

CHAPTER IX

FILE SYSTEMS

9.1 INTRODUCTION

- The file system is the part of the operating system that provides *long term storage* of information. Its first objective is therefore to store data in a permanent fashion in what is often called a *stable memory* :

RAM does not qualify because:

- *Dynamic RAM* loses its contents when powered off.
- *Static RAM* is too expensive,
- System crashes can corrupt the contents of the main memory.

Hence the use of magnetic recording: *disk drives* and tomorrow *Micro Electro-Mechanical Systems* (MEMS).

- The data managed by the file system are grouped in *user-defined* data sets called *files*. The file system must provide a mechanism for *naming* these data.

Each file system has its own set of conventions for naming files. All modern operating systems use a hierarchical directory structure. UNIX has pioneered the use of the file name space for naming other entities such as *devices* or *UNIX domain sockets*.

- A *good file system* should:
 - a) be easy to use, that is *simple* and *powerful*,
 - b) manage efficiently the *transfer of information* between the main memory and the stable memory
 - c) be capable to handle a *wide variety of file sizes*:

While most UNIX files are very small a few huge files take a large portion of the disk space (a few kilobytes at most).

9.2 FILE SYSTEM ORGANIZATION

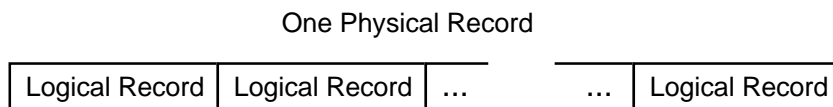
A. File Structures:

- Older file systems were organized into **records**, each representing a set of data to be read or written in a single I/O operation. The whole file appeared to its users as a *sequence of records*.

Most of the time these records were also reflecting the logical structure of the data. For instance a grade file would have consisted of several *student* records, each with a student name, a student ID, one field for each assignment or examination, one field for the semester average and one field for the final grade.

- There were two major problems with this approach:
 - a) It was very difficult to read a file if you did not know its structure, even more so if the file had been created in a different programming environment.
 - b) User-defined records were often too small: it is very likely that the student records of our previous example would contain less than 80 bytes each and reading in a file 80 bytes by 80 bytes is not very efficient.

With *blocking*, the programmer could define **physical records** grouping together several *logical records*:



Every time the first logical record of a physical record is read, the whole physical record could be brought into main memory. Instead of reading our student records one by one, we group them 10 by 10 and reduce by 90% the total number of disk accesses

- Some systems were even allowing users to define files organized into *trees of records*.

- UNIX pioneered a very different approach: a UNIX file is a *sequence of bytes* that is not interpreted by the file system:
 - Users are free to organize these bytes as they want but the logical records they will define are invisible to the file system.
 - The file system will divide the file into fixed-size physical records called **blocks** but these blocks are invisible to the users.

The three major advantages of this approach are its *generality*, its *flexibility* and its overall *good performance*. Selecting the "best" block size for a disk partition becomes the responsibility of the file system. This choice is normally the result of a compromise between the two contradictory goals of:

- a) Optimizing data transfers between the disk and the main memory: this implies selecting a large block size to reduce the number of disk accesses.
- b) Keeping internal fragmentation under control: if the average file size is 2 kilobytes, 8 kilobyte is not a good block size.

Berkeley UNIX solves the problem by allowing the allocation of *block fragments* to small files. MS-DOS has another problem. When the first hard disks for PC's were introduced, it was assumed that very few of these disks would ever be larger than 32 Megabytes. Hence MS-DOS assumed that there would never more than 64k blocks in a single disk partition. Until FAT/32, a 2-Gigabyte hard drive had a minimum block size of 32 kilobytes.

B. File Types:

- UNIX distinguish three broad classes of files:
 - a) **Ordinary files:**

These files can be *ASCII* or *binary* files; *executable* files are identified by a *magic number*.

b) Directories:

They are stored on disk like normal files with two important differences:

- Their contents can only be accessed through a predefined set of operations on directories
- Directory updates are immediately propagated to the disk instead of being delayed for up to a few seconds.

c) Special files:

Special files are *names* given to devices (`/dev/tty` represents the terminal on which the user is logged) or other system objects (`/dev/kmem` represents the kernel memory)

Any program expecting a file name as parameter can be passed instead a device name.

- The file system provides an excellent name space for *persistent objects*, sockets, semaphores, public mailboxes and so forth.

C. Access Methods:

- The most frequent mode of access is **sequential access** where the file contents are scanned from the beginning of the file. The file system must maintain an **index** pointing to the location of the current record or the current byte.
- **Direct access** allows the user to specify either which record should be accessed or at which byte offset the next operation should start
- Some other file systems, among which VMS, allow two additional access methods. They include:

1) Relative file organization:

Each file consists of a sequence of fixed-size numbered sequentially. Cells can either contain a record or be empty. The programmer can access any specific record by specifying its cell number acting as a relative record number.

2) Direct access with key:

The user can order the file system to search a given file for a record containing a given **key value**;

3) Sequential indexed:

When creating a sequential file, the user can ask the file system to create an **index** of the file for a specific key to speed up later searches.

9.3 FILE ATTRIBUTES

- File attributes—or *metadata*—consist of all the additional information maintained by the file system for each file
- **File protection** attribute may include:
 - the identity of the *file owner*,
 - a *password*: having the password gives access to the file; hence the password is a *ticket* to the file
 - an *access control list*, that is, a list of **who** can do **what**

Users	Access Rights
Alice	Read, Write
Bob	Read
Cindy	Append

The append right gives to the user the right to add data at the end of the file but not to clobber the existing file contents

UNIX uses a fixed size access control list where one can specify three classes of users (owner, all members of one of the groups specified in */etc/groups*, all users) and three access rights (read, write and execute)

- *Historical information* may include the **file creation time**, the **last time** the file was **accessed** and the last time it was **modified**:
- **Accounting information** will include the **file size**.
- The attributes of MS-DOS files include several *flags*, such as a *read-only* flag, a *hidden flag* (indicating that the file should not appear in directory listings) and an *archive flag* (indicating if the file has been modified since the last backup).

9.4 OPERATIONS ON FILES

- UNIX implements five basic operations:
 - `open()` returns a file descriptor,
 - `read()` reads a given number of bytes from the current location,
 - `write()` writes a given number of bytes from the current location,
 - `lseek()` moves the byte pointer to a new byte offset,
 - `close()` deletes the file descriptor.
- All reading and writing operations are sequential and start from the current position of the byte pointer
- Direct access is achieved by manipulating the offset through `lseek()`.
- UNIX maintains a file cache in main memory and buffers all its file I/O: This *delayed write* policy increases the I/O throughput but will result in lost writes whenever a process terminates in an abnormal fashion or the system crashes.
- Terminal I/O is also buffered one line at a time, which makes it difficult to read anything that is not terminated by a return.
- Access rights are always checked at *open time*: that is one of the reasons why the process opening the file must specify if it wants to read the file, to modify it or to do both. Once a process has successfully opened a file, that process will be able to access file even if the owner of the file changes the file access rights.

The file descriptor acts like a ticket granting access to the file.

- There are too many operations on file attributes to list them!

9.5 MAPPED FILES

- A mapped file is a file that is mapped into the virtual memory space of the processes accessing it.
Once a file is mapped, it becomes part of the virtual address space of the process.
- Two new system calls are introduced:
 - **map(pathname, mode)**, which returns the address of the first byte of the mapped file and replaces the UNIX **open(...)**,
 - **unmap(...)**, which undoes the mapping and replaces the UNIX **close(fd)**.
- Virtual memory is used to bring file *pages* into main memory: it must include *external pagers* so that these pages are read from the file and not from the process swap area.
- Access to the mapped file is done through standard programming constructs:
 - Reads and writes are now done through library functions.
 - There are much less system calls and therefore much less context switches.
- The four major disadvantages of mapped files are:
 - OS will never know the exact file size; it knows only the number of pages in the file.
 - If two or more processes on the same machine access one file at the same time, OS must map the file into a *shared memory segment*.
 - Shared access by two or more processes on different machines is even harder to implement
 - Some files are too large to be mapped into a 32-bit virtual address space.

9.6 DISK IMPLEMENTATIONS

- There are three basic approaches for allocating disk blocks to files:

a) Sequential Allocation:

- Each file occupies a set of *contiguous set of blocks* on the disk.
- The major advantages of this approach are its **simplicity** and the **low access times** it procures: we can even read several blocks in a single I/O operation.
- Its major disadvantages are *fragmentation* and the need to *predict* the *maximum size* of the file when disk space is allocated to the file.

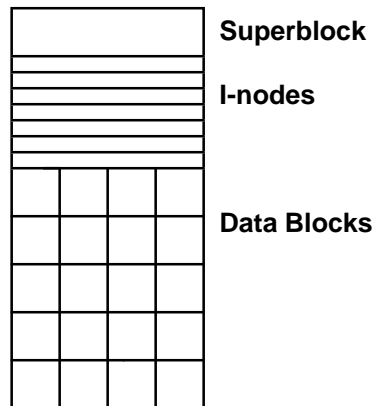
b) Linked Allocation:

- File blocks are scattered on the disk and each disk block contains the address of the next one.
- There is no external fragmentation.
- File growth is not a problem.
- *Sequential access* will be *slower* and *direct access impossible* to implement efficiently.

c) Indexed Allocation:

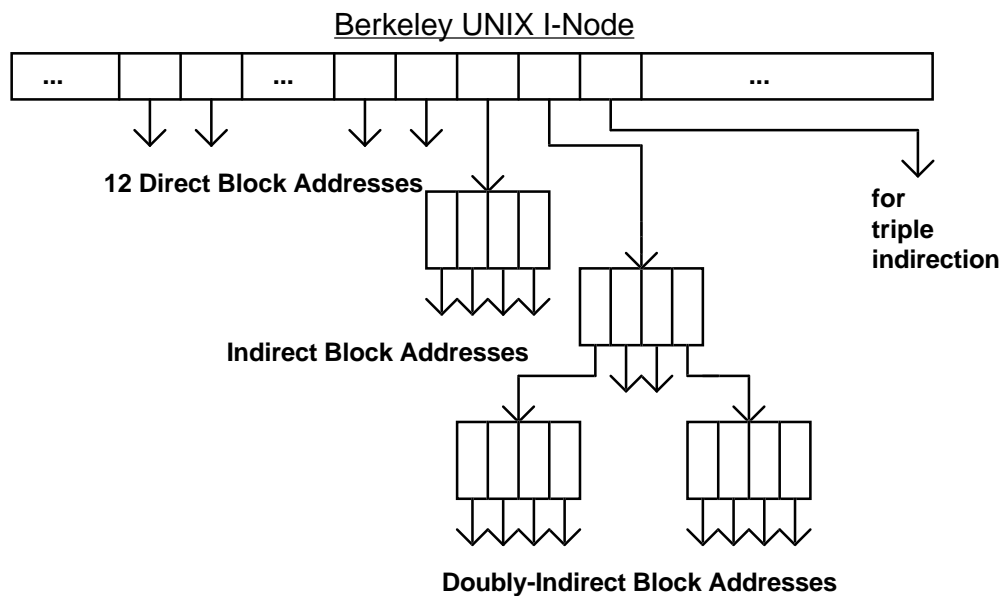
- File blocks are scattered on the disk and there is a *file index* containing the addresses of all the blocks in the file; the file index plays the role of a *page table*.
- To improve access times, we can **cache** the file index in main memory.
- Major problem is selecting the size of the file index: we can expect to have many small files and a few very big files.

• **The UNIX Solution:**



Each disk partition contains:

- 1) a **superblock** containing the parameters of the file system disk partition; Berkeley UNIX replicates it for more security;



- 2) an **i-list** with one **i-node** for each file or directory in the disk partition and a **free list**.

Each *i-node* contains:

- a). the *user-ID* and the *group-ID* of the owner of the file,
- b) the file protection bits,
- c) the file size,
- d) the times of file creation, last usage and last modification,
- e) the *number* of directory entries pointing to the file, and
- f) a flag indicating if the file is a directory, an ordinary file, or a special file.
- g) thirteen (System V) or fifteen (Berkeley) block addresses

The file name(s) can be found in the directory entries pointing to the i-node.

3) *data blocks:*

The original block size *b* was 512 bytes; it must now be at least 4 kilobytes for Berkeley UNIX.

In Berkeley UNIX, the 12 first block addresses in the *i-node* point at the 12 first blocks of the file. For larger files:

- a) the 13th block address points to an *indirect block* containing the addresses of $b/4$ additional blocks;
- b) the 14th block address points to a *double indirect block* containing the addresses of $b/4$ indirect blocks, each containing the addresses of $b/4$ data blocks;
- c) a 15th block address is not in use.

Hence a total of $12 + b/4 + b^2/16$ blocks can be addressed.

Since System V has only 13 block addresses, the first 10 blocks point at the first 10 blocks in the file. The 11th block points to an *indirect block* and the 12th block points to a *double indirect block*.

The 13th block is only used when the block size $b < 4,096$ and points to a *triple indirect block*.

UNIX always loads into memory the *i*-node of any opened file and caches all recently accessed blocks:

- first 10 or 12 blocks of a file can always be accessed directly,
- other blocks are likely not to require any extra disk access.

- **MS-DOS Solution:**

- MS-DOS and Windows 95 use a variant of the linked allocation method: each disk partition contains a file allocation table (FAT) having one entry per block.
- FAT entries are indexed by block numbers: so entry number 5 correspond to the fifth block in the disk partition.
- Each entry contain the block number of the *next block* in each file: the entry corresponding to the last block of a file thus contains an end-of-file marker:

3	2	4	6	5	EOF	7	8	9	EOF
---	---	---	---	---	-----	---	---	---	-----

- The above FAT represents a ten block partition where one file was allocated blocks 0, 3, 6, 7, 8, 9, 10 and another file blocks 1, 2, 4 and 5.
- The method is quite efficient as long as the whole FAT can be cached in main memory. MS-DOS avoided the problem by decreeing that no disk partition could contain more than 64K allocation units. The solution was quite acceptable as long as disk partitions did not exceed 128 or 256 MB. It leads to unacceptable internal fragmentation with today's very large disk partitions.

9.7 DIRECTORIES

- *UNIX Implementation:*

- UNIX_directories are just tables that map directory entries with numbers representing the index number (*i-number*) of the file in the *i-node* table of the disk partition.

Name	I-node
vi	203
edit	203
w	426
csH	173
...	...

- *They do not contain any other information!*
- Two directory entries can point to the same i-node; in that case the file will have two names.
- Directory subtrees cannot cross disk partition boundaries unless a new *file system* is **mounted** somewhere in the subtree.
- Only *non-directory* files may have more than one pathname; otherwise we could have loops in the directory structure.
- Berkeley UNIX has *symbolic links* that can bypass these restrictions: you can write:

```
ln -s /usr/bin/programs /bin/programs
```

even though `/usr/bin/programs` and `/bin/programs` are in two different partitions and might even be directories.

- Symbolic links point to another directory entry instead of the i-node.

- **MS-DOS Implementation:**
 - A MS-DOS directory entry contains:
 - a) the file *name*,
 - b) the *time* and *date* of its last modification,
 - c) the **READ-ONLY**, **HIDDEN** and **ARCHIVE** bits,
 - d) the file *size* in bytes,
 - e) the *address* of its first block.
 - The addresses of the other blocks can be found in the FAT of the disk partition.
 - MS-DOS has no feature allowing directory subtrees to cross disk partition boundaries.

9.8 KEY CONCEPTS

Be sure that you understand and can explain the following concepts:

direct access

directory

double indirect block

external pager

file access table (FAT)

file attribute

i-list

indirect block

indexed allocation

i-node

i-number

linked allocation

mapped file

metadata

sequential access

sequential allocationsuperblock

triple indirect block

UNIX special file