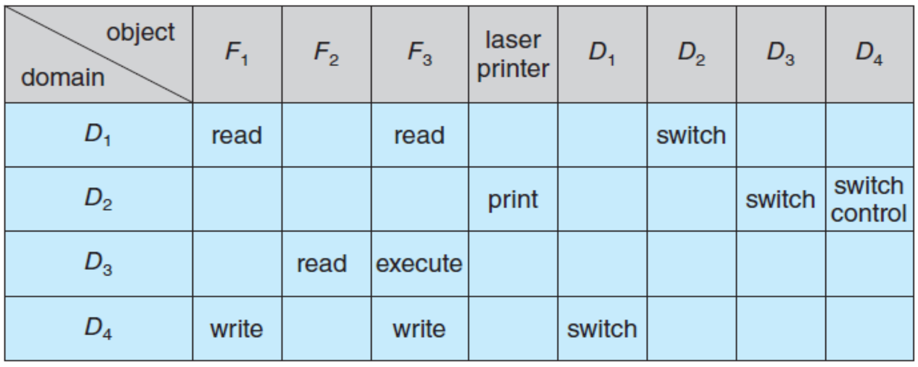
(a)

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)

Figure 17.8 Access matrix with owner rights.

- The ***owner*** right controls addition of new rights and removal of some rights. If access(i,j), includes the ***owner*** right, then a process executing in domain D_i can add and remove any right in any entry in column ***j***.



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			

Figure 17.9 Modified access matrix of Figure 17.6.

- The ***owner*** and ***copy*** rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row.
- The ***control*** right is applicable only to domain objects. If access(i,j) include the ***control*** right, then a process executing in domain D_i can remove any access right from row j.
- For example, suppose that in Figure 17.9, we include the ***control*** right in access (D_2,D_4). Then a process executing in domain D_2 could modify Domain D_4 as shown in Figure 17.9