Module 6
Virtualization and File System Management

Virtual Machines - Virtualization (Hardware/Software, Server, Service, Network - Hypervisors - Container virtualization - Cost of virtualization - File system interface (access methods, directory structures) - File system implementation (directory implementation, file allocation methods) - File system recovery - Journaling - Soft updates - Log-structured file system - Distributed file system.

FILE SYSTEM INTERFACE

- File System provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system.
- The file system consists of two distinct parts:
  - a collection of files, each storing related data
  - a directory structure, which organizes and provides information about all the files in the system.
- Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks.
- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file.
- Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

FILE CONCEPTS

- A file is defined as a named collection of related information that is stored on secondary storage device.
- Different types of information may be stored in a file such as source or executable programs, numeric or text data, alphabetic, alphanumeric, or binary, photos, music, video, and so on.
- A file is a sequence of bits, bytes, lines, or records.
- A file has a certain defined structure, which depends on its type.
- Types of Files:
  - A text file is a sequence of characters organized into lines.
  - 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's attributes vary from one operating system to another but consist of these:

- **Name:** The file name is the 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).
- **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.

FILE OPERATIONS

- A file is an abstract data type.
- The operating system can provide system calls to create, open, write, read, reposition, delete, and truncate files.
- The basic operations on a file includes
  - Creating a File
  - Opening a File
  - Writing a File
  - Reading a File
  - Repositioning within a File
  - Deleting a file
  - Truncating a file

FILE OPERATIONS

- **Creating a file:** OS first finds space in the file system for the file. Second, an entry for the new file must be made in the directory.
- **Opening a file:** File operations specify a file name, causing the operating system to evaluate the name, check access permissions and call 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 specifying both the name of the file 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.
- **Reading a file:** To read from a file, use a system call that specifies the name of the file. The system needs to keep a read pointer to the location in the file where the next read is to take place.
  - Current File Position Pointer: 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.
- **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 file operation is also known as files 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.

Open File Table

- The file operations involve searching the directory for the entry associated with the named file.
- The operating system keeps a table, called the open-file table, containing information about all open files.
- When a file operation is requested, the file is specified via an index into this table, so no searching is required.
- When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table.
- The open-file table also has an open count associated with each file to indicate how many processes have the file open.
- Each close() decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.
- When several processes may open the file , the operating system uses two levels of internal tables:
- A per-process table: The per process table tracks all files that a process has open and information regarding the process's use of the file such as the current file pointer, access rights.
- A system-wide table: The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size.

FILE LOCKS

- File locks allow one process to lock a file and prevent other processes from gaining access to it.
- File locks are useful for files that are shared by several processes.
  - Shared Lock
  - Exclusive Lock
- A shared lock is similar to a reader lock in that several processes can acquire the lock concurrently.
- An exclusive lock behaves like a writer lock; only one process at a time can acquire such a lock.
- Operating systems provide either mandatory or advisory file-locking mechanisms.
  - If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file.
  - If a lock is advisory then the operating system will allow the process to access the locked file.
- 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 or a dot.
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

FILE TYPES

- 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.
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a .com, .exe, or .sh extension can be executed.
- The .com and .exe files are two forms of binary executable files, whereas the .sh file is a shell script.
- Application programs also use extensions to indicate file types. For example, Java compilers expect source files to have a .java extension, and the Microsoft word processor expects its files to end with a .doc or .docx extension.

FILE TYPES

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

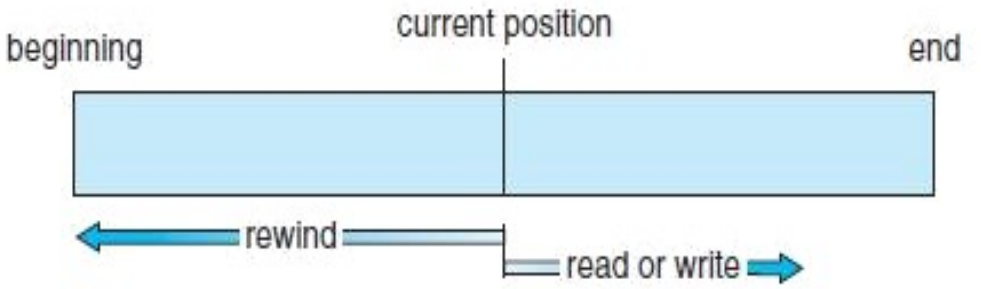
Figure 13.3 Common file types.

FILE 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.
  - Sequential access Method
  - Direct Access method
  - Indexed 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.
- Example: Editors and compilers usually access files in sequential manner.
- A read operation—read_next ()—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- The write operation—write next ()—appends to the end of the file and advances to material (the new end of file).



Direct Access Method

- It is also called Relative Access.
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- For direct access, the file is viewed as a numbered sequence of blocks or records.
- Direct-access files are of great use for immediate access to large amounts of information.
- EXAMPLE: Databases. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.
- For the direct-access method, the file operations must be modified to include the block number as a parameter.
- Thus, we have read (n) rather than write next ().

sequential access	implementation for direct access
reset	cp
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Direct Access Method

- 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.
- Example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number.
- Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read(n), where n is the block number, rather than read next(), and write(n) rather than write next().
- An alternative approach is to retain read next() and write next() and to add an operation position file(n) where n is the block number. Then, to effect a read(n), we would position file(n) and then read next().

Indexed Access Methods

- These methods generally involve the construction of an index for the file.
- The index, is like an index in the back of a book, contains pointers to the various blocks.
- To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.
- 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 index file contains pointers to secondary index files, which point to the actual data items.

DIRECTORY STRUCTURE

- The Operating System divides the hard disk into various partitions and stores its File System in each partition.
- Any entity containing a file system is known as volume. Each volume that contains a file system must also contain information about the files in the system.
- This information is kept in entries in a device directory or volume table of contents.
- The device directory records information such as name, location, size, and type for all files on that volume.
- The directory translates file names into their directory entries.
- The directory can be viewed as a symbol table that translates file names into their file control blocks.

DIRECTORY STRUCTURE

Operations that are to be performed on a directory includes

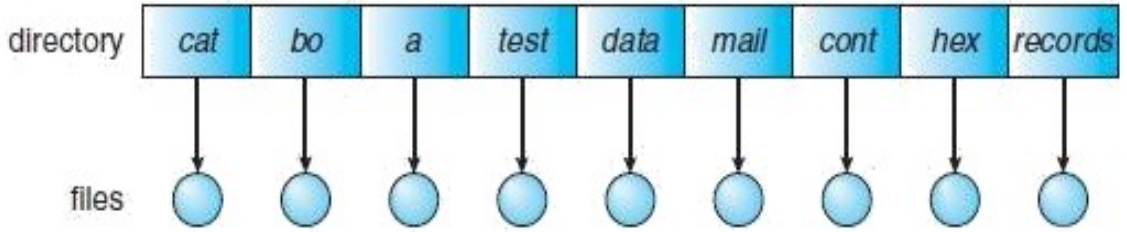
- Search for a file - We need to search a directory structure to find the entry for a particular file.
- Create a File - New files need to be created and added to the directory
- Delete a file - When a file is no longer needed, we want to be able to remove it from the directory. Note a delete leaves 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 - 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.

DIRECTORY STRUCTURE

- Types of Directories
  - Single level Directory
  - Two Level Directory
  - Tree Structured Directory
  - Acyclic Graph directory
  - General Graph directory

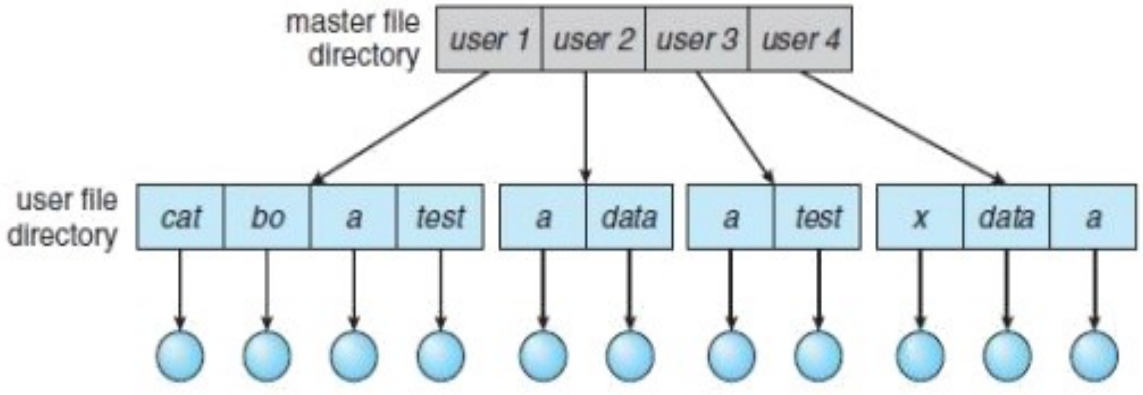
Single level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory.
- When the system has more than one user, the files created by different users should have unique names because all the files will be stored under a single directory.
- When the number of files increases the user may find it difficult to remember the names of all the files.



Two Level directory

- In the two-level directory structure, each user has his own directory called 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.
- 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 check whether another file of that name exists.
- To delete a file, the ope annot accidentally delete another user's file



Tree structured directory

- Tree structured directory is an expansion of two level directory extended to a tree of arbitrary height.
- It allows users to create their own subdirectories and to organize their files.
- The tree has a root directory, and every file in the system has a unique path name.
- The current directory should contain most of the files that are of current need to the process.
- Path names can be of two types:
- Absolute path name: An absolute path name begins at the root and follows a path down to the specified file.
- Relative path name: A relative path name defines a path from the current directory.
- Users can be allowed to access the files of other users by specifying the path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- A bit in each directory as a subdirectory (1).

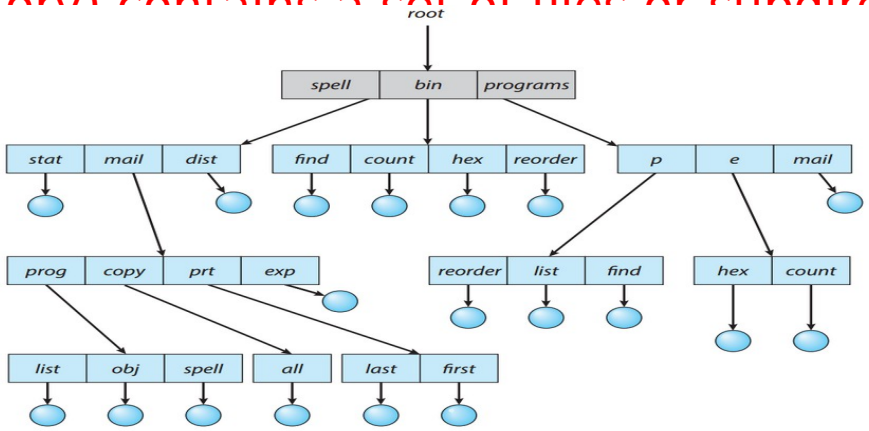


Figure 13.9 Tree-structured directory structure.

FILE DELETION IN TREE STRUCTURED DIRECTORY

- Special system calls are used to create and delete directories.
- If a directory is empty, its entry in the directory that contains it can simply be deleted.
- If the directory to be deleted is not empty and contains several files or subdirectories, one of two approaches can be taken.
  - Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can also be deleted.
  - If when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. This approach is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command is issued in error, a large number of files and directories will need to be restored.

Acyclic Graph Directories

- An acyclic graph is a graph with no cycles.
- It allows directories to share subdirectories and files.
- The same file or subdirectory may be in two different directories.
- The acyclic graph is a natural generalization of the tree structured directory scheme.
- With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.
- Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.
- All the files the user wants to share can be put into one directory.
- The home directory could contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories.
- Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal.

Acyclic Graph Directories

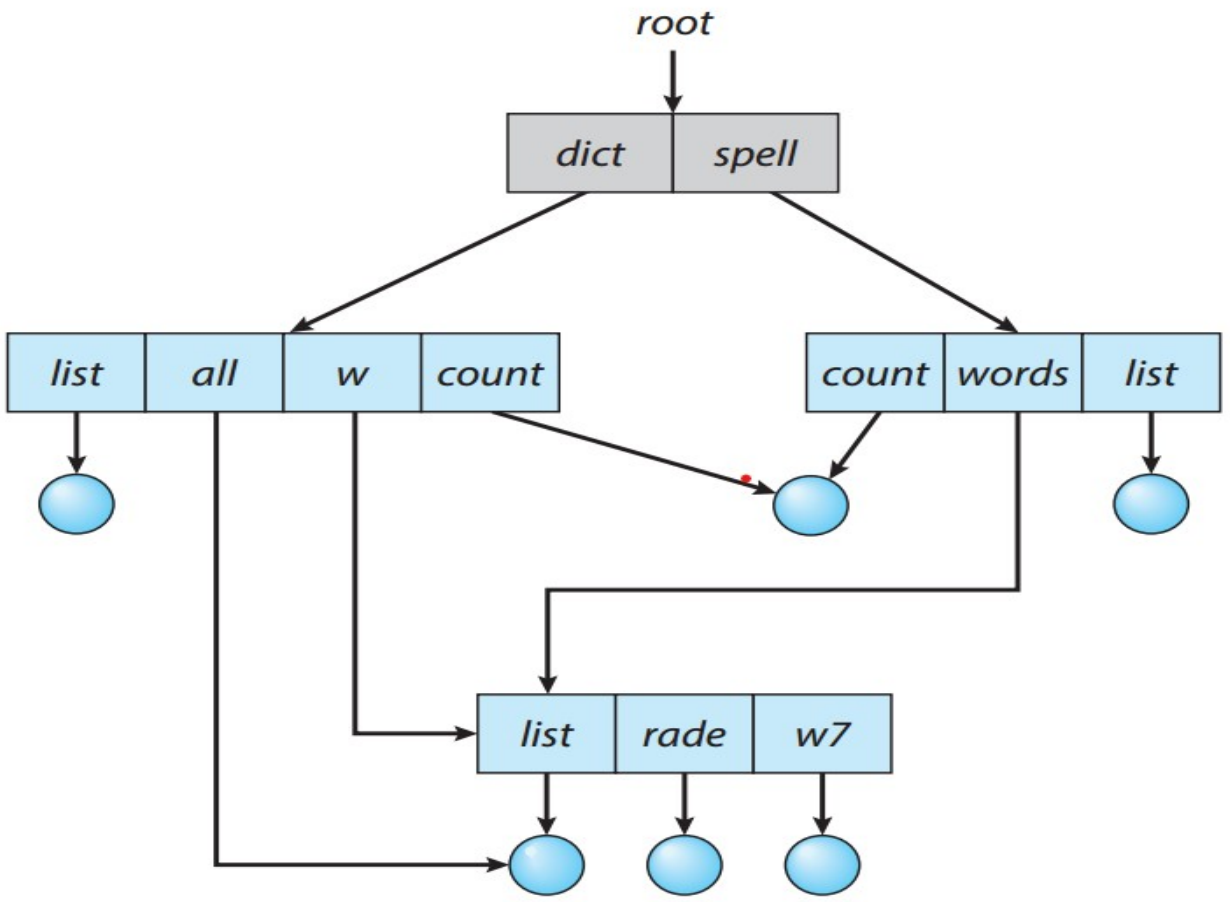
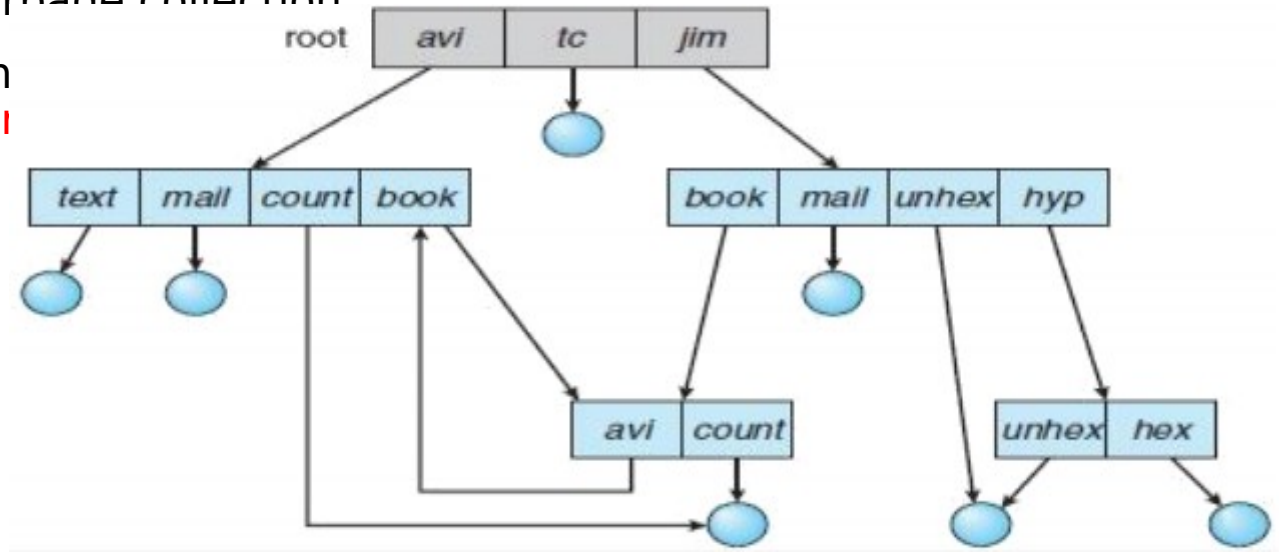


Figure 13.10 Acyclic-graph directory structure.

General Graph Directory

- If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results.
- Adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature.
- When we add links, the tree structure is destroyed, resulting in a simple graph structure.
- If cycles are allowed to exist in the directory, we want to avoid searching any component twice, to improve the performance.
- With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.
- However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file.
- In this case we use Garbage collection
- Garbage collection: Th s been deleted and the disk space can be



File System Implementation

File-System Structure

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics for this purpose are:

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.

Nonvolatile memory (NVM) devices are increasingly used for file storage. They differ from hard disks in that they cannot be rewritten in place and they have different performance characteristics.

To improve I/O efficiency, I/O transfers between memory and mass storage are performed in units of blocks.

Each block on a hard disk drive has one or more sectors.

Depending on the disk drive, sector size is usually 512 bytes or 4,096 bytes. NVM devices usually have blocks of 4,096 bytes, and the transfer methods used are similar to those used by disk drives.

File System Implementation

File-System Structure

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 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.

File System Implementation

File-System Structure

The file system is generally composed of many different levels. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The file system structure is an example of a layered design and it is shown in the below fig.

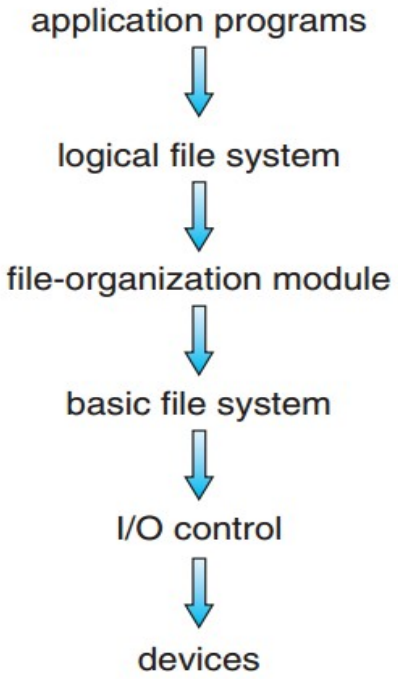


Figure 14.1 Layered file system.

File System Implementation

I/O Control Level:

- The I/O control level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- A device driver can be thought of as a translator. Its input consists of high level commands, such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system.
- The device driver usually 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.

File System Implementation

The Basic File System:

- The basic file system (called the “block I/O subsystem” in Linux) 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.
- This layer also manages the memory buffers and caches that hold various file system, directory, and data blocks.
- A block in the buffer is allocated before the transfer of a mass storage block can occur.
- When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.
- Caches are used to hold frequently used file-system metadata to improve performance, so managing their contents is critical for optimum system performance.

File System Implementation

The File Organization module:

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.

The Logical File System:

The logical file system manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files).

The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name.

It maintains file structure via file-control blocks. A file control block (FCB) (an inode in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection.

File System Implementation

- When a layered structure is used for file-system implementation, duplication of code is minimized.
- The I/O control and 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.
- 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.
- UNIX uses the UNIX fil system (UFS), which is based on the Berkeley Fast File System (FFS).
- Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats.
- Linux supports over 130 different file systems, the standard Linux file

Structures and Operations for File System Implementation

- 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.
- 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.
- A boot 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 operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the boot block. In NTFS, it is the partition boot sector.
- A volume control block (per volume) 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. In UFS, this is called a superblock. In NTFS, it is stored in the master fil table.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.
- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

Structures and Operations for File System Implementation

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.

- 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. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- The system-wide open-fil table contains a copy of the FCB of each open file, as well as other information.
- The per-process open-fil 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 th … The logical file system knows the format of the directory struct … file, it allocates a new FCB. The system then reads the appropri… …ory, updates it with the new file name and FCB, and writes it bac… …typical FCB is shown in Figure 14.2.

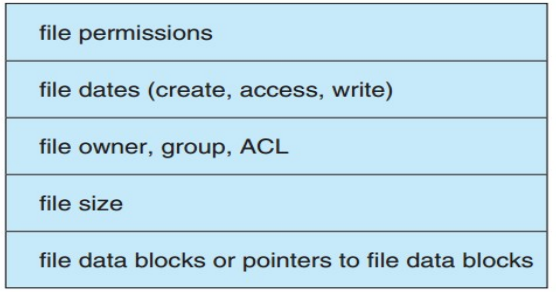


Figure 14.2 A typical file-control block.

Directory Implementation

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

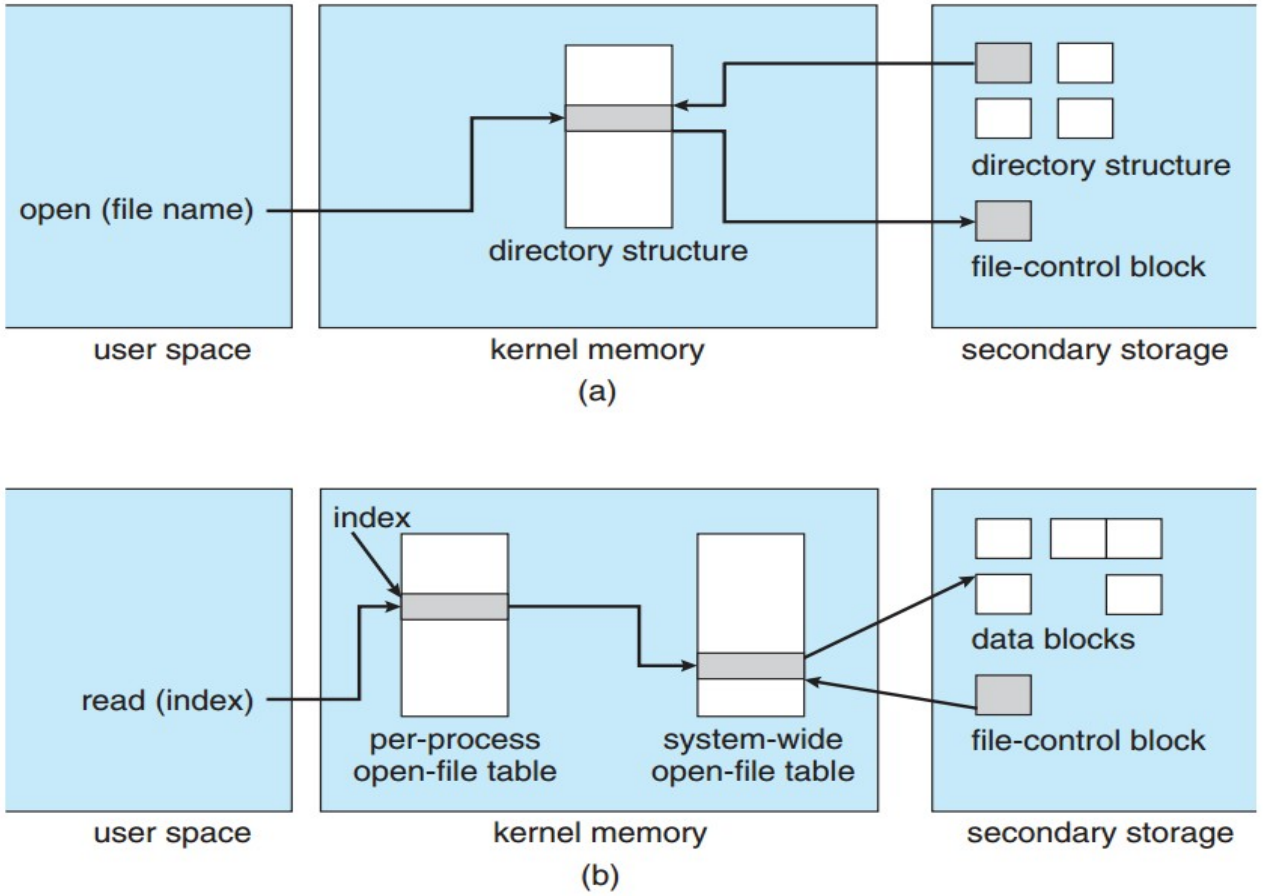


Figure 14.3 In-memory file-system structures. (a) File open. (b) File read.

Directory Implementation

Linear List:

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- 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 unused (by assigning it a special name, such as an all-blank name, assigning it an invalid inode number (such as 0), or by including a used –unused bit in each entry), 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

Directory Implementation

Linear List:

The 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. Many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread 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 tree data structure, such as a balanced tree, might be useful.

An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

Directory Implementation

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 possible but 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).

Directory Implementation

Hash Table:

Alternatively, we can use a chained-overflow hash table.

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

Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries.

Still, this method is likely to be much faster than a linear search through the entire directory.

Allocation Methods

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,
- linked,
- and indexed.

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 (from the last sector of one cylinder to the first sector of the next cylinder), 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 (assuming blocks with close logical addresses are close physically), as is seek time when a seek is finally needed.

Contiguous Allocation

- 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 (Figure 14.4).
- Contiguous allocation is easy to implement but has limitations, and is therefore not used in mode

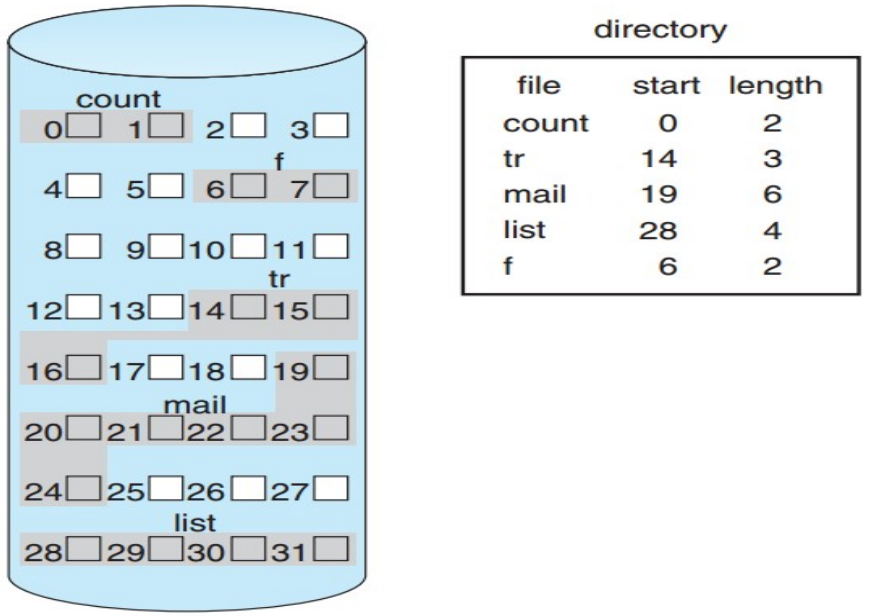


Figure 14.4 Contiguous allocation of disk space.

Contiguous Allocation

- 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.
- Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished;
- The contiguous-allocation problem can be seen as a particular application of the general dynamic storage-allocation problem, which involves how to

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.
- 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, and a block address (the pointer) requires 4 bytes, then 508 bytes are available to a user.

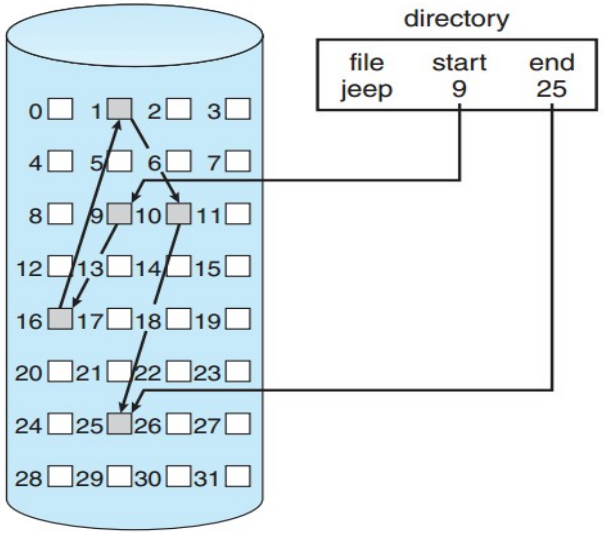


Figure 14.5 Linked allocation of disk space.

Linked Allocation

- 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 (the end-of-list pointer value) to signify an empty file.
- The size field is also set to 0. A write to the file causes the free space management system to 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.

Linked Allocation

- Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files.
- To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block. Each access to a pointer requires a storage device read, and some require an HDD seek. (Larger seek time)
- Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.
- The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the secondary storage device only in cluster units.
- Another problem of linked allocation is reliability. Recall that the files are linked

Linked Allocation

- An important variation on linked allocation is the use of a file-allocation table (FAT). This efficient method of disk-space allocation was used by the MS-DOS operating system.
- A section of storage at the beginning of each volume is set aside to contain the table. The table has one entry for each block and is indexed by block number.
- 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.

Linked Allocation

- An illustrative example is the FAT structure shown in Figure 14.6 for a file consisting of disk blocks 217, 618, and 339.
- The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.
- The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the loc
- In the worst case, both moves occur for each of the bl

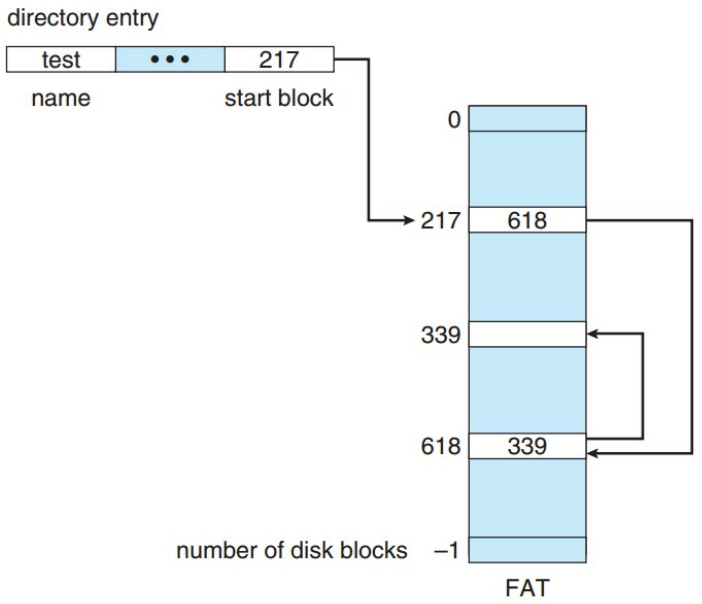


Figure 14.6 File-allocation table.

Indexed Allocation

- Linked allocation solves the 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 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 ith entry in the index block points to the ith block of the file. The directory contains the address of the index block (Figure 14.7).
- To find and read the ith block, we use the pointer in the ith index-block entry.
- When the file is created, all pointers in the index block are set to null.
- When the ith block is first written, a block is obtained from the free-space

Indexed Allocation

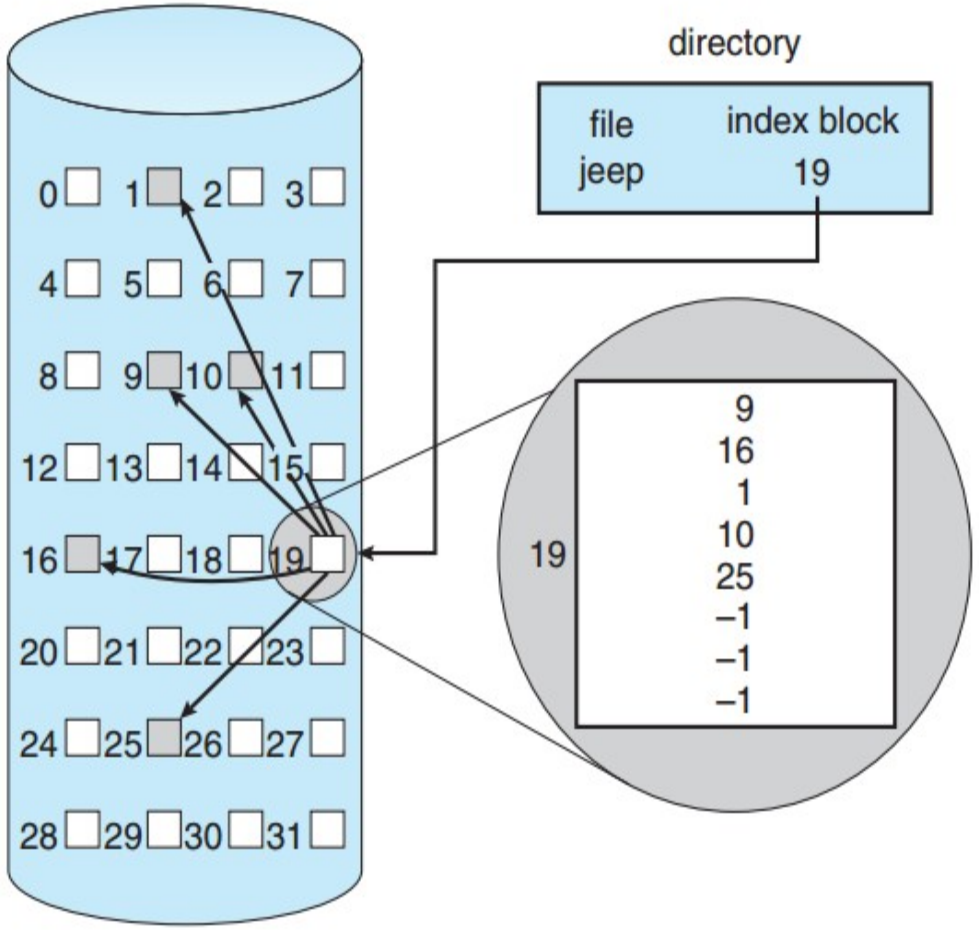


Figure 14.7 Indexed allocation of disk space.

Indexed Allocation

- 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

Indexed Allocation

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

Indexed Allocation

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. (A UNIX inode is shown in

Indexed Allocation

Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems.

A 32-bit file pointer reac

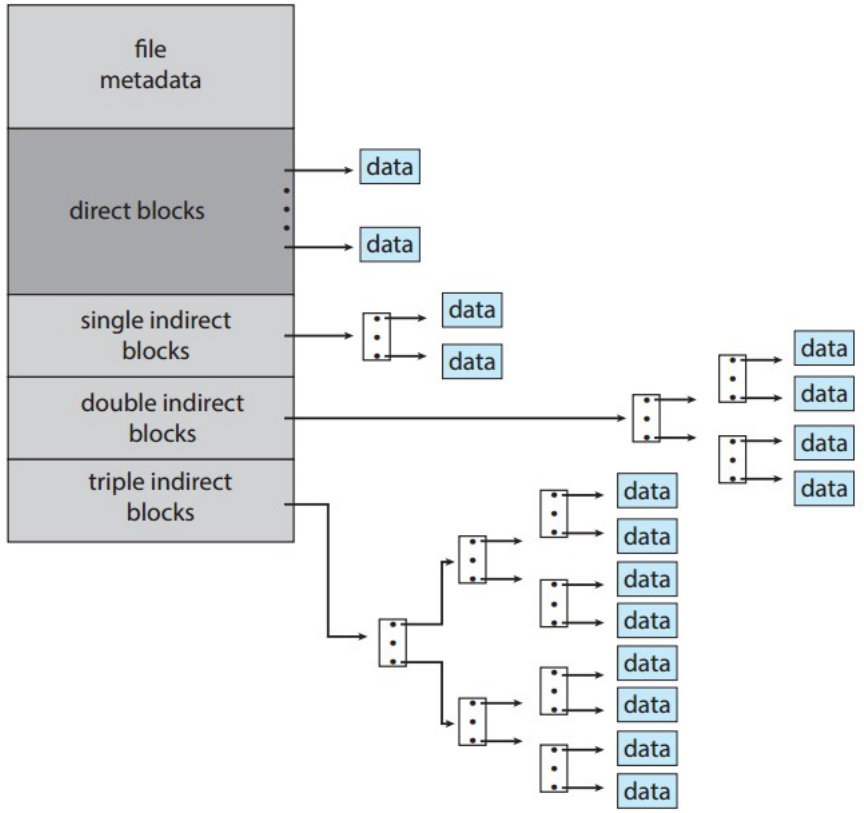


Figure 14.8 The UNIX inode.

Recovery

- Files and directories are kept both in main memory and on the storage volume, and care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency.
- A system crash can cause inconsistencies among on-storage file-system data structures, such as directory structures, free-block pointers, and free FCB pointers.
- A typical operation, such as creating a file, can involve many structural changes within the file system on the disk.
- Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased.
- These changes can be interrupted by a crash, and inconsistencies among the structures can result

Consistency Checking

- A file system must first detect the cause of corruption and then correct them.
- For detection, a scan of all the metadata on each file system can confirm or deny the consistency of the system. This scan can take minutes or hours and should occur every time the system boots.
- Alternatively, a file system can record its state within the file-system metadata.
- At the start of any metadata change, a status bit is set to indicate that the metadata is in flux.
- If all updates to the metadata complete successfully, the file system can clear that bit. If, however, the status bit remains set, a consistency

Consistency Checking

- The consistency checker—a systems program such as fsck in UNIX compares the data in the directory structure and other metadata with the state on storage and tries to fix any inconsistencies it finds.
- The allocation and free-space-management algorithms dictate what types of problems the checker can find and how successful it will be in fixing them. For instance, if linked allocation is used and there is a link from any block to its next block, then the entire file can be reconstructed from the data blocks, and the directory structure can be recreated.
- In contrast, the loss of a directory entry on an indexed allocation system can be disastrous, because the data blocks have no knowledge of one another. For this reason, some UNIX file systems cache directory entries for reads, but any write that results in space allocation, or other metadata changes, is done synchronously, before the corresponding data blocks are written.

Log Structured File systems

- Log-based recovery algorithms have been applied successfully to the problem of consistency checking. The resulting implementations are known as log-based transaction-oriented (or journaling) file systems.
- There are several problems in consistency checking approach.
- The consistency check may not be able to recover the structures, resulting in loss of files and even entire directories.
- Consistency checking can require human intervention to resolve conflicts, and the system can remain unavailable until the human tells it how to proceed.
- Consistency checking also takes system and clock time.
- The solution to these problems is to apply log-based recovery

Log Structured File systems

- Fundamentally, all metadata changes are written sequentially to a log.
- Each set of operations for performing a specific task is a transaction. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution.
- Meanwhile, these log entries are replayed across the actual file system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, and entry is made in the log indicating that.
- The log file is actually a circular buffer. A circular buffer writes to the

Log Structured File systems

- If the system crashes, the log file will contain zero or more transactions.
- Any transactions it contains were not completed to the file system, even though they were committed by the operating system, must be completed.
- The transactions can be executed from the pointer until the work is complete so that the file-system structures remain consistent.
- The only problem occurs when a transaction was aborted— that is, was not committed before the system crashed.
- Any changes from such a transaction that were applied to the file system must be undone, again preserving the consistency of the file system.
- A benefit of using logging on disk metadata updates is that those

Back up and Restore

- Storage devices sometimes fail, and care must be taken to ensure that the data lost in such a failure are not lost forever.
- To this end, system programs can be used to back up data from one storage device to another, such as a magnetic tape or other secondary storage device.
- Recovery from the loss of an individual file, or of an entire device, may be a matter of restoring the data from backup.
- To minimize the copying needed, we can use information from each file’s directory entry.
- For instance, if the backup program knows when the last backup of a file was done, and the file’s last write date in the directory indicates that the

Back up and Restore

A typical backup schedule may then be as follows:

- Day 1. Copy to a backup medium all files from the file system. This is called a full backup.
- Day 2. Copy to another medium all files changed since day 1. This is an incremental backup.
- Day 3. Copy to another medium all files changed since day 2.
- ...
- Day N. Copy to another medium all files changed since day N−1. Then go back to day 1.
- The new cycle can have its backup written over the previous set or onto a new set of backup media.
- Using this method, we can restore an entire file system by starting