OPERATING SYSTEMS

MODULE-5

MASS STORAGE STRUCTURE

File System: File System Interface: File concept, Access methods, Directory Structure; File system Implementation: File-system structure, File-system Operations, Directory implementation, Allocation method, Free space management; File-System Internals: File-System Mounting, Partitions and Mounting, File Sharing.
Protection: Goals of protection, Principles of protection, Protection Rings, Domain of protection, Access matrix.

THE CONCEPT OF A FILE

Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks. The operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage. Data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data.

A file has a certain defined structure, which depends on its type. A **Text File** is a sequence of characters organized into lines (and possibly pages). A **Source File** is a sequence of functions, each of which is further organized as declarations followed by executable statements. 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. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not.

When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file example.c, and another user might edit that file by specifying its name. Unless there is a sharing and synchronization method, that second copy is now independent of the first and can be changed separately.

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.

Timestamps 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 resides on the same device as the files themselves. 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. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic 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 a directory.

Opening a file

Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file open() first. If successful, the open call returns a file handle that is used as an argument in the other calls.

Writing a file

To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.

Reading a file

To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a read pointer to the location in the file where the next read is to take place, if sequential. 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. Both the read and write operations use this same pointer, saving space and reducing system complexity.

Repositioning within a file

The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as file seek.

Deleting a file

To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry.

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—except for file length. The file can then be reset to length zero, and its file space can be released.

FILE TYPES

If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the

binary- object form of a program. This attempt normally produces garbage; However, the attempt can succeed if the operating system has been told that the file is a binary-object program.

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. In this way, the user and the operating system can tell from the name alone what the type of a file is.

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, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure Common file types.

FILE STRUCTURE

A File Structure should be according to a required format that the operating system can understand. File types also can be used to indicate the internal structure of the file. A file has a certain defined structure according to its type. A text file is a sequence of characters organized into lines. A source file is a sequence of procedures and functions. An object file is a sequence of bytes organized into blocks that are understandable by the machine. 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.

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 systems provide only one access method for files. Others support many access methods, and choosing the right one for a particular application is a major design problem.

File access methods are

- **SEQUENTIAL ACCESS METHOD**
- **DIRECT ACCESS METHOD**
- **INDEXED SEQUENCIAL ACCESS METHOD**

SEQUENTIAL ACCESS METHOD

The simplest access method is 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 example, editors and compilers usually access files in this fashion. Reads 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, the new end of file.

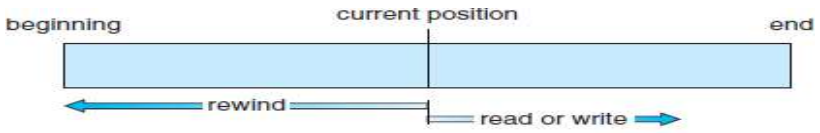


Figure Sequential-access file.

Such a file can be reset to the beginning. On some systems, a program may be able to skip forward or backward n records for some integer n, perhaps only for n = 1. Sequential access is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

DIRECT ACCESS METHOD

Another method is direct access or relative access. Here, 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. Thus, 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.

INDEXED SEQUENTIAL ACCESS METHOD

These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks.

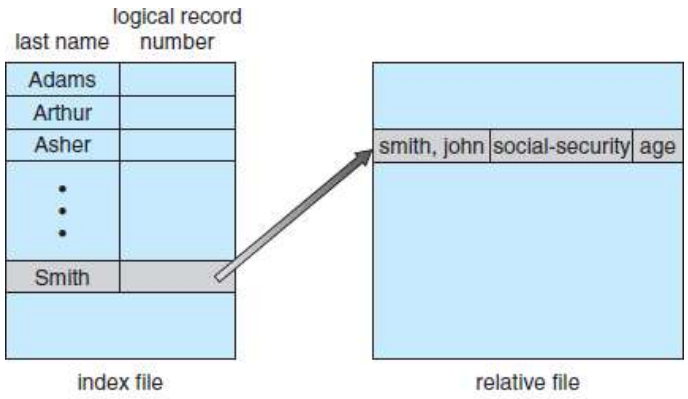


Figure Example of index and relative files.

To find a record in the file,

- we first search the index
- then use the pointer to access the file directly
- find the desired record

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items. The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. When considering a particular directory structure, different operations are to be performed on a directory. The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. When considering a particular directory structure, different operations are to be performed on a directory.

DIRECTORY STRUCTURE

The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

The operations that are to be performed on a directory:

Search for a file

We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

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. A delete leaves a hole in the directory structure and the file system may have a method to defragment the directory structure.

List a directory

We need to be 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, we must be able 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

We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape, other secondary storage, or across a network to another system or the cloud. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied the backup target and the disk space of that file released for reuse by another file.

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

- **Single Level Directory**
- **Two Level Directory**
- **Tree Structured Directory**
- **Acyclic Graph Directories**

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 as shown in the figure.

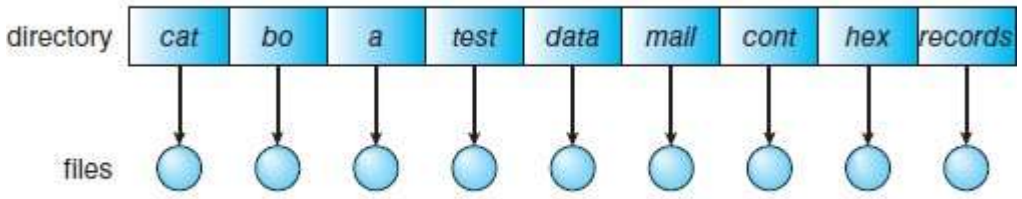


Figure Single-level directory.

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.

For example, in one programming class, 23 students called the program for their second assignment prog2.c; another 11 called it assign2.c. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Advantages:

- Since it is a single directory, so its implementation is very easy.
- If the files are smaller in size, searching will become faster.
- The operations like file creation, searching, deletion, updating are very easy in such a directory structure.

Disadvantages:

- There may chance of name collision because two files can have the same name.
- Searching will become time taking if the directory is large.
- This can not group the same type of files together.

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 the two-level directory structure, each user has his own **User File Directory** (UFD). The UFDs 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 as shown in the figure.

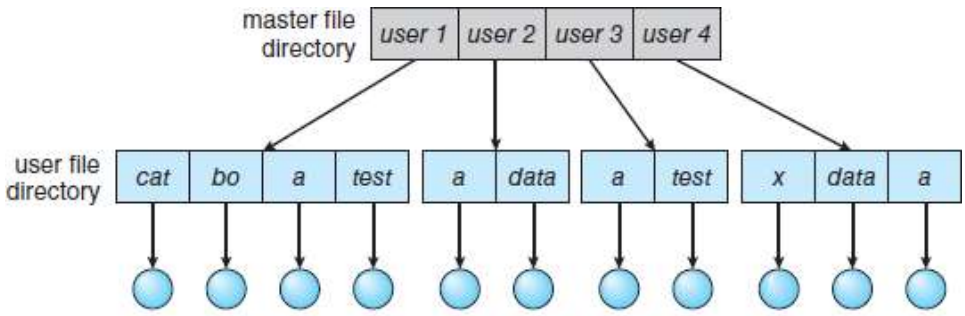


Figure Two-level directory structure.

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 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.

Advantages:

- We can give full path like /User-name/directory-name/.
- Different users can have the same directory as well as the file name.
- Searching of files becomes easier due to pathname and user-grouping.

Disadvantages:

- A user is not allowed to share files with other users.
- Still, it not very scalable, two files of the same type cannot be grouped together in the same user.

TREE STRUCTURED DIRECTORIES

To view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height as shown in the figure

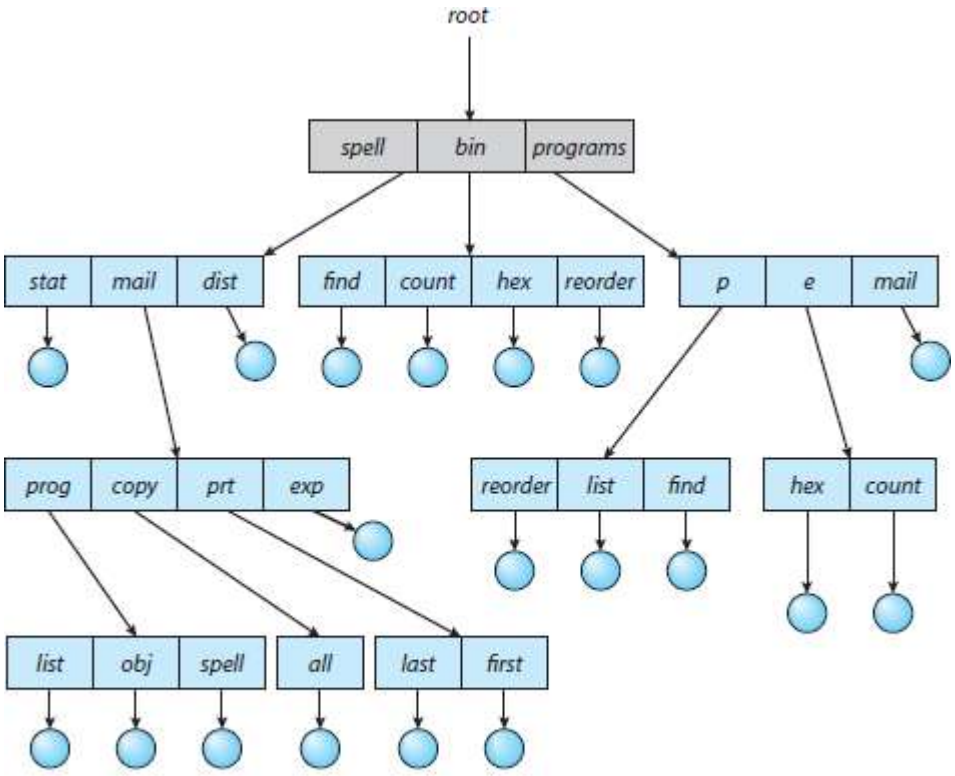


Figure Tree-structured directory structure.

This generalization allows users to create their own subdirectories and to organize their files accordingly. 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.

In tree directory structure, the users can create directories under the root directory and can also create sub-directories under this structure. As the user is free to create many sub-directories, it can create different sub-directories for different file types.

Here, the files are accessed by their location using the path. There are two types of paths to locate the file in this directory structure

Absolute Path

The path for the desired file is described by considering the root directory as the base directory.

Relative Path

Either the user's directory is considered as the base directory or the desired file directory is considered as the base directory.

Advantages:

- We can give full path like /User-name/directory-name/.
- Different users can have the same directory as well as the file name.
- Searching of files becomes easier due to pathname and user-grouping.

Disadvantages:

- A user is not allowed to share files with other users.
- Still, it not very scalable, two files of the same type cannot be grouped together in the same user.

ACYCLIC GRAPH DIRECTORIES

Consider two programmers who are working on a joint project. The files associatedwith that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be ***shared.*** A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **Acyclic Graph** that is, a graph with no cycles 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.

With the help of aliases, and links, we can create this type of directory graph. We may also have a different path for the same file.

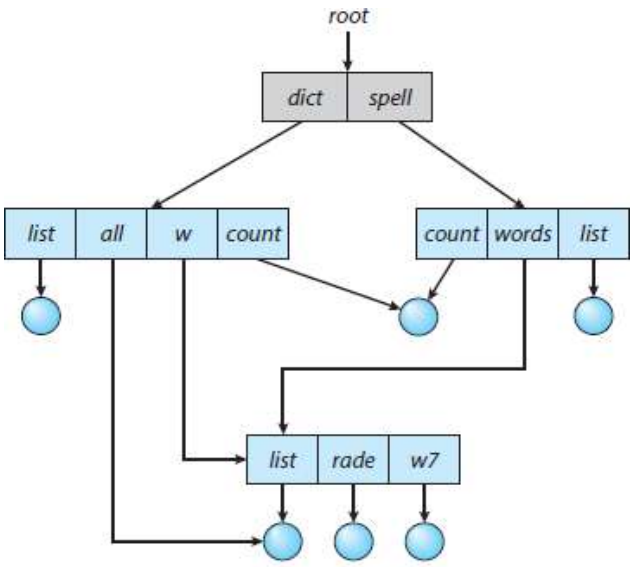


Figure Acyclic-graph directory structure.

Links may be of two kinds, which are hard link (physical) and symbolic (logical).
If we delete the files in acyclic graph structures, then

1. In the hard link (physical) case, we can remove the actual files only if all the references to the file are deleted.
2. In the symbolic link (logical) case, we just delete the file, and there is only a dangling point that is left.

Advantages:

- We can share files.
- Searching is easy due to different-different paths.

Disadvantages:

- We share the files via linking, in case deleting it may create the problem,
- If the link is a soft link then after deleting the file we left with a dangling pointer.
- In the case of a hard link, to delete a file we have to delete all the references associated with it.

FILE-SYSTEM STRUCTURE

Disks 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 block.
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 the drive moving the read–write heads and waiting for the media to rotate.

File systems provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. 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. 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.

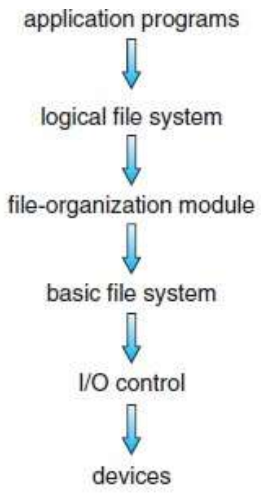


Figure Layered file system.

- The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- The **basic file** needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device. It issues commands to the drive based on logical block addresses. It is also concerned with I/O request scheduling.
- The **file-organization module** knows about files and their logical blocks. Each file's logical blocks are numbered from 0 (or 1) through *N*. 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.
- Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data. 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. A **file control block**

  (**FCB**) contains information about the file, including ownership, permissions, and location of the file contents.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules.

Unfortunately, layering can introduce more operating-system overhead, which may result in decreased performance. The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

FILE SYSTEM OPERATIONS

Operating systems implement open() and close() systems calls for processes to request access to file contents. We delve into the structures and operations used to implement filesystem operations. Several on-storage and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply.

On storage, 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. Many of these structures are

- A **boot control block** can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume.
- A **volume control block** contains volume details, such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.
- A **directory structure** 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 pointers to the appropriate entries in the system-wide open-file table, as well as other information, for all files the process has open.
- Buffers hold file-system blocks when they are being read from or written to a file system.

To create a new file, a process calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the file system. Atypical FCB is shown in figure

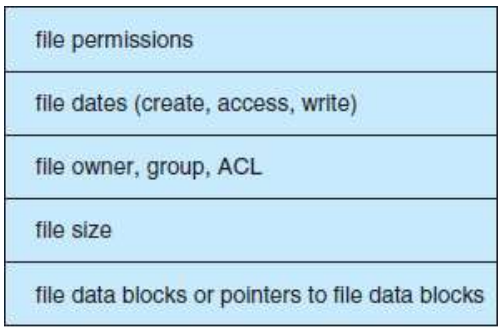


Figure A typical file-control block.

DIRECTORY IMPLEMENTATION

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

LINEAR LIST

The simplest method of implementing a directory is to use a linear list of file names with pointers 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 sure 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. To reuse the directory entry, we can do one of several things. We can mark the entry as or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly re-read the information from secondary storage.

A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

HASH TABLE

Another data structure used for a file directory is a hash table. Here, a linear list stores the directory entries, but a hash 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. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63 (for instance, by using the remainder of a division by 64). If we later try to create a 65th file, we must enlarge the directory hash table say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

ALLOCATION METHODS

The direct-access nature of secondary storage gives us flexibility in the implementation of files. In almost every case, many files are stored on the same device. The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly.

Three major methods of allocating secondary storage space are in wide use

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

CONTIGUOUS ALLOCATION

Contiguous allocation requires that each file occupy a set of contiguous blocks on the device. Device addresses define a linear ordering on the device. With this ordering, assuming that only one job is accessing the device, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed, the head need only move from one track to the next. Thus, for HDDs, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is

finally needed.

Contiguous allocation of a file is defined by the address of the first block and length 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 as shown in the figure. Contiguous allocation is easy to implement but has limitations, and is therefore not used in modern file systems.

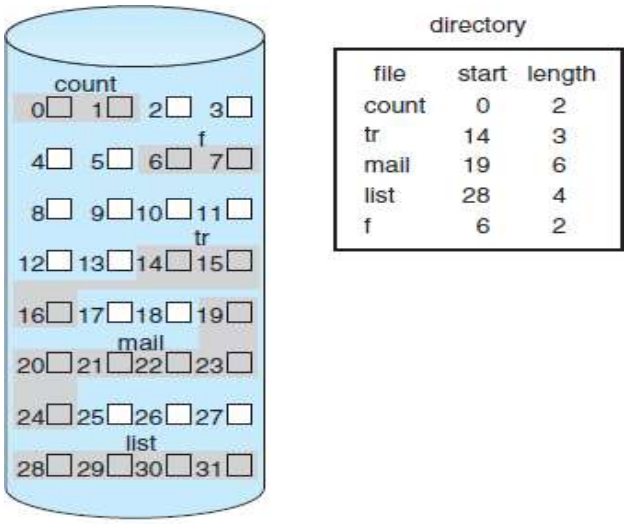


Figure Contiguous allocation of disk space.

Advantages:

1. In the contiguous allocation, sequential and direct access both are supported.
2. For the direct access, the starting address of the k^{th} block is given and further blocks are obtained by b+K,
3. This is very fast and the number of seeks is minimal in the contiguous allocation method.

Disadvantages:

1. Contiguous allocation method suffers internal as well as external fragmentation.
2. In terms of memory utilization, this method is inefficient.
3. It is difficult to increase the file size because it depends on the availability of contiguous memory.

LINKED ALLOCATION

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of storage blocks. The blocks may be scattered

anywhere on the device. 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 as shown in the figure. 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 block address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

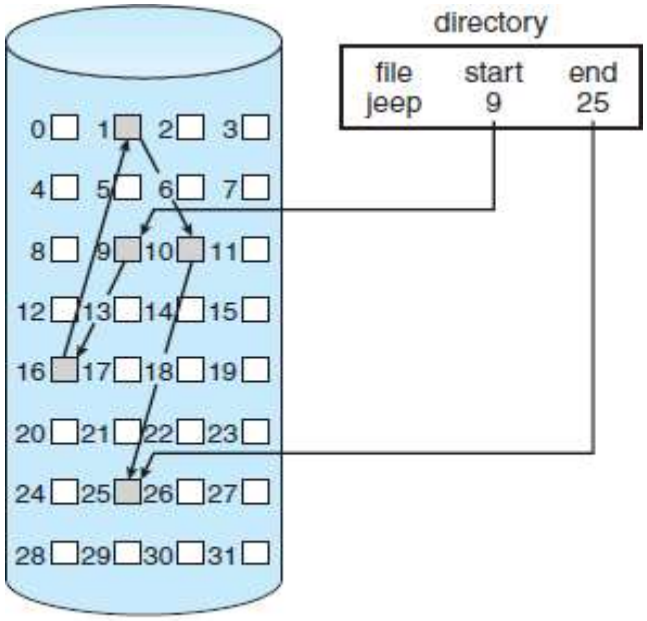


Figure Linked allocation of disk space.

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 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. 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. Consequently, it is never necessary to compact disk space.

Advantages:

1. In terms of the file size, this scheme is very flexible.

2. We can easily increase or decrease the file size and system does not worry about the contiguous chunks of memory.
3. This method is free from external fragmentation and this makes it better in terms of memory utilization.

Disadvantages:

1. In this scheme, there is large number of seeks because the file blocks are randomly distributed on disk.
2. Linked allocation is comparatively slower than contiguous allocation.
3. Random or direct access is not supported by this scheme we cannot access the blocks directly.
4. The pointer is extra overhead on the system due to the linked list.

INDEXED ALLOCATION

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However 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** solves this problem by bringing all the pointers together into one location, the **index block**.

Each file has its own index block, which is an array of storage-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block as shown in the figure. To find and read the i^{th} block, we use the pointer in the ith index-block entry. This scheme is similar to the paging scheme.

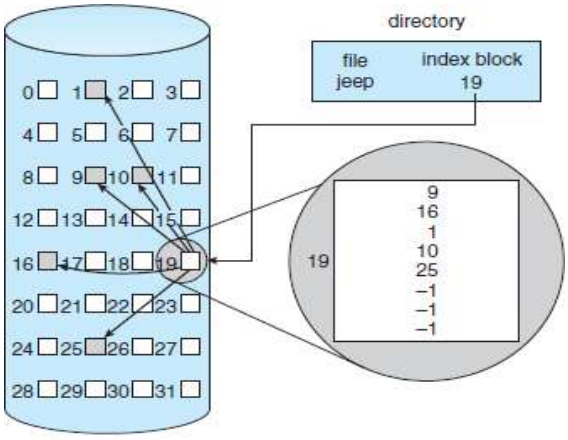


Figure Indexed allocation of disk space.

When the file is created, all pointers in the index block are set to null. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the storage device can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null. This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. 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:

Linked scheme

An index block is normally one storage 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 null (for a small file) or is a pointer to another index block (for a large file).

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. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

Combined scheme

Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. 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**.

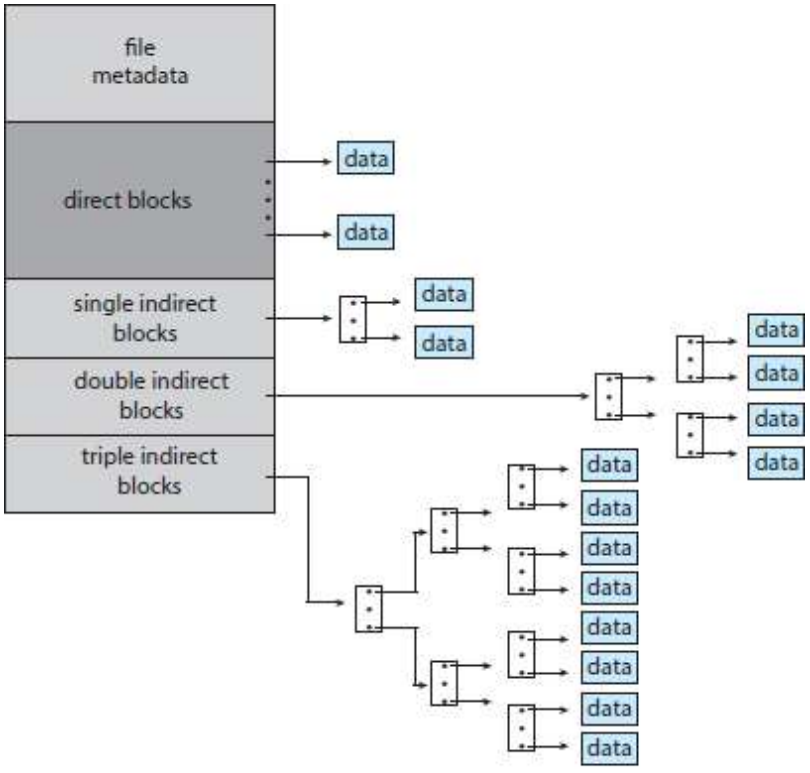


Figure The UNIX inode.

Advantages:

1. This scheme supports random access of the file.
2. This scheme provides fast access to the file blocks.
3. This scheme is free from the problem of external fragmentation.

Disadvantages:

1. The pointer head is relatively greater than the linked allocation of the file.
2. Indexed allocation suffers from the wasted space.
3. For the large size file, it is very difficult for single index block to hold all the pointers.
4. For very small files say files that expend only 2-3 blocks the indexed allocation would keep on the entire block for the pointers which is insufficient in terms of memory utilization.

FREE SPACE MANAGEMENT

Since storage space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a **free-space list.** The free- space list **records all free device blocks** those not allocated to some file or directory.

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 space is added to the free-space list. The free-space list, despite its name, is not necessarily implemented as a list.

It can be implemented as

- **Bitmap or Bit Vector**
- **Linked List**
- **Grouping**
- **Counting**

BITMAP OR BIT VECTOR

Frequently, the free-space list is implemented as a **bitmap 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 bitmap would be

00111100111111000110000011100000 ...

One technique for finding the first free block on a system that uses a bit vector to allocate space is

- To sequentially check each word in the bitmap to see whether that value is not 0.
- Since a 0-valued word contains only 0 bits and represents a set of allocated blocks.
- The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.

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.

Disadvantages are

- This technique requires a special hardware support to find the first 1 in a word it is not 0.
- This technique is not useful for the larger disks.

LINKED LIST

Another approach to free-space management is to **link together all the free blocks.** Then **keeping a pointer to the first free block** in a special location in the file system and caching it in memory. This first block contains a pointer to the next free block, and so on.

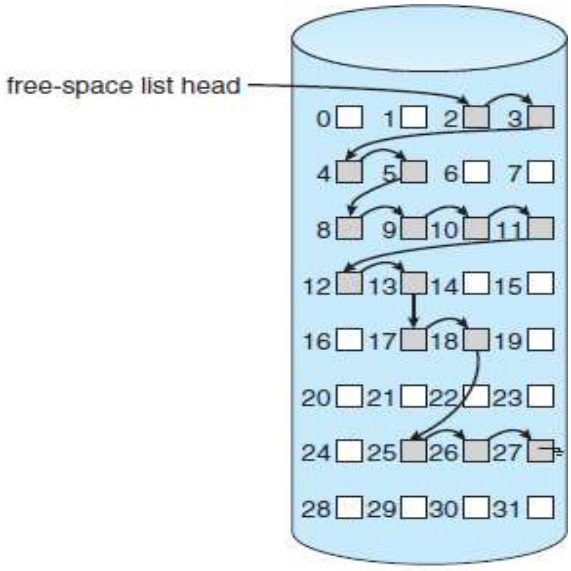


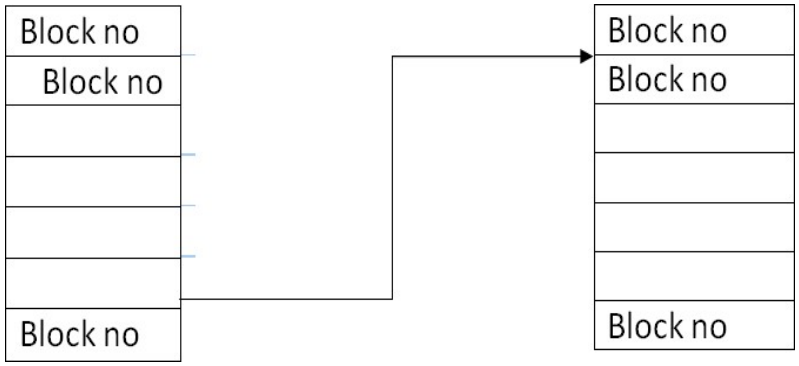
Figure Linked free-space list on disk.

For example blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were 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, which would point to block 8, and so on.

This scheme is not efficient, to traverse the list, we must read each block, which requires substantial I/O time on HDDs. However, traversing the free list is not a frequent action. Usually the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

GROUPING

A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first n−1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.



COUNTING

Another approach takes advantage of the fact that, generally, **several contiguous blocks** may be allocated or freed simultaneously. Particularly, when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free block addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a **device address and a count.**

FILE SYSTEM MOUNTING

Just as a file must be opened before it can be used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple file-system-containing volumes, which must be mounted to make them available within the file system name space.

The mount 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. Some operating systems require that a file-system type be provided, while others inspect the structures of the device and determine the type of file system.

Typically, a mount point is an empty directory. 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. This scheme enables the operating system to traverse its directory

structure, switching among file systems, and even file systems of varying types, as appropriate.

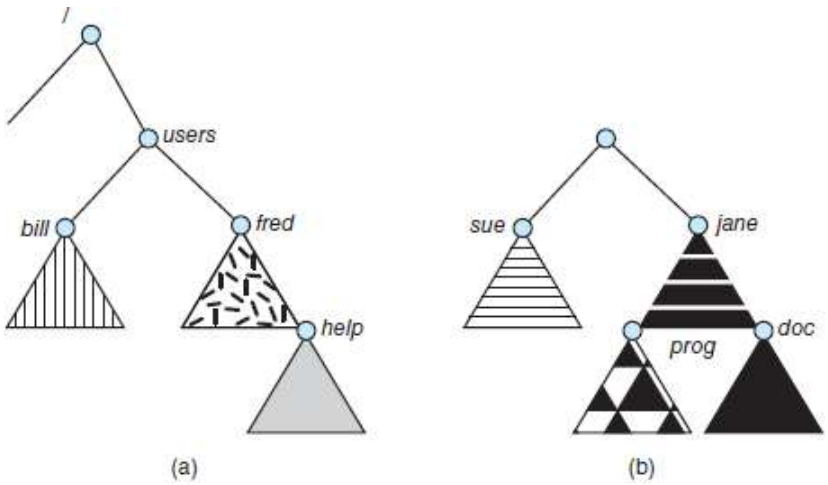


Figure 3 File system. (a) Existing system. (b) Unmounted volume.

To illustrate file mounting, consider the file system depicted in Figure 3, where the triangles represent subtrees of directories that are of interest. Figure 3(a) shows an existing file system, while Figure 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 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 3.

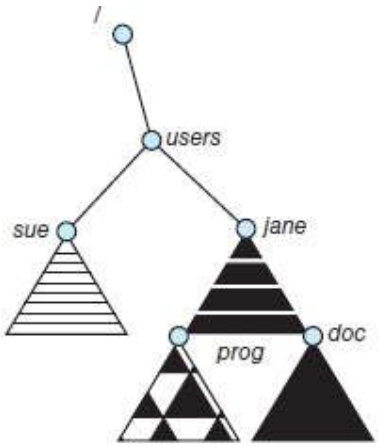


Figure 4 Volume mounted at /users.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files; or it may make the mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory.

Partitions and Mounting

Partitions and mounting are fundamental concepts in the design and operation of file systems in an operating system (OS). They help organize storage devices, enable the use of multiple file systems, and provide users with access to storage resources.

1. Partitions in File Systems

What is a Partition?

A partition is a logical division of a storage device (e.g., hard disk, SSD, or USB drive) into separate, independent sections. Each partition behaves as if it were a distinct disk and can have its own file system.

Why Partition a Disk?

- **Organization**: Separate different types of data (e.g., system files, user data, backups).
- **Multiple Operating Systems**: Allow multiple operating systems to coexist on a single disk.
- **File System Choice**: Different partitions can use different file systems (e.g., NTFS for Windows, ext4 for Linux).
- **Data Protection**: Reduces the risk of data corruption affecting all stored data.
- **Boot Management**: Create dedicated boot partitions for better boot process control.

Types of Partitions

1. **Primary Partitions**:
    - Limited to a maximum of 4 per disk in MBR (Master Boot Record) partitioning.
    - Can directly host a file system or an operating system.
2. **Extended Partitions**:
    - Used to overcome the 4-partition limit in MBR.
    - Acts as a container for **logical partitions**.
3. **Logical Partitions**:
    - Exist within an extended partition.
    - Can host file systems like primary partitions.
4. **GPT Partitions**:
    - Modern partitioning scheme replacing MBR, supporting up to 128 partitions per disk and larger disk sizes.

Partition Table

The **partition table** stores metadata about partitions (e.g., start and end locations, type, and size). It exists in formats like:

- **MBR (Master Boot Record)**: Legacy format with a 4-partition limit.
- **GPT (GUID Partition Table)**: Modern format, part of UEFI, supports larger disks and more partitions.

2. Mounting in File Systems

What is Mounting?

Mounting is the process of making a file system accessible to the operating system by attaching it to a directory (mount point) in the existing directory hierarchy.

How Mounting Works

1. **File System Detection**:
   - The OS reads the file system metadata from the partition.
2. **Association with a Mount Point**:
   - The file system is attached to a specific directory (e.g., /mnt/data in Linux or D:\ in Windows).
3. **Integration into File Hierarchy**:
   - After mounting, the contents of the file system become accessible through the mount point.

Mount Points

- A **mount point** is a directory in the file system where the partition or device is attached.
- Example:
  - In Linux: /mnt/usb or /media/user/external_drive.
  - In Windows: Drive letters like C:\ or D:\.

Unmounting

- Unmounting detaches the file system from the mount point, ensuring that no further access is made to the storage device.
- Command examples:
  - Linux: umount /mnt/data
  - Windows: Use the disk management tool or Eject.

3. Partition and Mounting Workflow

1. **Partitioning**:
   - Tools like fdisk, parted (Linux), or Disk Management (Windows) are used to divide the disk into partitions.
   - Each partition can then be formatted with a specific file system (e.g., ext4, NTFS).
2. **Formatting**:
   - After partitioning, each partition must be formatted with a file system to store data.

3. **Mounting**:
    - Linux: Use mount command or edit /etc/fstab for automatic mounting at boot.
    - Windows: Assign a drive letter or mount the partition as a folder.

4. Key Differences Between Partitions and Mounting

Aspect	Partitions	Mounting
Purpose	Divides a disk into logical sections.	Makes a file system accessible to the OS.
Level of Operation	Disk-level operation.	File system-level operation.
Tools	fdisk, parted, Disk Management.	mount, /etc/fstab, drive letters.
Outcome	Logical division of storage.	Integration of storage into the directory hierarchy.

Summary

- **Partitions** are logical divisions of a disk used to organize data, support multiple operating systems, and manage file systems.
- **Mounting** is the process of attaching a file system to a directory, making it accessible to users and applications.

Understanding these concepts is essential for effective storage management in any operating system.

FILE SHARING

The ability to share files is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

MULTIPLE USERS

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. 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.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, 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.

For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. 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 a given file (or directory) are stored with the other file attributes. When a user requests 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. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Many systems have multiple local file systems, including volumes of a single disk or multiple volumes on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted. But consider an external disk that can be moved between systems. What if the IDs on the systems are different? Care must be taken to be sure that IDs match between systems when devices move between them or that file ownership is reset when such a move occurs. For example, we can create a new user ID and set all files on the portable disk to that ID, to be sure no files are accidentally accessible to existing users.

GOALS OF PROTECTION

Protection involves **controlling the access of processes and users** to the resources defined by a computer system. To provide this protection, we can use various mechanisms to ensure that only processes that have gained **proper authorization** from the OS can operate on the **files, memory segments, CPU, networking, and other resources of a system.** These mechanisms must provide a means for specifying the **controls to be imposed**, together with a means of **enforcement.**

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.** Of

more general importance, however, is the need to ensure that each process in a system uses system resources only in ways **consistent with stated policies.** This requirement is an absolute one for a reliable system.

An **unprotected resource** cannot defend against use or misuse by an **unauthorized or incompetent user.** 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. Still others are **defined by individual users** to protect resources they "own."

A protection system, then, must have the **flexibility** to **enforce a variety of policies.** Policies for resource use **may vary by application**, and they may 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**.

Note that **mechanisms** are distinct from **policies. Mechanisms** determine **how** something will be done. **Policies** decide **what** will be done. The separation of policy and mechanism is important for **flexibility. Policies are likely to change** from place to place or time to time. In the worst case, every **change in policy** would require a change in the **underlying mechanism**.

PRINCIPLES OF PROTECTION

Frequently, a **guiding principle** can be used **throughout a project**, such as the design of an operating system. Following this principle **simplifies design decisions** and **keeps the system consistent** and **easy to understand.** A key, time tested guiding principle for protection is the **principle of least privilege.** This principle dictates that programs, users, and even systems **be given just enough privileges** to perform their tasks.

Consider one of the **tenets of UNIX** is that **a user should not run as root.** In UNIX, only the **root user** can execute **privileged commands.** Most users innately respect that, **fearing an accidental delete operation** for which there is no corresponding undelete. Because **root is virtually omnipotent**, the potential for human error when a user acts as root is grave, and its consequences far reaching.

Now consider that rather than human error, **damage may result from malicious**

attack. A virus launched by an accidental click on an attachment is one example. Another is a **buffer overflow** or other **code-injection attack** that is successfully carried out **against a root- privileged process** (or, in Windows, a process with administrator privileges). Either case could prove **catastrophic** for the system.

Observing the **principle of least privilege** would give the system **a chance to mitigate the attack.** If malicious code **cannot obtain root privileges**, there is a chance that adequately defined **permissions may block all, or at least some,** of the damaging operations. In this sense, **permissions can act like an immune system** at the operating-system level.

Another important principle, often seen as a derivative of the principle of least privilege, is **compartmentalization. Compartmentalization** is the process of **protecting each individual system component** through the use of specific **permissions and access restrictions**. Then, if a component is subverted, **another line of defense will "kick in"** and keep the attacker from compromising the system any further. Compartmentalization is implemented in many forms, from **network demilitarized zones (DMZs)** through virtualization.

PROTECTING RINGS

The operating system functions on different layers and each of them has its privileges. These privileges are mentioned using a protection ring that is used for sharing resources and the hardware systems, that manage the resources stored in the computer system like CPU processing time and memory access time. Protection rings are arranged in a hierarchical order from most trusted to least trusted privilege. The central ring at the **kernel** level can access all resources and has the highest privilege whereas the subsequent layers have a lesser level of access permissions. This mechanism is hardware imposed by the architecture of the CPU at different access modes. Processor with x86 uses four levels of rings implemented utilizing ring 0 to ring 3, where ring 0 has the highest privilege.

Importance of Protection Ring

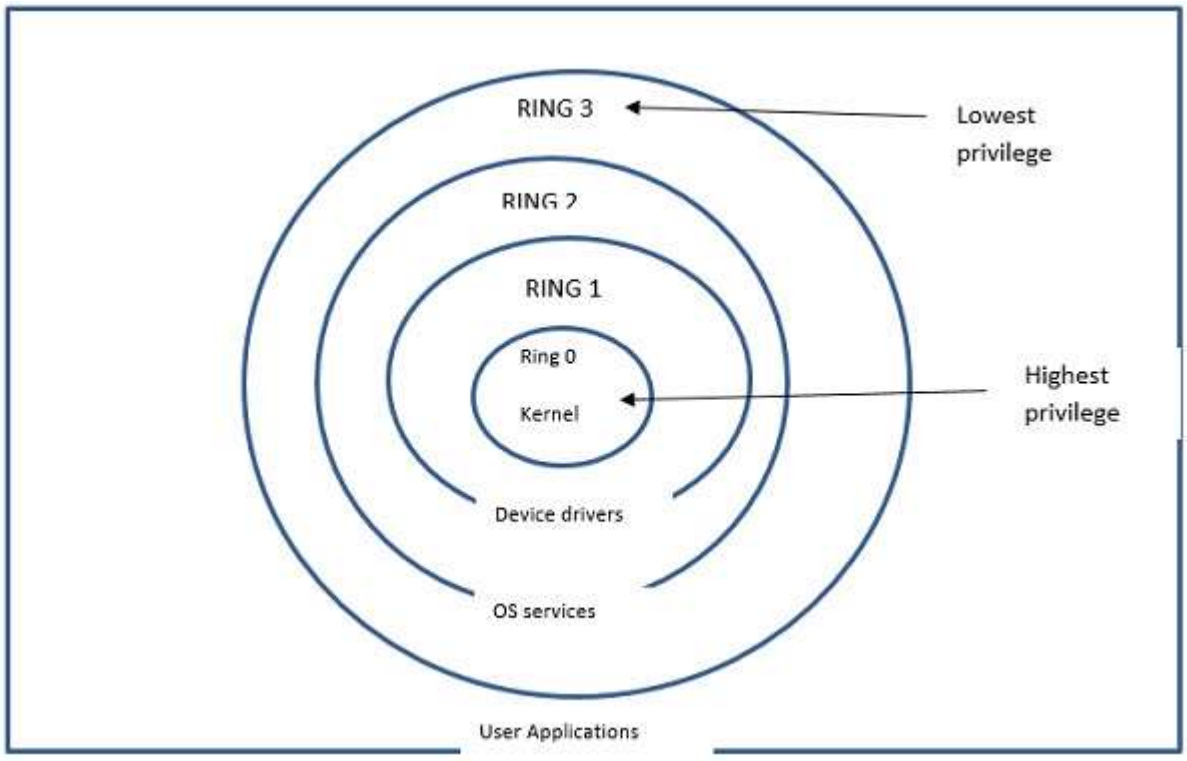
This method of providing a layered protection model offers advantages that are listed below

- It provides computer security when a process needs a set of instructions with several **CPU** resources then the process initiates a request to the **Operating system**. After receiving the request, the Operating system may decide whether to grant access privileges or not. This activity prevents the system from malicious attacks or behavior from other processes or external sources.
- System crashes can be reduced by providing fault tolerance to the applications that have direct access to the kernel space with ring 0 protection.

Levels in Protection Ring

As said earlier, the x86 instruction set uses four privilege levels from ring 0 to ring 3. Ring 0 has access to the kernel part which is called the core of the operating system. The process that runs in kernel mode can access all resources of the system. It can access the physical functions of hardware in CPU and motherboard chipsets. Computer architects have limited the interactions to this layer as it contains all the functions of system processes.

- Ring 1 and 2 have distinct privileges supported than Ring 3. Ring 1 is used to interact with hardware that would execute commands for streaming of video using a camera to the display interface. Commands that are needed for storing, loading, and saving are done in Ring 2.
- Ring 3 has the access right of the user's application and it offers the least privilege. When any resources are needed for the process then, it should request to ring 0 which is near to kernel for accessing the needed application, and all the information's passed to the lower levels.



Protection Ring for x86 processor

It is not necessary for the system needs to use all four levels of protection rings. Windows, Mac OS, Android, and UNIX operating systems use a paging mechanism to define the privilege mode as a supervisor or user mode. Ring 0 and 3 are the needed ones and other levels of Ring 1 and 2 are optional.

Interaction in Privilege Rings

Before knowing how the levels of the ring interact with the processes, one should be familiar with two types of modes.

- Supervisor mode can be modified by the code or process running at the system software level. System-level process or threads has this supervisor flag set whereas other user-level applications cannot do it. This mode provides instructions to execute the processes and it can modify the registers functions or disabling interrupt signals. Operating system with microkernel usually runs in supervisor mode.
- Hypervisor mode is supported by a recent CPU with x86 configuration to provide control to Hardware space in Ring 0. Intel VT-x and AMD-v create a new Ring 1 which is used by the guest operating system and can execute Ring 0 functions without interrupting guest users or host users.

Ring 0 functions in the supervisor mode that does not require any input interaction from the user. If any interaction made to this mode could result in security threats and a high risk of system errors. Due to these security reasons, this level is not accessible to all the host users. Linux and Windows operating system uses Ring 0 and Ring 3 for kernel and applications whereas modern OS systems with new CPU configurations used the other two ring levels also.

Protection of rings can be combined with any of the processer modes either with a supervisor or kernel or hypervisor. Hardware-supported systems use both either one or both layers of protection. Earlier versions of Windows 95 and Windows 98 provide fewer shielding factors between the privilege level of rings and this in turn leads to more security errors.

Conclusion

An operating system that uses protection rings to restrict access and operations by enabling different hierarchy levels of security improves fault tolerance issues. These extra layers of rings to each process can reduce the performance of the system and maintaining these privilege levels is quite a tedious task. Recent advancements in the processor built in various operating systems use the combination of protection rings to support extra security and improve the overall performance of the system when used in the network environment.

DOMAIN OF PROTECTION

The **careful use of access restrictions** can help make a system more secure and can also be beneficial in producing an **audit trail. Audit trail tracks divergences** from allowed accesses. If monitored closely, it can **reveal early warnings of an attack or provide clues as to which attack vectors were used, as well as accurately assess the damage caused.**

Perhaps most importantly, no single principle is a solution for security vulnerabilities. **Defence in depth** must be used, **multiple layers of protection** should be applied one on top of the other. At the same time, of course, **attackers use multiple means to bypass defence** in depth, resulting in an ever-escalating arms race. A **computer system** can be treated as 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.

Each **object has a unique name** that differentiates it from all other objects in the system. Each can be **accessed only through well-defined and meaningful operations.** The operations that are possible depend on the object.

For example,

- On a CPU, we can only execute
- Memory words can be read and written
- A 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 **objects for which it has authorization**. Furthermore, at any time, a process should be able to access only those objects that **it currently requires to complete its task.** This second requirement, the **need-to-know principle,** is useful in **limiting the amount of damage a faulty process or an attacker** can cause in the system.

For example, when **process p invokes procedure A( ).** The procedure should be **allowed to access** only its **own variables and the formal parameters** passed to it. It should not be able to access **all the variables of process p.** Similarly, consider the case in which **process p invokes a compiler to compile a particular file.**

The compiler should not be able to access files arbitrarily but should have **access only to a well-defined subset of files** (such as the source file, output object file, and so

on) related to the file to be compiled. In comparing **need-to-know with least privilege**, it may be easiest to think of **need-to-know as the policy** and **least privilege as the mechanism** for achieving this policy.

For example, in file permissions, **need-to-know** might dictate that a **user have read access but not write** or execute access to a file. The principle of **least privilege** would require that the operating system provide a mechanism **to allow read but not write or execute access.**

DOMAIN STRUCTURE

A process may operate 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.** It cannot, however, perform any other operation on that object. **Domains may share access rights.**

For example, we have three domains: D1, D2, and D3. The access right <O4, {print}> is shared by D2 and D3, implying that a process executing in either of these two domains can print object O4. Note that a process must be executing in domain D1 to read and write object O1, while only processes in domain D3 may execute object O1.

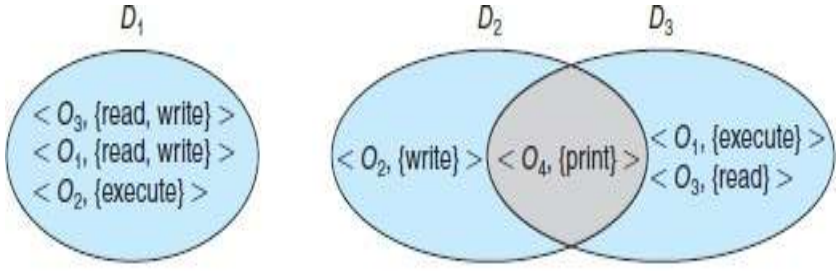


Figure System with three protection domains.

The association between a process and a domain may be either **Static or Dynamic. Static, if** the set of resources available to the process is fixed throughout the process's lifetime. **Establishing dynamic protection** domains is **more complicated** than establishing static protection domains. 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.**

Consider the standard dual-mode (kernel–user mode)model of operating system execution. When a process is in kernel mode, it can execute privileged instructions and thus gain complete control of the computer system. In contrast, when a process executes in user mode, it can invoke only non-privileged instructions.

Consequently, it can execute only within its predefined memory space. These two modes protect the operating system (executing in kernel domain) from the user processes (executing in user domain). In a multiprogrammed operating system, two protection domains are insufficient, since users also want to be protected from one another.

ACCESS MATRIX

The general model of protection can be viewed abstractly as a matrix, called an **access matrix.** The **rows represent domains,** and the **columns represent objects. Each entry** in the matrix consists of a **set of access rights.** Because the column defines objects explicitly, we can omit the object name from the access right.

The **entry access(i, j)** defines the **set of operation that a process executing in domain D_i can invoke on object O_j.**

To illustrate these concepts, we consider the access matrix shown in figure. There are four domains and four objects, three files F1, F2, F3 and one laser 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. The laser printer can be accessed only by a process executing in domain D2.

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 Access matrix.

The access-matrix scheme provides us with the **mechanism for specifying a variety of policies.** More specifically, we must ensure that a process executing in domain D_i can access only those objects specified in row *i,* and then only as allowed by the access-matrix entries. The **access matrix can implement policy decisions concerning protection.** The policy decisions involve which rights should be included in the $(i, j)^{th}$ entry.

The access matrix provides an appropriate mechanism for defining and **implementing strict control for both static and dynamic association between processes and domains.** When we **switch a process** from one domain to another, we are **executing an operation (switch) on an object (the domain).** We can **control domain switching by including domains** among the objects of the access matrix. Processes 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, a process executing in domain D2 can switch to domain D3 or to domain D4. A process in domain D4 can switch to D1, and one in domain D1 can switch to D2.

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 Access matrix with domains as objects.

IMPLEMENTATION OF ACCESS MATRIX

There are various methods of implementing the access matrix in the operating system. They are

- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- Lock-Key Mechanism

GLOBAL TABLE

The **simplest implementation** of the access matrix is a **global table consisting of a set of ordered triples <domain, object, rights-set>.** Whenever an operation M is executed on an object O_j within domain D_i, the **global table is searched for a triple <Di, Oj, R_k>, with M ∈ R_k.** If this **triple is found**, the **operation is allowed** to continue**; otherwise, an exception (or error) condition is raised.**

This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, this object must have a separate entry in every domain.

ACCESS LIST OF OBJECTS

Each **column** in the access matrix can be **implemented as an access list for one object.** Obviously, the empty entries can be discarded. The **resulting list** for each object consists of **ordered pairs <domain, rights-set>**, which define all domains with a nonempty set of access rights for that object. This approach can be extended easily to

define a list plus a **default set of** access rights.

When an operation M on an object Oj is attempted in domain Di, we search the access list for object Oj, looking for an entry <Di, Rk> with M ∈ Rk. 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.

CAPABILITY LISTS OF DOMAINS

Rather than associating the columns of the access matrix with the objects as access lists, **we can associate each row with its domain.** 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 physical 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.** The **capability list is associated with a domain**, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a **protected object**, maintained by the OS and accessed by the user **only indirectly.**

To provide **inherent protection**, we must **distinguish capabilities** from other kinds of **objects.** Capabilities are usually distinguished from other data in one of two ways: Each object has a **tag** to denote whether it is a capability or accessible data. Alternatively, 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.

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 **locks.** Similarly, **each domain** has a **list of unique bit patterns** called **keys. 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.** As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not allowed to examine or modify the list of keys (or locks) directly.

END OF MODULE-5