File system

File concept

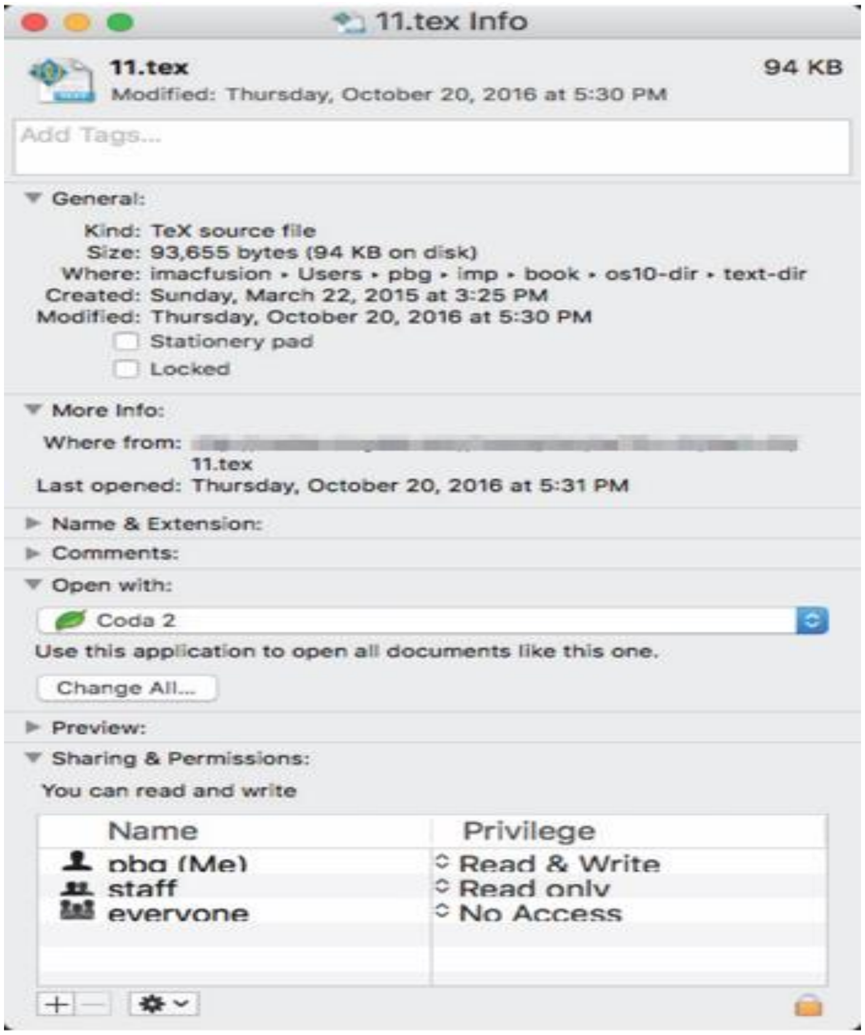
computers use various types of non-volatile storage devices (like HDDs, SSDs, tapes, etc.) to store data, and the operating system provides a consistent, logical view of this storage by using files. A file is a named collection of related information and is the smallest unit in which data can be stored on secondary storage. Files can contain different types of data such as text, programs, images, or system information and their structure depends on the type of content. The operating system abstracts physical storage through the use of files, making data storage and retrieval more manageable for users and applications

File attribute

A file is identified by a human-readable name, making it easy for users to reference, regardless of who created or uses it. Once created, a file becomes independent of its origin and can be copied or shared across systems while retaining its name. Additionally, files have various attributes such as name, type, location, size, access permissions, timestamps, and unique identifiers that help the operating system manage, protect, and track the file. Some modern file systems also include extended attributes like character encoding and security features.

- ✓ **Name** The symbolic file name is the only information kept in human readable form.
- ✓ **Identifie** 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.



11.tex Info

11.tex 94 KB
Modified: Thursday, October 20, 2016 at 5:30 PM

Add Tags...

General:

Kind: TeX source file
Size: 93,655 bytes (94 KB on disk)
Where: imacfusion ▸ Users ▸ pbg ▸ imp ▸ book ▸ os10-dir ▸ text-dir
Created: Sunday, March 22, 2015 at 3:25 PM
Modified: Thursday, October 20, 2016 at 5:30 PM
☐ Stationery pad
☐ Locked

More Info:

Where from: [illegible] 11.tex
Last opened: Thursday, October 20, 2016 at 5:31 PM

Name & Extension:

Comments:

Open with:

Coda 2
Use this application to open all documents like this one.
Change All...

Preview:

Sharing & Permissions:

You can read and write

Name	Privilege
pbg (Me)	Read & Write
staff	Read only
everyone	No Access

File operation

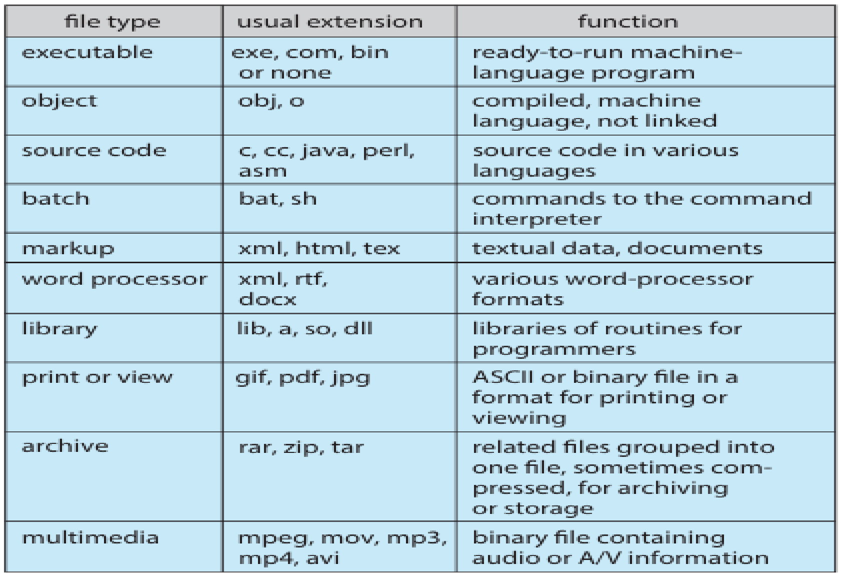
A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete and truncate files.

- **Creating a file** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current f ile-position pointer. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a 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 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 re create it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.
- **File pointer** On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Disk location** of the file Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- **Access rights** Each process opens a file in an access mode. This information is stored on the per process table so the operating system can allow or deny subsequent I/O requests.

File types

the importance of file types in operating systems and how they help the system and applications interact appropriately with files. A common way to indicate a file's type is by using a file extension (e.g., .docx, .exe, .java). These extensions help identify what kind of file it is and what operations can be performed on it. While some operating systems rely on extensions to recognize file types, others treat them as hints for applications. Recognizing file types helps prevent errors, such as trying to read binary files as text, and allows programs to find and work with the files they expect.



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 com-pressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

File structure

file types can define a file's internal structure, which is necessary for the operating system and applications to properly read and use the file. While supporting multiple file structures can make the OS more functional, it also increases complexity and size. To avoid this, many systems (like UNIX and Windows) keep file structures minimal and treat files simply as byte sequences, leaving structure interpretation to applications. However, the OS must still recognize certain essential file types, like executables, to function properly.

Internal file structure

Managing file data on disk involves dealing with differences between logical data organization and the physical storage structure of disks. Disks perform input/output operations in fixed-size units called blocks, which are determined by the hardware's sector size (e.g., 512 bytes per block). However, the logical records that applications or users work with—such as lines of text or individual bytes—often do not align perfectly with these fixed block sizes. As a result, the operating system or application must pack multiple logical records into physical blocks to make efficient use of disk space and ensure correct data access.

For instance, UNIX treats files simply as streams of bytes, where each byte is addressable by its offset from the beginning of the file. This provides flexibility and simplicity, allowing any program to access any part of a file without needing to worry about record boundaries. The operating system handles packing and unpacking of bytes into physical disk blocks behind the scenes, allowing developers to focus on the logical structure of their data rather than physical disk limitations.

However, this method introduces internal fragmentation, where the last block of a file may not be fully used. Since disk space is always allocated in whole blocks, any unused portion of the last block represents wasted space. For example, if a file is 1,949 bytes and the block size is 512 bytes, the system will allocate 2,048 bytes (four full blocks), leaving 99 bytes unused. This waste increases as block sizes grow, creating a trade-off between larger block sizes (which may improve performance) and efficiency of space usage.

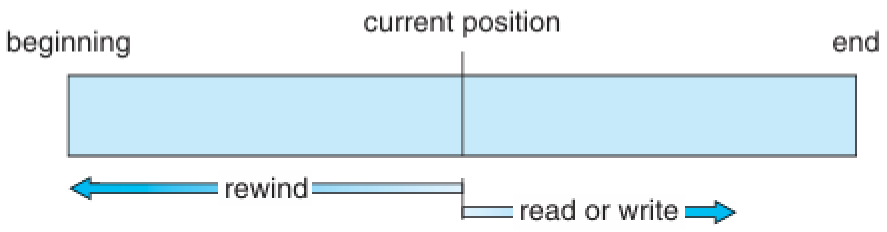
the operating system must bridge the gap between how data is logically organized and how it is physically stored. It does so by translating logical records into physical blocks and managing the resulting overhead, such as internal fragmentation. While this introduces some inefficiency, especially with larger block sizes, it is a necessary part of making file systems practical and efficient for a wide variety of applications and data types.

Access methods

Sequential access

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). Such a file can be reset to the beginning, and on some systems a program may be able to skip forward or backward n records for some integer n perhaps only for n = 1. Sequential access,



Sequential access file

which is depicted in the Figure above 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

the direct-access (or relative access) method for file systems, which allows programs to read or write fixed-length records in any order, unlike sequential access which processes data in a specific sequence. Direct access is especially useful for applications like databases or reservation systems, where fast, random access to specific records is essential. With this method, file operations are based on block numbers that reference the position of data relative to the file's start, rather than its physical location on disk. This abstraction gives the operating system flexibility in file placement and helps manage security.

Some systems support only sequential or direct access, while others require declaring the access type when a file is created. Although sequential access can be simulated on direct-access files easily, the reverse is inefficient. Overall, direct access enhances performance for data-heavy applications that need quick, targeted retrieval of information.

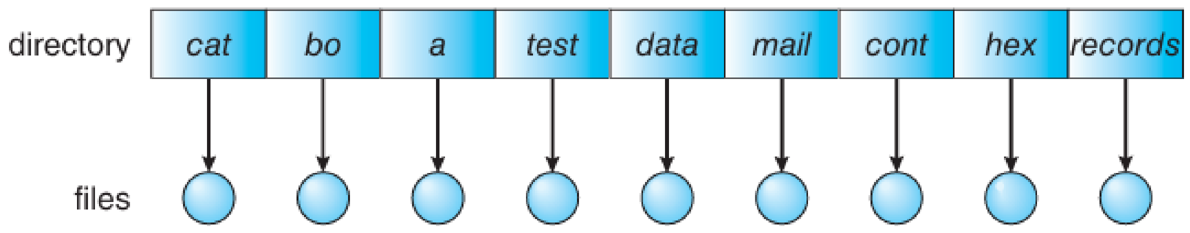
Directory structure

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. When considering a particular directory structure we need to keep in mind 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. Note a delete leaves a hole in the directory structure and the file system may have a method to defragement 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.

Single level directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file test.txt, then the unique-name rule is violated



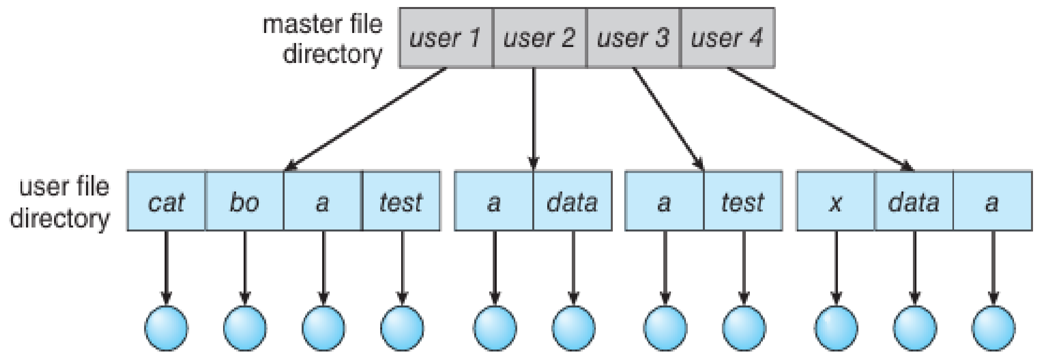
Single level directory

Two level directory

As we have seen a single level directory often leads to confusion of file names among different users. But In a **two-level directory structure** each user has a separate User File Directory (UFD), and all UFDs are listed in a Master File Directory (MFD). This structure helps prevent filename conflicts between users but limits collaboration, as it isolates user files. To allow access to another user's file, a user must know the full path name (e.g., /user b/test.txt), combining both the username and filename.

Different operating systems use different file naming conventions. For example, Windows uses volume letters (C:\user b\test.txt), while UNIX/Linux uses hierarchical paths (/u/pgalvin/test). Some systems (like OpenVMS) include volume, directory, and version information in file names.

For system files (e.g., loaders or compilers), storing copies in every UFD is inefficient. Instead, a special directory stores these files, and the OS searches user directories first, then this system directory, based on a search path. The search path can be customized, allowing flexible file and command location resolution, commonly seen in UNIX and Windows systems.



Two level directory structure

Tree structured directories

A tree-structured directory extends the two-level directory into a hierarchy of arbitrary depth, allowing users to create subdirectories to organize files logically. The structure starts with a root directory, and each file has a unique path name—either absolute (starting from root) or relative (from the current directory).

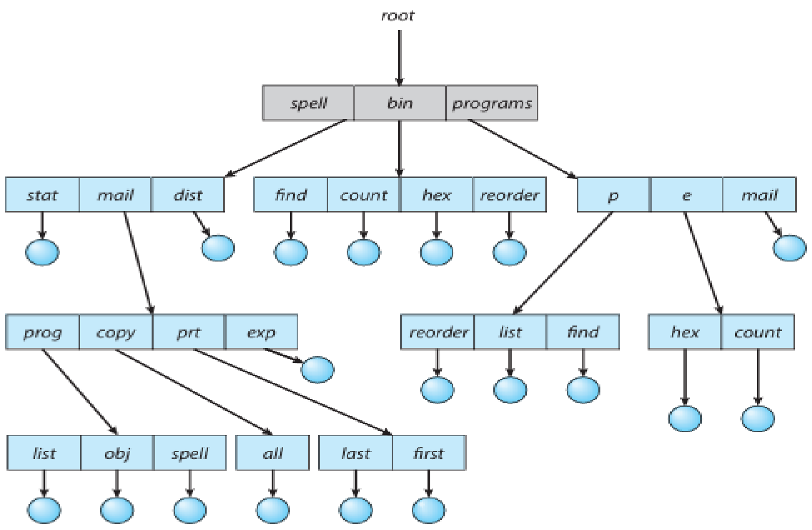
Each process typically has a current directory, set at login and inherited by subprocesses. Users can change this directory to access different files more easily.

Directories are special files with entries marked as either files or subdirectories, managed using system calls. This model provides flexibility, allowing logical grouping of files (e.g., separating source code and binaries).

Deleting directories can follow two policies:

- Only allow deletion if empty, requiring manual file removal.
- Recursive deletion, as in UNIX's rm -r, which removes all contents but risks accidental data loss.

Users can access other users' files by specifying full path names, supporting both file organization and file sharing.



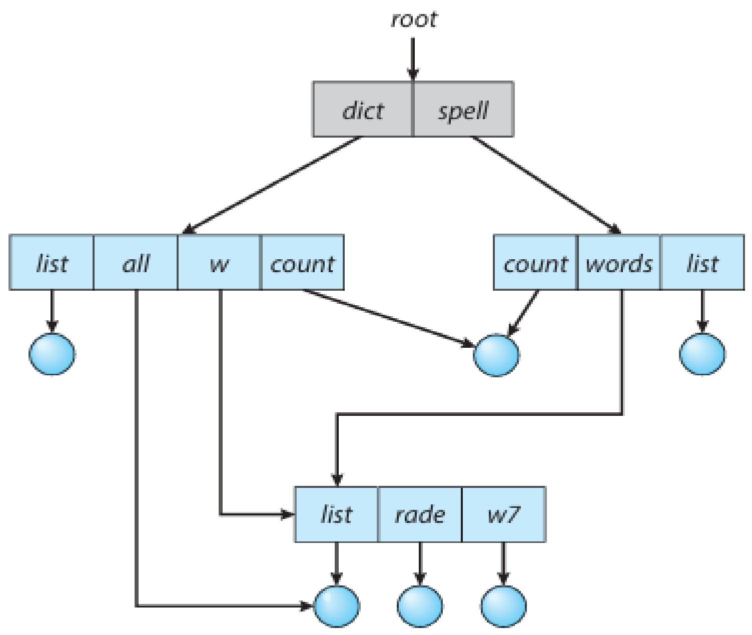
Acyclic graph directories

An acyclic graph directory structure is an extension of the tree structure that allows file and directory sharing across multiple locations in the file system without duplication.

1. **Purpose**: Enables multiple users or directories to share the same file or subdirectory, avoiding unnecessary duplication and ensuring changes are reflected everywhere.
2. **Structure**:
   - A tree prohibits sharing.
   - An acyclic graph (no cycles) allows files/subdirectories to exist in multiple directories.

3. **Implementation Methods**:
   - Links (symbolic or hard) point to the actual file or directory.
     - Symbolic links (soft): Store the path to the original file; may break if the target is deleted.
     - Hard links: Share the same inode and use a reference count to manage deletion.
   - Duplicate entries (not recommended): Cause consistency issues when a file is modified.
4. **Challenges**:
   - Multiple path names to the same file (aliasing problem).
   - Traversal issues during backups or system scans.
   - Deletion complications:
     - If one user deletes a file, others may be left with dangling pointers.
     - Reference counting is used (as in UNIX) to delete a file only when all references are removed.
5. **Design Considerations**:
   - Some systems (for simplicity and safety) do not allow shared directories or links.
   - Proper management of links and deletion policies is crucial to maintain file system integrity.

An acyclic graph directory offers flexibility for sharing but adds complexity in managing references, updates, and deletions. Systems like UNIX manage this using reference counts and links while avoiding cycles to maintain structure.



Acyclic-graph directory structure

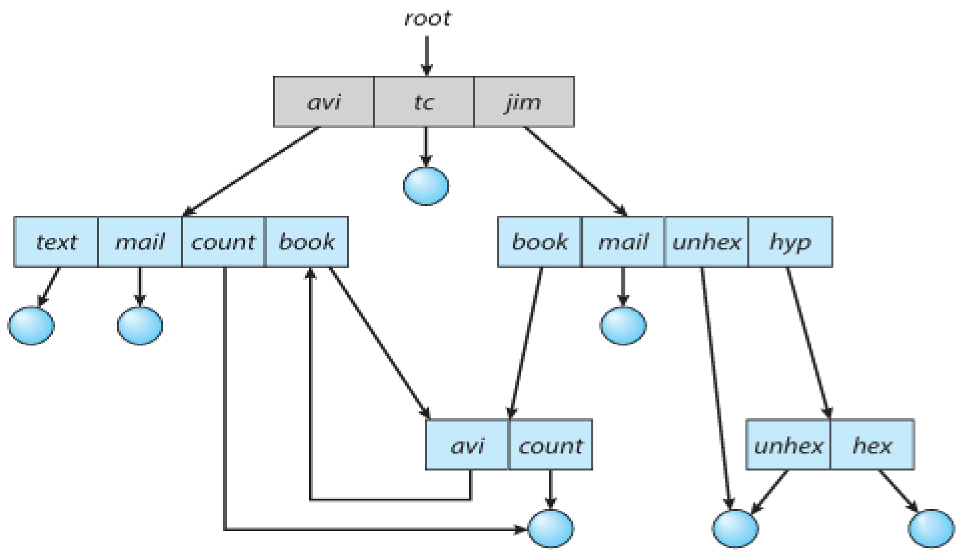
General graph directory

An acyclic graph directory structure is designed to allow file and directory sharing while avoiding cycles to maintain simplicity and performance. However, introducing links can unintentionally create cycles, which cause serious issues.

1. **Acyclic vs. Cyclic Structure:**
    - A tree structure is naturally acyclic.
    - Adding links can turn it into a graph, and if not managed carefully, cycles can form.
2. **Problems Caused by Cycles:**
    - Infinite loops during file searches or traversal due to re-entering the same directories.
    - Incorrect deletion handling because files in a cycle might still show non-zero reference counts even if they're no longer accessible.

3. **Garbage Collection Requirement**:
   - If cycles exist, reference counts alone are unreliable.
   - Garbage collection (mark-and-sweep) is needed to identify and delete unreachable files.
   - However, garbage collection is slow and rarely used in disk-based systems.
4. **Cycle Prevention Strategies:**
   - Use cycle detection algorithms (costly in disk-based graphs).
   - A practical approach is to ignore links during traversal, which avoids cycles and overhead.

To keep directory management efficient and safe, acyclic structures are preferred. Allowing cycles introduces complexity, risks infinite loops, and necessitates expensive garbage collection. Thus, systems must carefully prevent cycles when implementing file sharing via links.



General graph directory

Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection). Reliability is generally provided by duplicate copies of files.Manycomput ers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surgesor failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system soft ware can also cause file contents to be lost.

Protection can be provided in many ways. For a laptop system running a modern operating system, we might provide protection by requiring a user nameandpasswordauthentication to access it, encrypting the secondary stor age so even someone opening the laptop and removing the drive would have a difficult time accessing its data, and firewalling network access so that when it is in use it is difficult to break in via its network connection. In multiuser system, even valid access of the system needs more advanced mechanisms to allow only valid access of the data.

Types of access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, wecould providecomplete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access. Protection mechanisms provide controlled access by limiting the types of f ile access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read** Read from the file.
- **Write** Write or rewrite the file.
- **Execute** Load the file into memory and execute it.
- **Append** Write newinformation at the end of the file.
- **Delete** Delete the file and free its space for possible reuse.
- **List** List the name and attributes of the file.
- **Attribute change** Changing the attributes of the file.

Access control

The main idea is that file and directory access in operating systems is controlled based on user identity, using Access Control Lists (ACLs) and simplified user classifications (Owner, Group, Other) to manage permissions efficiently.

1. **Access Control Lists (ACLs):**
   - Each file/directory can have an ACL specifying which users have which types of access (read, write, execute).
   - Offers fine-grained control but can be long and hard to manage.
2. **Simplified Access Scheme**:
   - Most systems simplify by using three user classes:
     - Owner: The file creator.
     - Group: A set of users with shared access.
     - Other: All remaining users.
   - Each class has read (r), write (w), and execute (x) permissions.
3. **Combining ACLs with User Classes**:
   - Modern systems like UNIX and Solaris use the owner/group/other model by default, with ACLs added only when finer control is needed.
   - Example: A project team can be a group, while temporary access for an outsider can be managed via ACLs.

4. **Implementation Details**:
    - UNIX: Uses 9 permission bits (rwx for each class), with optional ACLs indicated by a "+" sign in listings.
    - Solaris: Uses commands like setfacl and getfacl.
    - Windows: Uses a GUI to manage ACLs and permissions.
5. **Permission Conflicts**:
    - When ACLs and standard group permissions conflict, ACLs take precedence, as they provide more specific rules.

ACLs provide flexible and detailed control over file access, while user classifications (owner, group, other) simplify standard permission settings. Most systems combine both approaches for efficiency and fine-tuned security, with ACLs overriding default permissions when conflicts arise.

Other protection approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files then once it is discovered all files are accessible protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory rather than with an individual file to address this problem. In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes knowledge of the existence and name of a file is significant in itself.

Thus, listing the contents of a directory must be a protected operation. Similarly, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic and general graphs), a given user may have different access rights to a particular file, depending on the path name used.

Memory mapped files

There is one other method of accessing files, and it is very commonly used. Consider a sequential read of a file on disk using the standard system calls open(), read(),andwrite(). Each file access requires a system call and disk access. Alternatively, we can use the virtual memory techniques in to treat file I/O as routine memory accesses. This approach, known as memory mapping a file, allows a part of the virtual address space to be logically associated with the file. As we shall see, this can lead to significant performance increases.

Basic mechanism

Memory-mapped files are a powerful mechanism used by modern operating systems to optimize file access and enable interprocess communication. Instead of reading and writing files through traditional system calls like read() and write(), memory mapping allows a file to be directly mapped into the virtual memory space of a process. This approach not only improves performance by eliminating the overhead of frequent system calls but also simplifies file manipulation by treating file content as if it were part of the main memory.

When a file is memory-mapped, the operating system links disk blocks to pages in virtual memory. The first time a part of the file is accessed, a page fault occurs, prompting the OS to load that portion of the file from disk into a physical memory page. Subsequent accesses are handled like ordinary memory operations. Writes to the file, however, are not immediately saved to disk. Instead, they are temporarily stored in memory and only written back to disk either when the file is closed or under memory pressure, ensuring data is not lost.

Different operating systems implement memory mapping in different ways. For instance, Solaris automatically memory-maps files, whether they are accessed through explicit memory-mapping system calls (like mmap()) or through traditional file access methods. If accessed through standard system calls, Solaris maps the file to the kernel address space, while memory-mapped files are placed in the process address space. In either case, the file I/O benefits from the speed and efficiency of memory operations.

An important advantage of memory-mapped files is their ability to enable data sharing between processes. When multiple processes map the same file, they share the same physical memory pages, so changes made by one process can be instantly seen by the others. This sharing mechanism can be controlled using mutual exclusion methods to ensure consistency and synchronization. Additionally, copy-on-write can be used to allow multiple processes to share read-only access while creating separate memory copies if any process attempts to modify the data.

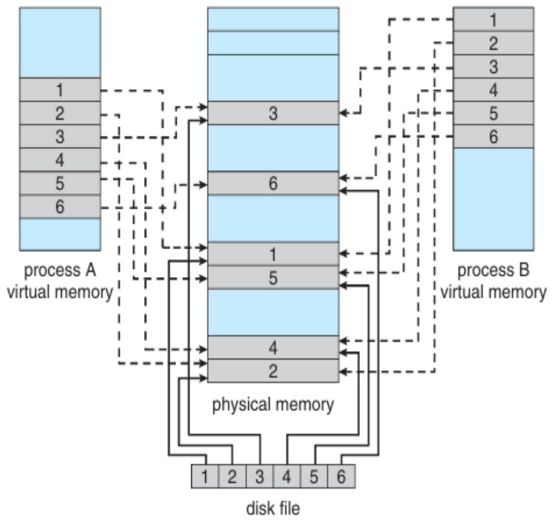
Beyond file access, memory mapping also plays a vital role in interprocess communication (IPC). Shared memory segments are often created by memory-mapping files into the address space of communicating processes. This technique provides an efficient way for processes to exchange information by reading and writing to a common memory region without the need for slower I/O operations.

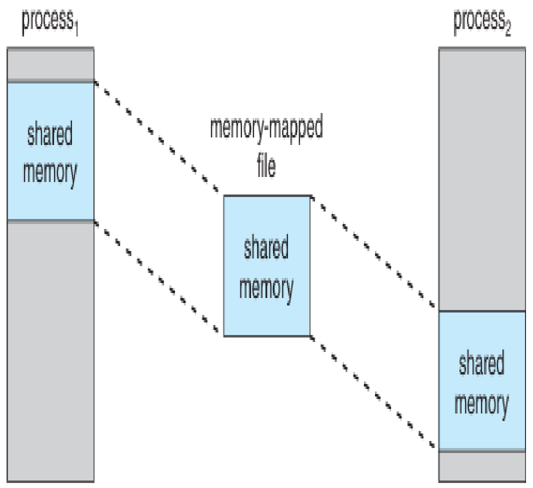
memory-mapped files streamline file access and enhance performance by reducing system call overhead and enabling direct memory manipulation. They also provide a foundation for efficient shared memory, facilitating fast and simple communication between processes. By integrating file I/O and memory management, memory mapping is a key feature in modern operating systems that supports both performance optimization and process collaboration.

Shared memory in the window API

The general outline for creating a region of shared memory using memory mappedfiles in the Windows API involves first creating a fil mapping for the file to be mapped and then establishing a view of the mapped file in a process's virtual address space. A second process can then open and create a view of the mapped file in its virtual address space. The mapped file represents the shared-memory object that will enable communication to take place between the processes

We next illustrate these steps in more detail. In this example, a producer process first creates a shared-memory object using the memory-mapping fea tures available in the Windows API. The producer then writes a message to shared memory. After that a consumer process opens a mapping to the shared memory object and reads the message written by the consumer





Memory mapped files

shared memory using memory mapped I/O

References

Operating system concepts tenth edition by ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE

Operating system concepts ninth edition by ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE

Modern operating system fourth edition by ANDREW S. TANENBAUM HERBERT BOS

internet