# FILE SYSTEM

- **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.
- The file system consists of two distinct parts: a collection of **files**, each storing related data, and a **directory structure**, which organizes and provides information about all the files in the system
- OS **abstracts from the physical properties of its storage devices** to define a **logical storage unit**, the file.
- Files are mapped by OS onto physical devices. These storage devices are usually **nonvolatile**
- Commonly, files represent **programs and data**. Data files may be numeric, alphabetic, alphanumeric, or binary.
- **File is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.**
- Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
- A **text file** is a sequence of characters organized into lines.

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

## FILE ATTRIBUTES

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

1. **Name**. The symbolic file name is the only information kept in human-readable form. It is the most important attribute used in all file operations
2. **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
3. **Type**. This information is needed for systems that support different types of files.
4. **Location**. This information is a pointer to a device and to the location of the file on that device.
5. **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
6. **Protection**. Access-control information determines who can do reading, writing, executing, and so on.

7. **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 (ADT) with six basic file operations.

1. **Creating a file**. Two steps
   - Space in the file system must be found for the file.
   - An entry for the new file must be made in the directory.
2. **Writing a file**.
   - Make a system call specifying both the name of the file and the information to be written to the file.
   - Given the name of the file, the system searches the directory to find the file's location.
   - The system must keep a write pointer to the location in the file where the next write is to take place.
   - The write pointer must be updated whenever a write occurs.
3. **Reading a file**.
   - Use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
   - 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 **current file position pointer.**
    - Both the read and write operations use this same pointer.
4. **Repositioning within a file**.
    - The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.
    - Repositioning within a file need not involve any actual I/0.
    - This file operation is also known as **file seek.**
5. **Deleting a file**.
    - 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.
6. **Truncating a file**.
    - The user may want to erase the contents of a file but keep its attributes.
    - Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged – except for file length-but lets the file be reset to length zero and its file space released.
- These six basic operations comprise the minimal set of required file operations.

- Other common operations include **appending** new information to the end of an existing file and **renaming** an existing file.
- **These primitive operations can then be combined to perform other file operations.**
- For eg, we can create a **copy** of a file, or copy the file to another I/O device, such as a printer or a display, by creating a new file and then reading from the old and writing to the new.
- We also want to have operations that allow a user to **get and set the various attributes** of a file.
- Most of the file operations involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an **open() system call** be made before a file is first used actively.
- OS keeps a small table, called the **open file table** containing information about all open files.
- When a file operation is requested, the file is specified via an **index into this table**, so no searching is required.
- When the file is no longer being actively used, it is **closed** by the process, and OS removes its entry from the open-file table.
- The open() operation takes a file name and searches the directory, copying the directory entry into the open-file table.

- The open() call can also accept **access mode information- create, read-only, read-write, append-only, and so on.** This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process.
- The open() system call typically **returns a pointer to the entry in the open-file table**. This pointer, not the actual file name, is used in all I/0 operations, avoiding any further searching
- In a multiprocessing system, **more than one process may be accessing the same file**.
- Typically, 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.
- The following information are associated with an open file.

1. **File pointer**. 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
2. **File-open count**. Multiple processes may have opened a file; the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes and reaches zero on the last close.

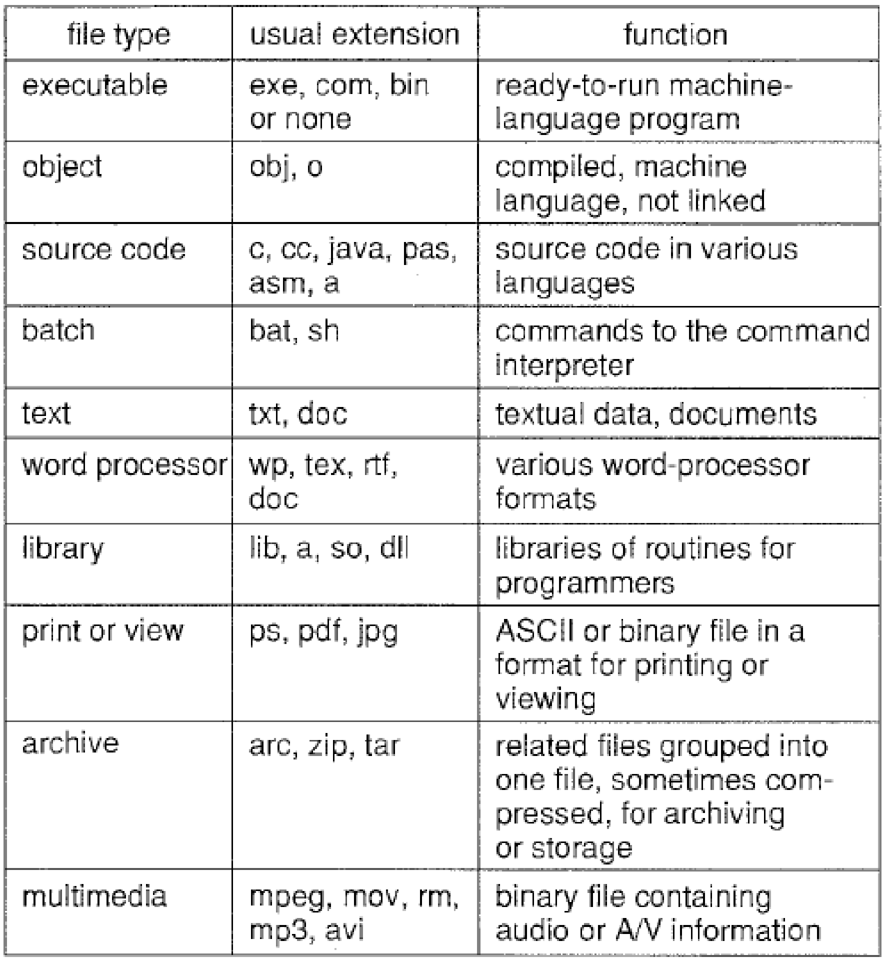
3. **Disk location of the file**. Most file operations require the system to modify data within the file. The information needed to locate the **file on disk is kept in memory** so that the system does not have to read it from disk for each operation.
4. **Access rights**. Each process opens a file in an access mode. This information is stored on the process table so the OS can allow or deny subsequent I/0 requests.

- Some OS provide facilities for **locking an open file**
- File locks allow one process to lock a file and prevent other processes from gaining access to it.
- 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.

## FILE TYPES

- Different types of files may be there.
- We always consider whether the OS should **recognize and support file types**. If OS recognizes the type of a file, it can then operate on the file in reasonable ways.
- A common technique for implementing file types is to include the **type as part of the file name**. The name is split into two parts- a name and an extension, usually **separated by a period character ("dot").**

- Examples: resume.doc, Server.java, and ReaderThread.c.
- 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 .bat extension can be executed, for instance. The .com and .exe files are two forms of binary executable files, whereas a .bat file is a **batch file** containing, in ASCII format, commands to OS
- For example, assemblers expect source files to have an .asm extension, and the Microsoft Word processor expects its files to end with a .doc extension.
- These extensions are **not mandatory**, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects.



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

**Figure 10.2** Common file types.

- The UNIX system uses a **magic number** stored at the beginning of some files to indicate roughly the type of the file like executable program, batch file, Script file, and so on.
- Not all files have magic numbers, so system features cannot be based solely on this information
- Extensions are meant mostly to **aid users** in determining what type of contents the file contains. Extensions can be used or ignored by a given application

## FILE STRUCTURE

- Each type of file has different structure. File types can be used to indicate the internal structure of the file.
- Eg: source and object files have structures that match the expectations of the programs that read them.
- Eg: OS requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
- **OS must support multiple file structures**. Eg: If OS defines five different file structures, it needs to contain the code to support these file structures.
- In addition, it may be necessary to define every file as one of the file types supported by OS. When new applications require information structured in ways not supported by OS, severe problems may result.
- Some OS impose (and support) a minimal number of file structures. This approach has been adopted in UNIX and MS-DOS
- This scheme provides maximum flexibility but little support.
- All OS must support at least one structure- that of an executable file- so that the system is able to load and run programs.
- **Locating an offset** within a file can be complicated for OS

- Because disk operations are performed in units of one block (physical record), and all blocks are the same size. It will not match with the exact logical block size (file units).
- **Packing** a number of logical records into physical blocks is a common solution to this problem.
- For example, the UNIX defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks say, 512 bytes per block-as necessary.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted.
- This is a case of **internal fragmentation.** Larger the block size, the greater the internal fragmentation.

## FILE ACCESS METHODS

- The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems support many access methods, and choosing the right one for a particular application is a major design problem.

## 1. <u>Sequential Access</u>

- The **simplest** access method is Sequential Access.
- Information in the file is processed **in order, one record after the other**.
- For **example, editors and compilers** usually access files in this fashion.
- A read operation - **read next**- reads the next portion of the file and automatically advances a file pointer. Similarly, the write operation-**write next**-appends to the end of the file and advances to the end of the newly written material
- Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward n records

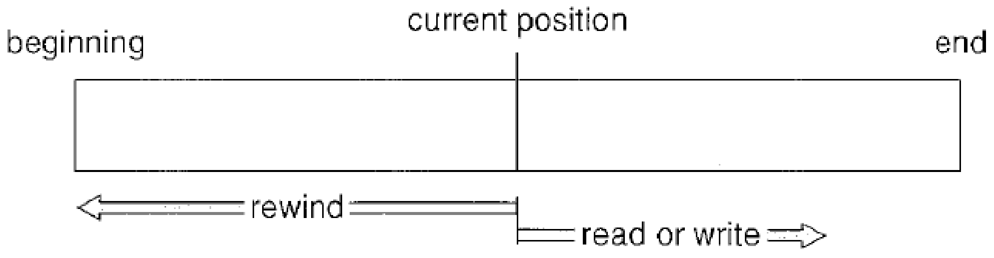


**Figure 10.3** Sequential-access file.

## 2. <u>Direct Access (Relative Access)</u>

- A file is made up of **fixed length logical records** that allow programs to **read and write records rapidly in no particular order.**
- The file is viewed as a numbered sequence of blocks or records.

- 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.
- **Databases are often of this type**. 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**, where n is the block number, rather than read next, and **write n** rather than write next.
- The block number provided by the user to the OS is normally a **relative block number**
- A relative block number is an index relative to the beginning of the file. Thus, **the first relative block of the file is 0, the next is 1, and so on**, even though the absolute disk address may be 14703 for the first block and 3192 for the second.
- We can easily simulate sequential access on a direct-access file by simply keeping a variable cp that defines our current position and adding a constant value 'position' with cp

  **cp = cp + position**

## 3. <u>Other Access Methods</u>

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

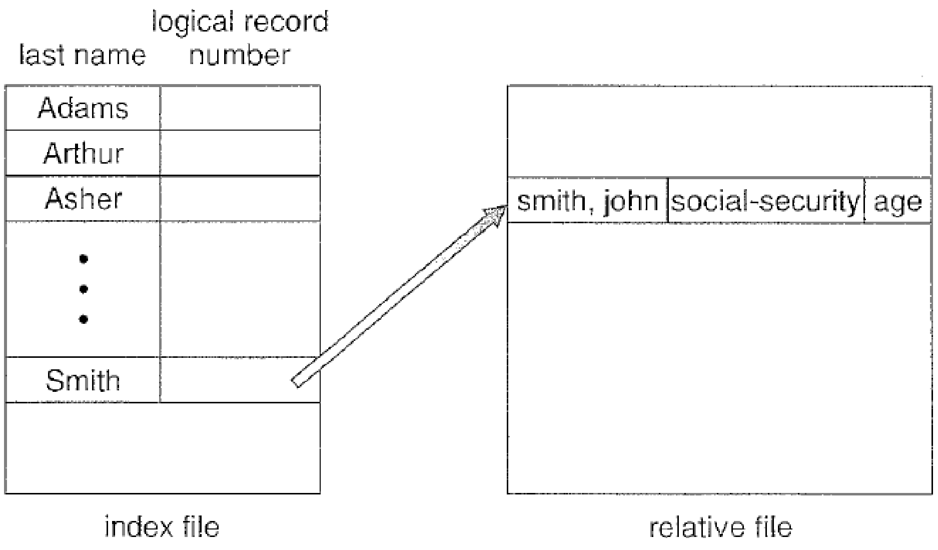


**Figure 10.5** Example of index and relative files.

- 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 would contain pointers to secondary index files, which would point to the actual data items.
- **Index searching can be implemented using binary search.**

## FILE SYSTEM PROTECTION

- When information is stored, we want to keep it safe from physical damage (the issue of **reliability**) and improper access (the issue of **protection**).
- File systems can be damaged by hardware problems, power surges or failures, head crashes, dirt, temperature extremes etc. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.
- Reliability is generally provided by **duplicate copies of files**.
- Many computers have systems programs that automatically copy disk files to tape at regular intervals to maintain a copy when a file system is accidentally destroyed.
- **Protection** can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet.
- In a multiuser system, however, other mechanisms are needed.

## Types of Access

- Protection mechanisms provide controlled access by limiting the types of file access that can be made.
- The following 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 new information 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.
- Other operations, such as renaming, copying, and editing the file, may also be controlled.

## Access Control

- Different users may need different types of access to a file or directory.
- The most general scheme to implement access control is to associate with each file and directory an **Access Control List (ACL).**
- ACL specifies user names and the types of access allowed for each user.
- When a user requests access to a particular file, OS checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.
- **The main problem with access lists is their length**. If we want to allow everyone to read a file, we must list all users with read access.

- To condense the length of the access-control list, many systems recognize **three classifications of users** in connection with each file:

1. **Owner**. The user who created the file is the owner.
2. **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
3. **Universe**. All other users in the system constitute the universe.

- For each of the above classes, 3 access permissions are defined.

1. **Read (r)**
2. **Write (w)**
3. **Execute (x)**

   Eg1: r w x r w x r w x : All 3 permissions for all the categories

   Eg2: r w x r – x – – x : Owner has all the permissions, Group can read and execute but cannot write, others can only execute

## 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:
    - The number of passwords that a user **needs to remember** may become large, making the scheme impractical.
    - If **only one password is used for all the files**, then once it is discovered, all files are accessible
- Most common scheme is to allow a user to associate **a password with a subdirectory**, rather than with an individual file, to deal with this problem.