Systems Programming and Operating Systems	**Induprakas Kodukula**
CS414, Summer 1996	**Class#13, 7/29/96**

1 Class #10

The file system is typically the most visible aspect of an operating system. It provides a mechanism to users to "store" program as well as data permanently on secondary storage. It also provides the interface between the I/O operations requested by a program and the data that actually resides on a device. In particular, in UNIX, the file system is used to hide the various details of the individual I/O devices and present a uniform interface to the application program. A file system typically contains two classes of objects: **files** and **directories**. **Files** are named placeholders of data. A **directory** organizes and structures all the files in a system and provides information about all the individual files. A file is a sequence of records, where a **record** is the smallest allowable unit of secondary storage (sometimes also referred to as a **block**).

1.1 File System Fundamentals

The goals of a filesystem are the following:

- Present a logical or abstract view of files to the users by hiding the physical detail of the I/O devices.
- Facilitate sharing of I/O devices and optimize their uses.

The user's view of a file system is like an ADT on which specific operations can be performed. Typical opertions are the following:

- Creating a file
- Opening a file
- Closing a file
- Read/Write a file
- Reposition with in a file (called *lseek* in UNIX)
- Deleting a file
- Truncating a file - when the file contents are deleted, but ownership information etc remains the same. This could be faster than deleting a file and recreating a file with the old permissions.

A file has attributes - properties related to individual files:

- Name of a file - how does the user access the file?
- Type - is it a text file, binary file, executable file?
- Location - where is it stored on disk?
- Size - how many bytes is it?

Protection - who is allowed to access the file and in what form?

Time of important events - when last used, modified, etc.

Finally, a file has structure - which is to say that the data in a file is organized in a certain manner. Different operating systems store and operate files in different manners.

UNIX stores files as simply sequences of 8-bit bytes.

Macintosh expects executables to have two parts: a *resource fork* and a *data fork*. The resource fork consists of information of interest to the user such as the labels of the buttons displayed by a program. The *resource fork* is typically reconfigurable by the user. The *data fork* on the other hand, consists of program code and data that may not be changed by a user.

IBM's old operating systems used to support files as multiple units of records.

DEC's old VMS system used to support three differnt file structures. For example, executable files had different structure than text files. The advantage of this approach is that different file types can be told apart from one another. The disadvantage is that the code to handle different file types becomes cumbersome. It also becomes hard to add new file types to the system.

1.2 Directories

To organize file systems (which typically run into gigabytes), a file system is typically broken down into partitions. A partition is a single unit in the filesystem context. A partition could be smaller than a disk (several *volumes* on the same disk), or could span across several disks. Each partition has a *directory* that contains all the files residing on that partition. This way, a partition becomes the single unit of management. The directory for each partition is known as either the **device directory** or as a **volume directory**.

There is a fundamental dichotomy between the user's preferred access mechanism for files (names in alphabetical order). At the same time, for the file system it is convenient to access files by their physical location (which is typically some location on disk - a track, and a block on a track). So, it is necessary to maintain a mapping between the alphabetical names that people use for files and their actual locations on disk. There are several kinds of possible directory structures possible.

Single level directories: A directory is a single level when all the files are contained in a single directory. This type of structure is especially simple to implement to maintain. However, the problem with this approach is that it is quite hard to share files between users. For example, if two users have two files with the same name, then there would be a conflict with the directory entries corresponding to those names. Finally, even in the case of a single user, when the number of files increases dramatically, it is hard for the user to remember the names of all the files used so far to remember to create uniquely named files. This was the mechanism used in DOS machines (the FAT - file allocation table mechanism)

Two level directories The immediate problem of multiple users having potentially the same file name can be simply solved by using a two-level mechanism. Each user has a directory to keep track of all their own files. In addition, there is a top level directory which contains a pointer to the user-specific directory structure. For instance, in VMS (from DEC), a user file of name test.exe belonging to user y could be called u:[y]:test.exe. The two level directory mechanism makes isolation of two users from each other easier, but makes cooperation between two users harder. In addition, the problem of expanding file names for a single user is still a problem.

Tree directory scheme The two level directory scheme generalizes quite nicely to a tree directory mechanism. In this mechanism, the user directory contains entries which could themselves be directories, which could contain other directories and so on. This approach solves the problem that a particular user's number of files have to be large and different file names have to be unique. However, if anything, it makes the problem of sharing files even harder. File names which are absolute can be quite long. Relative path names can be used to access only the user's own files.

Acyclic graph system The acyclic graph file system is basically a tree based file system enhanced with **links**. A link file is a special file and is like a pointer. By creating a **link** to a file in another directory, one can avoid typing the (potentially large) name of the other file. UNIX provides the symbolic link (ln with -s as argument) utility to create symbolic links that point to any other file. If the file that a link file is pointing to gets deleted, the meaning of the link file is undefined.

2 File System Organization

File systems are sufficiently complex and massive and central to the operating system that they need to be organized carefully. To provide efficient and convenient access to the disk, a *file system* must be able to locate, store and retrieve data easily. For the following discussion, a file consists of logical blocks (numbered from say 0 to N) which are mapped onto actual physical blocks on disk.

A file system is composed of many levels as shown below.

```
 ---------------------------------
|                                 |
|       logical file system       |
|                                 |
 ---------------------------------
|                                 |
|     file-organization module    |
|                                 |
 ---------------------------------
|                                 |
|        basic file system        |
|                                 |
 ---------------------------------
|                                 |
|           I/O Control           |
|                                 |
 ---------------------------------
|                                 |
|             devices             |
|                                 |
 ---------------------------------
```

At the lowest level of the hierarchy lie the actual devices, such as disks, tapes, catridges and other devices.

At the second lowest level, I/O control consists of *device drivers* and interrupt handlers to do the actual transfer of data between the devices and the memory. A device driver provides an interface between the devices and the higher levels of the operating system. A device driver typically receives a high level command as input (e.g. fetch a particular block) and takes care of the device dependent details.

At the next level, the *basic file system* must issue the appropriate control instructions to the various device drivers it controls. The file system at this level can actually address the physical blocks that a file is mapped to.

The next level consists of the *file-organization* module. This module is aware of logical blocks of a file and is responsible for translating the logical block addresses of a file to the actual physical block addresses that need to be generated to do the actual access operations related to the file. Finally, this layer also manages the *free-space manager* which is responsible for allocating the free space on disk to files that are newly created or currently allocated files that become larger in size.

The final level is responsible for analyzing and maintaining the logical structure of the file system. For instance, if operations are performed on a symbolic link, it is the *logical* layer that is responsible for ensuring that the operations are performed on the actual file.

3 Files Access methods

Fundamentally, there are two kinds of file access methods. These are respectively *direct* and *sequential* access. Sequential access is usually the simplest kind of access. Information in the file is processed byte after byte (or record after record). Sequential read or write operations are usually associated with a *current* position in the file and operations are usually performed with respect to the current position. Typical applications using files in a *sequential* manner are editors and compilers.

The second access mechanism is called direct access, where there is a logical numbering of all the records in a file. Arbitrary blocks or bytes can be written in any order without any necessity of locality. This approach is especially desirable for very large files, such as databases, where transactions occur without any locality and a sequential access method would have grave access penalties.

4 Processes and Files

In an operating system, processes can access files using the mechanism discussed earlier: processes can open a file, close a file, read or write from specific locations etc. A system like UNIX maintains this information at two levels: a per-process file table and a system wide file table. Per-process file table consists of file specific information that changes for every process. For example, the current location being read from or written to in a file by a given process. The per-process file table contains an entry for a file opened for a particular process. The system wide file table contains an entry for all files open in the system. It contains process independent informaton such as how the logical blocks of a file are mapped into specific disk blocks, access information (such as who owns the file, what its privileges are, etc). The per process entry for a file contains a pointer to the corresponding entry in the system wide file table.

The *Berkeley File System* uses i-nodes (i stands for information) to store information about files. An i-node contains information about

ownership

file type (whether directory, link or ordinary file),

protection information,

number of directory entries pointing to it (useful for delete operations)

I-nodes are usually kept on a list. All files on a given device are stored as a list of i-nodes on that device.

5 Physical organization of files

Next, it is useful to study the physical organization of files, i.e. how files are actually allocated to disk blocks. There are several possibilities and each of them have different trade-offs.

Contiguous allocation: In this scheme all the logical blocks of a file are allocated to contiguous (adjacent) physical blocks. Only a pointer to the first block of a file and the number of blocks is stored in the *volume manager* and the file itself is stored at a block of contiguous addresses starting from the first block. This approach has the advanatage of simplicity and also the fact that the head of the disk need not move between access to adjacent blocks of the file. However, it has several disadvantages:

1. Finding space for a new file : after a while, the whole disk may become fragmented, i.e. there may be lots of free space on the disk, but no single space is large enough to accomodate the new file.
2. When starting a new file for writing, it may be impossible to know how large the file may become eventually. In that case, the file may simply outgrow the currently allocated space. It is very inefficient to move the file to a larger partition every time it outgrows the currently allocated chunk.
3. Pre allocation of a large chunk is inefficient usage of disk space if the file grows slowly.

Linked Allocation The second option to organize data is to have it in linked form. For example, every block of data on the disk belonging to some file would simply have a pointer to the next block of disk that belongs to the same file. The directory contains a pointer to the *first (for read)* and *last (for append)* block of a file. This approach solves the fragmentation problem, but introduces a bunch of new problems:

Direct access is out of the question.

Storage is wasted for pointers. This is not really an issue if the block size is quite large, but for very small block sizes, pointers would represent a significant portion of all the data stored.

Reliability is an issue: What if one of the pointers is corrupted? This problem can be alleviated somewhat by using doubly linked lists.

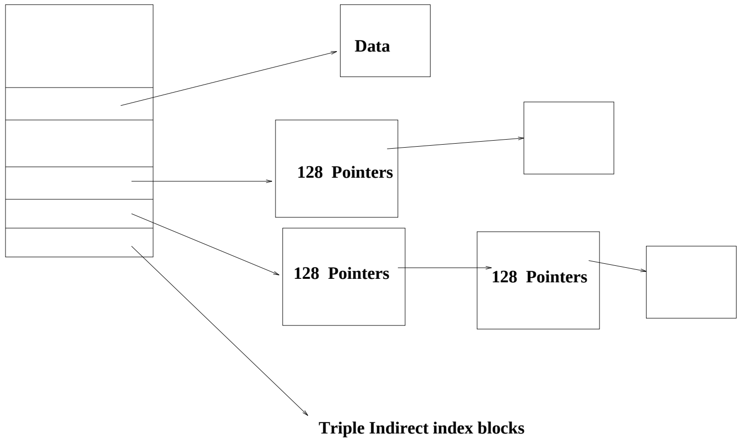

Figure 1: Combined multilevel indexing mechanism

The DOS and OS/2 file allocation tables (FAT) use this kind of mechanism. A directory entry to the file name points to the first block of the file. The first block contains the block number of the next block, which contains the block number of the next block and so on.

An example motivates the problems: Suppose a file is stored in blocks of 1K size each. To access the byte at position 32768 requires 33 disk reads. Clearly this is undesirable. A solution is to store the entire the entire FAT in memory. However, imagine that the disk has a storage of 64M, which is divided into blocks of size 1K each. There are 64K blocks, and each block can be represented by a 16 bit (2 byte) integer. To store the entire FAT in memory requires 64K*2 = 128K of memory, which is quite a large amount. The solution is to store in memory only the portion of the FAT which corresponds to the currently open files.

Indexed Allocation In this approach, each file has an index block which stores pointers to all the blocks of the file. For example, all index blocks have a fixed number of entries (say 12) and point to all the blocks of the file being referenced. The disadvantage of this approach is that index blocks are fixed in size and file sizes are not. This problem can be solved by using some pointers of the index block to point to other index blocks. UNIX solves this problem as follows: The index block has 12 entries. The first 9 entries are regular pointers to disk blocks containing file data. The 10th field contains a pointer to an index block which contains pointers only to data blocks. Essentially, the 10th entry in the initial index block represents a *one-level indirect* index. Similarly, the 11th entry points to index blocks, which themselves point to index blocks which point to data blocks (i.e. a two level indirect mapping) and the 12th pointer implements a three-level indirect mapping. This is shown in Figure 1.