A General-Purpose File System For Secondary Storage

R. C. Daley
Massachusetts Institute of Technology
Cambridge, Massachusetts
and
P. G. Neumann
Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

1. Introduction

The need for a versatile on-line secondary storage complex in a multiprogramming environment is immense. During on-line interaction, user owned off-line detachable storage media such as cards and tape become highly undesirable. On the other hand, if all users are to be able to retain as much information as they wish in machine-accessible secondary storage, various needs become crucial: Little-used information must percolate to devices with longer access times, to allow ample space on faster devices for more frequently used files. Furthermore, information must be easy to access when required, it must be safe from accidents and maliciousness, and it should be accessible to other users on an easily controllable basis when desired. Finally, any consideration which is not basic to a user's ability to manipulate this information should be invisible to him unless he specifies otherwise.

The basic formulation of a file system designed to meet these needs is presented here. This formulation provides the user with a simple means of addressing an essentially infinite amount of secondary storage in a machine-independent and device independent fashion. The basic structure of the file system is independent of machine considerations. Within a hierarchy of files, the user is aware only of symbolic addresses. All physical addressing of a multilevel complex of secondary storage devices is done by the file system, and is not seen by the user.

Section 2 of the paper presents the hierarchical structure of files, which permits flexible use of the system. This structure contains sufficient capabilities to assure versatility. A set of representative control features is presented. Typical commands to the file system are also indicated, but are not elaborated upon; although the existence of these commands is crucial, the actual details of their specific implementations may vary without affecting the design of the basic file structure and of the access control.

Section 3 discusses the file backup system, which makes secondary storage appear to the user as a single essentially infinite storage medium. The backup system also provides for salvage and catastrophe reload procedures in the event of machine or system failure. Finally, Section 4 presents a summary of the file system program modules and their interrelationship with one another. The modularity of design enables modules affecting secondary storage device usage to be altered without changing other modules. Similarly, the files are formatless at the level of the file system, so that any changes in format do not affect the basic structure of the file system. Machine independence is attempted wherever it is meaningful.

Sections 2 and 3 are essentially self-contained, and may be read independently of the companion papers (see references 1-5). Section 4 requires a knowledge of the first three papers.

2. The File Structure and Access Control

In this section of the paper, the logical organization of the file structure is presented. The file structure consists of a basic tree hierarchy of files, across which links may be added to facilitate simple access to files elsewhere in the hierarchy. Each file has an independent means for controlling the way in which it may be used.

If files are to be shared among various users in a way which can be flexibly controlled, various forms of safeguards are desirable. These include:

S1. Safety from someone masquerading as someone else;

S2. Safety from accidents or maliciousness by someone specifically permitted controlled access;

S3. Safety from accidents or maliciousness by someone specifically denied access;

S4. Safety from accidents self-inflicted;

S5. Total privacy, if needed, with access only by one user or a set of users;

S6. Safety from hardware or system software failures;

S7. Security of system safeguards themselves from tampering by non-authorized users;

S8. Safeguard against overzealous application of other safeguards.

These safeguards recur in the subsequent discussion. The various features of the file system presented below are summarized in Section 2.4, along with the way in which these features help to provide the above safeguards.

2.1 Basic Concepts

In the context of this paper, the word "user" refers to a person, or to a process, or possibly to a class of persons and/or processes. The concept of the user is rigorously defined in terms of a fixed number of **components**, such as an accounting number, a project number, and a name. (Classes of users may be defined by leaving certain components unspecified.) For present purposes, the only users considered are those who employ the file system by means of its normal calls.

A **file** is simply an ordered sequence of **elements**, where an element could be a machine word, a character, or a bit, depending upon the implementation. A user may create, modify or delete files only through the use of the file system. At the level of the file system, a file is formatless. All formatting is done by higher-level modules or by user-supplied programs, if desired. As far as a particular user is concerned, a file has one name, and that name is symbolic. (Symbolic names may be arbitrarily long, and may have syntax of their own. For example, they may consist of several parts, some of which are relevant to the nature of the file, e.g., ALPHA FAP DEBUG.) The user may reference an element in the file by specifying the symbolic file name and the linear index of the element within the file. By using higher-level modules, a user may also be able to reference suitably defined sequences of elements directly by context.

A **directory** is a special file which is maintained by the file system, and which contains a list of **entries**. To a user, an entry appears to be a file and is accessed in terms of its symbolic entry name, which is the user's file name. An **entry name** need be unique only within the directory in which it occurs. In reality, each entry is a pointer of one of two kinds. The entry may point directly to a file (which may itself be a directory) which is stored in secondary storage, or else it may point to another entry in the same or another directory. An entry which points directly to a file is called a **branch**, while an entry which points to another directory entry is called a **link**. Except for a pathological case mentioned below, a link always eventually points to a branch (although possibly via a chain of links to the branch), and thence to a file. Thus the link and the branch both **effectively point to** the file. (In general, a user will usually not need to know whether a given entry is a branch or a link, but he easily may find out.)

Each branch contains a description of the way in which it may be used and of the way in which it is being used. This description includes information such as the actual physical address of the file, the time this file was created or last modified, the time the file was last referred to, and access control information for the branch (see below). The description also includes the current state of the file (open for reading by N users, open for reading and writing by one user, open for data sharing by N users, or inactive), discussed in Section 4. Some of this information is unavailable to the user.

The only information associated with a link is the pointer to the entry to which it links. This pointer is specified in terms of a symbolic name which uniquely identifies the linked entry within the hierarchy. A link derives its access control information from the branch to which it effectively points.

2.2 The Hierarchy of the File Structure

The hierarchical file structure is discussed here. The discussion of access control features for selected privacy and controlled sharing are deferred until Section 2.3. For ease of understanding, the file structure may be thought of as a tree of files, some of which are directories. That is, with one exception, each file (e.g., each directory) finds itself directly pointed to by exactly one branch in exactly one directory. The exception is the root directory, or **root**, at the root of the tree. Although it is not explicitly pointed to from any directory, the root is implicitly pointed to by a fictitious branch which is known to the file system.

A file directly pointed to in some directory is **immediately inferior** to that directory (and the directory is **immediately superior** to the file). A file which is immediately inferior to a directory which is itself immediately inferior to a second directory **is inferior** to the second directory (and similarly the second directory is **superior** to the file). The root has level zero, and files immediately inferior to it have level one. By extension, inferiority (or superiority) is defined for any number of levels of separation via a chain of immediately inferior (superior) files. (The reader who is disturbed by the level numbers increasing with inferiority may pretend that level numbers have negative signs.) Links are then considered to be superimposed upon, but independent of, the tree structure. Note that the notions of inferiority and superiority are not concerned with links, but only with branches.

In a tree hierarchy of this kind, it seems desirable that a user be able to work in one or a few directories, rather than having to move about continually. It is thus natural for the hierarchy to be so arranged that users with similar interests can share common files and yet have private files when desired. At any one time, a user is considered to be operating in some one directory, called his **working directory**. He may access a file effectively pointed to by an entry in his working directory simply by specifying the entry name. More than one user may have the same working directory at one time.

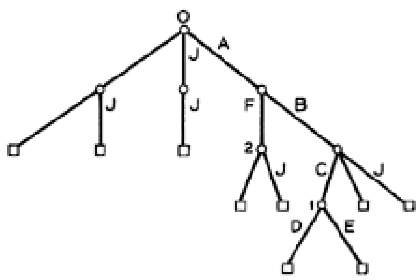

Figure 1. An example of a hierarchy without links.

An example of a simple tree hierarchy without links is shown in Fig. 1. Nonterminal nodes, which are shown as circles, indicate files which are directories, while the lines downward from each such node indicate the entries (i.e., branches) in the directory corresponding to that node. The terminal nodes, which are shown as squares, indicate files other than directories. Letters indicate entry names, while numbers are used for descriptive purposes only, to identify directories in the figure. For example, the letter "J" is the entry name of various entries in different directories in the figure, while the number "0" refers to the root.

An entry name is meaningful only with respect to the directory in which it occurs, and may or may not be unique outside of that directory. For various reasons, it is desirable to have a symbolic name which does uniquely define an entry in the hierarchy as a whole. Such a name is obtained relative to the root, and is called the **tree name**. It consists of the chain of entry names required to reach the entry via a chain of branches from the root. For example, the tree name of the directory corresponding to the node marked 1 in Fig. 1 is A:B:C, where a colon is used to separate entry names. (The two files with entry names D and E shown in this directory have tree names A: B: C: D and A: B: C: E, respectively.) In most cases, the user will not need to know the tree name of an entry.

Unless specifically stated otherwise, the tree name of a file is defined relative to the root. However, a file may also be named uniquely relative to an arbitrary directory, as follows. If a file X is inferior to a directory Y. the tree name of X relative to Y is the chain of entry names required to reach X from Y. If X is superior to Y. the tree name of X relative to Y consists of a chain of asterisks, one for each level of immediate superiority. (Note that since only the tree structure is being considered, each file other than the root has exactly one immediately superior file.) If the file is neither inferior nor superior to the directory, first find the directory Z with the maximum level which is superior to both X and Y. Then the tree name of X relative to Y consists of the tree name of Z relative to X (a chain of asterisks) followed by the tree name of Y relative to Z (a chain of entry names). For the example of Fig. 1, consider the two directories marked 1 and 2. The tree name of 1 relative to 2 is :*:B:C, while the tree name of 2 relative to 1 is :*:*:F. An initial colon is used to indicate a name which is relative to the working directory.

A link with an arbitrary name (LINKNAME) may be established to an entry in another directory by means of a command

```
LINK LINKNAME, PATHNAME.
```

(A command is merely a subroutine call.) The name of the entry to be linked to (PATHNAME) may be specified as a tree name relative to the working directory or to the root, or more generally as a path name (defined below). Note that a file may thus have different names to different users, depending on how it is accessed. A link serves as a shortcut to a branch somewhere else in the hierarchy, and gives the user the illusion that the link is actually a branch pointing directly to the desired file. Although the links add no basic capabilities to those already present within the tree structure of branches, they greatly facilitate the ease with which the file system may be used. Links also help to eliminate the need for duplicate copies of sharable files. The superimposing of links upon the tree structure of Fig. 1 is illustrated in Fig. 2. The dashed lines downward from a node show entries which are links to other entries. When the links are added to the tree structure, the result is a directed graph. (The direction is of course downward from each node.)

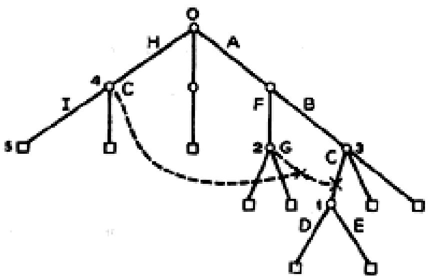

Figure 2. The example of Fig. 1 with links added.

In the example of Fig. 2, the entry named G in directory 2 is a link to the branch named C in directory 3. The entry named C in directory 4 (recall that entry names need not be unique except within a directory) is a link to the entry G in directory 2, and thus acts as a link to C in directory 3. Both of these links effectively point to the directory 1.

It is desirable to have a name analogous to the tree name which includes links. Such a name is the **path name**, and is assumed to be relative to the root unless specifically stated otherwise. The path name of a file (relative to the root) is the chain of entry names used to name the file, relative to the root. (For example, the directory 1 in Fig. 2 may have path name A:B:C;D, A:F:G or H:C, depending on its usage.) The working directory is always established in terms of a path name. A user may change his working directory by means of a command such as

```
CHANGEDlRECTORY PATHNAME,
```

where the path name may be relative to the (old) working directory or to the root. The definition of a path name relative to a directory other than the root is similar to the definition of a tree name, with the following exceptions: the concept of a file immediately inferior to a directory is replaced by the concept of a file effectively pointed to by the entry. The concept of a directory immediately superior to a file is replaced by a concept which is well defined only as the inverse of the above effective pointer, that is, dependent on what entry in which directory was previously used to reach the file.

In general, any file may be specified by a path name (which may in fact be a tree name, or an entry name) relative to the current working directory. A file may also be specified by a path name relative to the root. In the former case, the path name begins with a colon, in the latter case it does not.

To illustrate these somewhat elusive concepts, consider the example of Fig. 2. Suppose that the working directory has the path name H (i.e., directory 4). The command

```
CHANGEDIRECTORY :C
```

results in the working directory with path name H:C (i.e., directory 1). Subsequent reference to a file with path name : * :1 (relative to the working directory with path name H:C) refers to the file 5 in the figure. The command

```
CHANGEDIRECTORY :*
```

results in restoring the original working directory with path name H. (With this interpretation of *, the user believes he is working in a tree. Note that the design could be modified so that a path other than the one used on the way down could be used on the way back up toward the root, but not without adding considerable complexity to the design.)

The pathological case referred to above with respect to a link effectively pointing to a file arises as follows. Consider again Fig. 2. Suppose that the branch C in directory 3 is deleted from this directory; suppose also that in the same directory a link with name C is then established to the entry C in directory 4, e.g., by means of the command

```
LINK C,H:C.
```

Access to entry C in directory 3 (or to entry G in 2, or C in 4, for that matter) then results in a loop in which no branch is ever found. This and similar loops in which no branch is found may be broken in various ways, for example, by observing whether an entry is used twice on the same access. Note that much more devious loops may arise, as for example that resulting from the establishment of a link (named K) from directory 1 to entry H in the root. Then the path name :C:K relative to directory 4 refers to directory 4 itself. This and similar loops which involve chains of directories are inherent in the use of links, and may in fact be used constructively.

2.3 Access Control

An initial sign-on procedure is normally desirable in order to establish the identity of the user for accounting purposes. It may also be necessary to control the way in which the user may use the system. There are two basic approaches to using the hierarchy of files described here. First, the file structure may be **essentially open**, with initial access unrestricted and with subsequent access permitted to all other directories unless specifically denied. On the other hand, the file structure may be **essentially closed**, with initial access restricted for any user to a particular initial directory (assuming his ability to give a password, for example) and with subsequent access to other directories denied unless specifically permitted. There are in fact arguments for each extreme. The essentially open scheme implies that locks need be placed only where they are essential (and most effective). The essentially closed scheme provides well-defined working areas, frees the user from worrying about other users, and helps prevent the user's files from being accidentally altered. It may be observed that the scope of capabilities of the file structure described here does not depend on whether the structure is essentially open or closed. In practice, a position somewhere in between the two extremes is likely to result.

In attempting to access a file, a user may or may not be successful, depending upon what he is trying to do. The basic framework within which permissions are granted is now considered. This framework is independent of the file structure described above. Although the exact set of permissions may therefore vary from system to system, a flexible set adequate for normal usage is given here as an illustration. All permissions are logically on the branches which point to files. (In actual implementation, however, there may in some cases be permissions associated with a directory rather than repeated for each entry in that directory.)

The set of permissions with which a given user may access a particular branch is called the **mode** of the branch for that user. Associated with each branch is an **access control list**, which contains the list of users (or sets of users) along with the corresponding mode associated with each user. The permissions for any users on the list may be overridden (assuming permission to do so--see below) by adding subsequent users and modes to the list. The list is scanned in order of recency, and thus the addition acts as an override. (Each time the access control list is changed, a garbage collection is performed in order to keep the list nonredundant.) All access control information required for the use of a given file is contained in the list on the branch pointing to that file, and is thus independent of the way in which the file was accessed.

The mode consists of five **attributes**, named TRAP, READ, EXECUTE, WRITE and APPEND, each of which is either ON or OFF. In performing access control, the TRAP attribute is examined first. It is by itself powerful enough to accomplish the roles of the other four attributes, which are called **usage attributes**. However, the four usage attributes are included here for ease of description, as well as for ease of use of the system. The four usage attributes indicate permission to perform the given activity on the branch by the particular user only if the corresponding attribute is ON. The function of each attribute is now defined.

TRAP. When a branch has the TRAP attribute ON for a given user, a trap occurs on any reference by that user which affects the contents of the file to which that branch points. In this case, the access control module calls a procedure whose name is given as the first entry of a **trap list**. A trap list may be associated with each user in the access control list. Additional parameters may be defined in the trap list, and are passed as constants to the called procedure. Furthermore, all pertinent information regarding the branch as well as the calling sequence which caused the trap are passed to the called procedure. The traps are processed in the order specified by the trap list. The return to the access control module specifies the effective values of the four usage attributes which are to govern the access. The returned value may override the initial values of these attributes.

The user of a branch may inhibit the trap process. In this case, all references to an entry with the TRAP attribute ON cause an error return to the calling procedure. The TRAP attribute is extremely useful for monitoring of file usage, for placing additional restrictions on access (e.g., user-applied locks), for obtaining subroutines only if and when they are actually referenced, etc. A pair of commands such as LOCK and UNLOCK provide the user with a standard way of applying locks on an entry.

```
LOCK FILENAME,KEY

UNLOCK FILENAME
```

(FILENAME is the name of a branch given as a path name.) The command LOCK inserts a trap which on each attempted access may request the user to supply the designated key, and permit access only if the key is correctly supplied. UNLOCK removes the lock. (A timelock command might also be desirable, for example, to make a given branch available to a particular user only between certain times on certain days.) These commands are available to a user only if the branch pointing to the directory which contains the entry FILENAME has the WRITE attribute ON for that user (see below).

The Usage Attributes. The READ, EXECUTE, WRITE and APPEND attributes govern permission to perform operations upon files with certain intents, with an intent corresponding to each attribute. Every operation on a given branch implies one of the four intents, namely **read, execute, write or append**. The interpretation of the intent depends upon whether the accessed branch points to a directory (a **directory branch**) or to a nondirectory (a **nondirectory branch**), as seen below.

If a branch is a nondirectory branch, the meaning of each intent is quite simple. The read intent is the desire to read the contents of the file. The execute intent is the desire to execute the contents of the file as a procedure. The write intent is the desire to alter the contents of the file without adding to the end of it. The append intent is the desire to add to the end of the file without altering its original contents. The attribute on a nondirectory branch which corresponds to the particular intent of an operation on that branch indicates permission to carry out that operation only if that attribute is ON.

If a branch is a directory branch, the meaning of each intent is different. The read intent is the desire to read those contents of the directory which may be available to the user, i.e., to obtain an itemization of the directory entries. The execute intent is the desire to search the directory. The write intent is the desire to alter existing entries in the directory without adding new ones. This includes renaming entries, deleting entries, and changing the access control list for branches in that directory. The last of these includes adding traps to the trap list and changing the usage attributes. The append intent is the desire to add new entries without altering the original entries. The attribute on a directory branch which corresponds to the particular intent of an operation on that branch indicates permission to carry out that operation only if that attribute is ON.

Several additional examples of system commands are now given. Assuming the necessary WRITE attributes are ON for the appropriate directory branches, a user may by use of suitable commands change the access control list of entries or delete entries in various ways. For example, he may change (wherever permitted by the WRITE attribute of an inferior branch) the list for all inferior directory branches, or for all inferior nondirectory branches, or for all inferior nondirectory branches whose names include the parts FAP DEBUG, or for all directory branches not more than some number of levels inferior. Similarly, an elaborate delete command may be constructed. (The possibility of no one having the WRITE attribute ON for a given directory branch can be combated in various ways. One way is not to permit a change in the list to occur which brings about this circumstance, another way is to make this condition imply no restriction.)

Assuming that the necessary READ attributes are ON for the appropriate directory branches, a user may obtain an itemization of desired portions of desired inferior directories, possibly obtaining a graphical picture of the hierarchy.

2.4 Summary of File System Features

At this point, it is desirable to summarize the various features of the file system, and to state which of these contribute to which of the safeguards S1 through S8 mentioned above. The basic features of the file system may be stated as follows:

F1. The inherent hierarchical structure of the file system itself;

F2. The access control which may be associated with a directory branch;

F3. The backup procedures (discussed in the next section).

In addition, certain aspects of the hardware and of the central software also contribute to providing these safeguards:

F4. The hardware, and central software system.[2,3]

The ways in which the safeguards S1-S8 interact with the features F1-F4 are summarized in Table 1.

Table 1. The Features of the File System and Which Safeguards They Assist in Providing.

Safeguards / Features	MASQ S1	PERM S2	DENY S3	SELF S4	PRIV S5	BUGS S6	TAMP S7	ZEAL S8
F1. Hierarchy	Y		Y	Y	Y	Y		
F2. Attributes	Y	Y	Y	Y	Y	Y	Y	Y
F3. Backup	Y	Y		Y		Y		
F4. System			Y		Y	Y	Y	

Note: Y = YES, the feature does assist. Blank = NO, it does not.

3. Secondary Storage Backup and Retrieval

One important aspect of the file system is that the user is given the illusion that the capacity of file storage is infinite. This concept is felt to be extremely important, as it gives all responsibility for remembering files to the system rather than to the individual user. Many computer installations already find themselves in the business of providing tape and card-file storage for their users. It is intended that most of this need will be replaced by the file system in a more general and orderly manner.

That portion of the file system storage complex which is immediately accessible to the file system, i.e. disks and drums, is called the **on-line storage system**. Devices which are removable from the storage complex, such as tapes, data cells and disk packs which are used by the file system as an extension of the on-line facilities, are called the **file backup storage system**. To the user, all files appear to be on line, although access to some files may be somewhat delayed. For the purpose of discussion, a backup system consisting only of magnetic tape is considered. However, the system presented here is readily adaptable to other devices.

Incremental Dumping of New Files

Whenever a user signs off, additional copies of all files created or modified by that user are made in duplicate on a pair of magnetic tapes. At the end of every N hour period, any newly created or modified files which have not previously been dumped are also copied to these tapes. When this is done, the tapes are removed from the machine and replaced by a fresh set of tapes for the next N hour period. Typically N would be a period of between 2 and 4 hours. This procedure has the advantage that the effects of the most catastrophic machine or system failure can be confined to the N hour dumping period.

Weekly Dumping of Frequently Used Files

In the event of a catastrophe, the on-line storage system could be reloaded from these incremental dump tapes. However, since many valuable files, including system programs, may not have been modified for a year or over, this method of reloading is far too impractical. In order to minimize the time necessary to recover after a catastrophe, a weekly dump is prepared of all files which have been used within the last M weeks. This dump is also made on duplicate tapes for reliability.

Actually this weekly dump is taken in two parts. The first part consists of all files which must be present in order to start and run the basic system. The second part consists of all other files which have been used within the last M week period. Typically M would be a period of about three to five weeks. The weekly dump tapes may be released for other use after a period of about two or three months. The incremental dump tapes must be kept indefinitely. However, it may be advantageous to consolidate these tapes periodically by deleting obsolete files.

Catastrophe Reload Procedure

Should a catastrophe occur in which the entire contents of the on-line storage system is lost, the following reload procedure can be used. First reload a copy of the system files from the most recent weekly dump tapes. When this has been done the system may be started, with the rest of the reloading process continuing under the control of the system. Note that this does not necessarily represent the most recent copy of the system. If an important system change has been made since the weekly dump was taken, it may be necessary to reload the incremental tapes before starting the system.

After the system files are reloaded, the incremental dump tapes are reloaded in reverse chronological order, starting with the most recent set of incremental tapes. This process is continued until the time of the last weekly dump is reached. At this time, the second part of the weekly dump tapes is reloaded. During this process all redundant or obsolete files are ignored. The date and time a file is created or last modified is used to insure that only the most recent copy of a file remains in the on-line storage system. Since directories are dumped and reloaded in the same manner as ordinary files, the contents of the on-line storage system can be accurately restored.

It is possible to continue to load the older weekly tapes until the on-line system is totally reloaded. However, the amount of new information picked up from these tapes becomes increasingly small as one goes further back in time. In view of this, files which do not appear on the most recent set of weekly dump tapes, due to inactivity, are not reloaded at this time. Instead, a trap is added to the Appropriate directory branch so that a retrieval procedure is called when the file is first referenced. This allows these files to be reloaded as needed by the retrieval mechanism which is discussed later in this paper.

On-Line Storage Salvage Procedure

Although the catastrophe reload procedure can accurately reconstruct the contents of the on-line storage system, it is normally used only as a backstop against the most catastrophic of machine or system failures. When the milder and more common failures occur, it is often possible to salvage the contents of secondary storage without having to resort to the reload procedure. If this can be done, many files which have been created or modified since the end of the last incremental dump period can be saved. In addition, much of the time necessary to run the reload procedure can also be saved.

The usual result of a machine or system failure is that the contents of secondary storage are left in a state which is inconsistent. For example, two completely unrelated directory entries may end up pointing to the same physical location in secondary storage, while the storage assignment tables indicate that this area of storage is unused. If the system were restarted at this time, the situation might never be resolved. The usual effect is that any information subsequently assigned to that area of secondary storage is likely to be overwritten.

This situation arises when the system goes down before the file system has updated its assignment tables and directories on secondary storage. What has probably happened is that some user has deleted a file and another user created a new file which was assigned to the area of storage just vacated by the previous file. When the system goes down, the changes have not been recorded in secondary storage. This is only one example of the type of trouble which occurs when the system fails unexpectedly.

The salvage procedure is designed to read through all the directories in the hierarchy and correct inconsistent information wherever possible. The remaining erroneous files and directory entries are deleted or truncated at the point at which the error was found. Storage assignment tables are corrected so that only one branch points to the same area of secondary storage. Since it is necessary to read only the directories and the storage assignment tables, the salvage procedure can be run in a small percentage of the time necessary to run a complete reload procedure.

The salvage procedure also serves as a useful diagnostic tool, since it provides a printout of every error found and the action taken. This program can also be run in a mode in which it only detects errors but does not try to correct them.

Retrieval of Files from Backup Storage

Unless a file has been explicitly deleted by a user, the directory entry for that file remains in the file system indefinitely. If, for some reason, the file associated with this entry does not currently reside on an on-line storage device, the corresponding branch for that file contains a trap to a file retrieval procedure. When a user references a file which is in this condition, his process traps to the retrieval procedure. At this time the user may elect to wait until the file is retrieved from the backup system, to request that the file be retrieved while he works on something else, to abort the process that requested the files or to delete the directory entry.

If the user elects to retrieve the file, the date and time the file was created or last modified (which are available from the directory entry) are used to select the correct set of incremental dump tapes. The retrieval procedure requests the tape operator to find and mount these tapes. These tapes are then searched until the precise copy of the requested file is found and reloaded. At this time the original access control list of the branch is restored, and the file is now ready to be used by the user.

If a user deletes a file, both the file and the corresponding directory entry are deleted. However, if a copy of this file appears on a set of incremental dump tapes, this copy is not deleted at this time. This file can still be retrieved if the user specifies the approximate date and time when the file was created or last modified. To help the user in this situation, the incremental dump procedure provides a listing for the operations staff of the contents of each set of incremental tapes. These listings are kept in a log book which may be consulted by the operators in situations such as the above. Selected portions of this listing may be made available to the user.

The user is able to declare that he wishes a certain file to be removed from the on-line storage system without deleting the corresponding directory entry. This may be accomplished by using a system procedure which places the file in a state where it can be retrieved by the normal retrieval procedure.

General Reliability

Since the file system is designed to provide the principal information storage facility for all users of the system, the full responsibility for all considerations of reliability rests with the file system. For this reason all dumping, retrieval and reloading procedures use duplicate sets of tapes. These tapes are formatted in such a manner as to minimize the possibility of unrecoverable error conditions. When reading from these tapes during a reload or retrieval process, multiple errors on both sets of tapes can be corrected as long as the errors do not occur in the same physical record of both tapes. If an error occurs which cannot be corrected, only the information which was in error is lost. If the error is a simple parity error, the information is accepted as if no error occurred. When a user first attempts to use a file in which a parity or other error was found, he is notified of this condition through a system procedure using the trap mechanism.

Secondary Storage Allotments

The file system assigns all secondary storage dynamically as needed. In general, no areas of the on-line storage system are permanently assigned to a user. A user may keep an essentially infinite amount of information within the file system. However, it is necessary to control the amount of information which can be kept in the on-line storage system at a time.

When a user first signs on, the file system is given an account name or number.

All files subsequently created by this user are labeled with his account name. When the user wishes to increase his usage of secondary storage, the file system calls upon a secondary storage accounting procedure giving the user's account name and the amount and class of storage requested.

The accounting procedure maintains records of all secondary storage usage and allotments. A storage allotment is defined as the amount of information which a particular account is allowed to keep in the on-line storage system at one time. Normally the accounting procedure allows a process to exceed the allotment after informing the file system that the account is overdrawn. However, the accounting procedure may decide to interrupt the user's process if the amount of online storage already used seems unreasonable.

Multilevel Nature of Secondary Storage

In most cases a user does not need to know how or where a file is stored by the file system. A user's primary concern is that the file be readily available to him when he needs it. In general, only the file system knows on which device a file resides.

The file system is designed to accommodate any configuration of secondary storage devices. These devices may cover a wide range of speeds and capacities. All considerations of speed and efficiency of storage devices are left to the file system. Thus all user programs and all other system programs are independent of the particular configuration of secondary storage.

All permanent secondary storage devices are assigned a level number according to the relative speed of the device. The devices which have the highest transmission and access rates are assigned the highest level numbers. As files become active, they are automatically moved to the highest-level storage device available. This process is tempered by considerations such as the size of the file and the frequency of use.

As more space is needed on a particular storage device, the least active files are moved to a lower-level storage device. Files which belong to overdrawn accounts are moved first. Files continue to be moved to lower-level storage until the desired amount of higher-level storage is freed. If a file must be moved from the lowest-level on-line storage device, the file is removed and the branch for this file is set to trap to the retrieval procedure.

4. File System Program Structure

This section describes the basic program structure of the file system presented in the preceding sections, as implemented in the Multics system.[1] (It is assumed here that the reader is familiar with the papers referred to in references 1, 2 and 3.)

A user may reference data elements in a file explicitly through read and write statements, or implicitly by means of segment addressing. It should be noted here that the word "file" is not being used in the traditional sense (i.e., to specify any input or output device). In the Multics system a file is a linear array of data which is referenced by means of a symbolic name or segment number and a linear index. In general, a user will not know how or on what device a file is stored.

A Multics file is a segment, and all segments are files.[1,3] Although a file may sometimes be referenced as an input or an output device, only a file can be referenced through segment addressing. For example, a tape or a teletype cannot be referenced as a segment, and therefore cannot be regarded as a file by this definition.

Input or output requests which are directed to I/O devices other than files (i.e. tapes, teletypes, printers, card readers, etc.) will be processed directly by a Device Interface Module (see reference 4) which is designed to handle I/O requests for that device. However, I/O requests which are directed to a file will be processed by a special procedure known as the File System Interface Module (see reference 4). This module acts as a device interface module for files within the file system. Unlike other device interface modules, this procedure does not explicitly issue I/O requests. Instead, the file system interface module accomplishes its I/O implicitly by means of segment addressing and by issuing declarative calls to the basic file system indicating how certain areas of a segment are to be overlayed.

4.1 The Basic File System

Whether a user refers to a file through the use of read and write statements or by means of segment addressing, ultimately a segment must be made available to his process. The basic file system may now be defined as that part of the central software

which manages segments. In general this package performs the following basic functions.

1. Maintain directories of existing segments (files)
2. Make segments available to a process upon request.
3. Create new segments.
4. Delete existing segments.

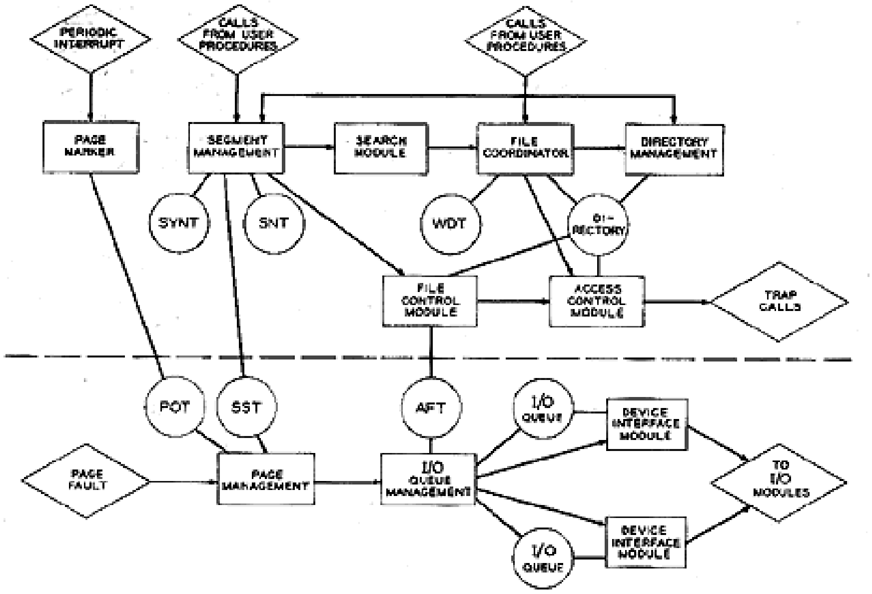

Figure 3. The basic file system.

Figure 3 is a rough block diagram of the modules which make up the basic file system. This diagram is by no means complete but is used here to give the reader an overall view of the basic flow. The directional lines indicate the flow of control through the use of formal calling sequences, with formal return implied. Lines with double arrowheads are used to indicate possible flow of control in either direction. The circles in the diagram indicate some of the data bases which are common to the modules indicated. The modules and data bases drawn below the dotted line must at least partially reside in core memory at all times since they will be invoked during a missing-page fault (see reference 3) .

Segment Management Module

The segment management module maintains records of all segments which are known to the current process. A segment is **known to a process** once a segment number has been assigned to that segment for this process. A segment which is known to a process is **active** if the page table for that segment is currently in core. If the page table is not currently in core, that segment is **inactive.**

If a segment is known to a process, an entry will exist for that segment in the Segment Name Table (SNT). This entry contains the call name, the tree name and the segment number of the segment (file) along with other information pertinent to the segment as used by this process. The **call name** is a symbolic name used by the user to reference a segment. This name normally corresponds to an entry in the user's directory hierarchy which effectively points to the desired file. It should be noted that a different copy of the segment name table exists for each individual process.

If a segment is active, an entry for that segment exists in the Segment Status Table (SST). This table is common to all processes and contains an entry for each active segment. If a segment is inactive (no page table is in core), no entry exists for that segment in this table. Each entry in the segment status table contains information such as the number of processes to which this segment is known and a pointer which may be used to reference the file or files which are to receive all I/O resulting from paging this segment in and out of core.[3] When a user references a segment for the first time, a directed fault will occur. At this time control is passed to a procedure known as the linker.[3] This procedure picks up the symbolic segment call name from a pointer contained in the machine word causing the fault. The linker must now establish a segment number from this symbolic name. An entry to the segment management module is provided for precisely this purpose. When a call is made to the segment management module to establish a segment number from a call name, the segment name table is searched for that call name. If the call name is found in the segment name table, the segment number from this table is returned immediately to the calling procedure. However, if this is not the case, the segment management module must take the following steps.

1. Locate the segment (file) in the user's directory hierarchy via a call to the search module.
2. Assign a segment number for this segment.
3. Update the segment name table indicating that this segment is now known to this process.
4. Open the file or files which are to receive I/O resulting from paging.
5. Create or update the appropriate entry in the segment status table.
6. Establish a page table and segment descriptor for this segment if the segment was not already active for some other process.
7. Return the segment number to the calling procedure.

If a segment is known to a process but is not currently active, the descriptor for that segment will indicate a fault condition. If and when this fault occurs, the segment can be reactivated by locating the approve private entry in the segment name table and repeating steps 4 through 7. Note that the segment does not have to be located again in the directory hierarchy since the tree name is retained in the segment name table.

If a segment is to be modified during its use in a process, the user may elect to modify a copy of that segment rather than the original. When this is the case, the copying of this segment is done dynamically as a by-product of paging. However, if the copying is not complete at the time the segment becomes inactive, the copying must be completed at this time.

If a segment is to be copied, there are actually two open files involved, the original file and the copy or **execution file**. When a page table is initially constructed by the segment management module, each entry in that page table will contain a fault indication and a flag indicating what action should be taken if and when that fault occurs. This flag may indicate one of the following actions:

1. Assign a blank page.
2. Retrieve the missing page from the original file.
3. Retrieve the missing page from the execution file.

Once a page has been paged out (written) into the execution file, it must be retrieved from that file.

An entry to the segment management module is provided by which a user may declare a synonym or list of synonyms for a segment name. For example, a user may have a certain procedure which references a segment called "Gamma" and another procedure which references a segment called "Alpha." If the user wishes to operate both procedures as part of the same process using a segment called "Data" he may do so by declaring Alpha and Gamma to be synonyms for Data. This association is kept by the segment management module in a Synonym Table (SYNT). Whenever the segment management module is presented with a call name which has been defined as a synonym, the appropriate name is substituted before any further processing takes place.

In addition to the functions described above, the segment management module provides entries through which the user may ask questions or make declarations involving the use of segments known to his process. Some of these functions are listed below.

1. Declare that a segment or some specific locations within a segment are no longer needed at this time.
2. Declare that a segment or some specific locations within a segment are to be reassigned rather than paged in as needed. (The user is about to overwrite these locations.)
3. Ask if a segment or some specific locations within a segment are currently in core.
4. Declare that a certain segment is to be created when first referenced.
5. Terminate a segment, indicating that this segment is no longer to be considered as known to this process.

Search Module

The search module is called by the segment management module to find a particular segment (file) in the user's directory hierarchy. The search module directs the search of individual directories in the user's hierarchy in a predetermined pattern until the requested branch is found or the algorithm is exhausted. This module calls the file coordinator to search particular directories and to move to other directories in the hierarchy. The user is able to override this search procedure by providing his own search procedure at the initiation or during the execution of his process.

The File Coordinator

The file coordinator provides all the basic tools for manipulating entries within the user's current working directory. The functions provided by this module perform only the most primitive operations and are usually augmented by more elaborate system library procedures. The following is a list (of some of these operations.

1. Create a new directory entry.
2. Delete an existing entry.
3. Rename an entry.
4. Return status information concerning a particular entry.
5. Change the access control list for a particular branch.
6. Change working directory.

Whenever a user wishes to perform any operation through the use of the file coordinator, the access control module is consulted to determine if the operation is to be permitted.

Since most calls to the file coordinator refer to entries contained in the user's working directory, the file coordinator must maintain a pointer to this directory. This is done by keeping the tree name of the working directory in a Working Directory Table (WDT) for this process.

Directory Management Module

When the file coordinator wishes to search the user's working directory, the actual search is accomplished by use of the directory management module. This module searches a single directory specified by a tree name for a particular entry or group of entries. The actual directory search is confined to this module to isolate the recursion process which may be required to search a given directory.

The directory management module issues calls to the segment management module to obtain a segment number for the directory for which it has only a tree name. When the directory management module obtains this segment number and references the directory by means of segment addressing, a descriptor fault may occur indicating that this segment is no longer active. If this happens, the segment management module will try to reactivate this segment by attempting to find this directory in the next superior directory by means of the tree name in the segment name table. To do this the segment management module issues a direct call to the directory management module to search the next superior directory for the missing directory. After obtaining a segment number for the superior directory, the directory management module may cause another descriptor fault to occur when attempting to search this directory. This process may continue until a directory is found to be an active segment or until the root of the directory hierarchy is reached. Since the root is always known to the directory management module, the depth of recursion is finite.

File Control Module

The file control module is provided to open and close files for the segment management module. A file is said to be open, or **active**, if it has a corresponding entry in the Active File Table (AFT). If a file is active, the corresponding entry in the active file table provides sufficient information to control subsequent I/O requests for that file.

If the file is inactive, the open procedure needs only to open the file to the requested state and make the corresponding entry in the active file table. If the file is active, it may have N users reading, or 1 user reading and writing, or N users data sharing (using file as a common data base). If the requested state is incompatible with the current state of the file, the current process must be blocked.[3] For example, if the current user wishes to read a stable copy of the file and there is currently a user writing into that file, the requested state (reading) and the current state (reading and writing) are said to be incompatible.

If the requested state and the current state of the file are found to be compatible, the number of users using the file in that state is increased by one. When a file has been successfully opened by the file control module (with the permission of the access control module), the pointer to the corresponding entry in the active file table is returned to the calling procedure. This pointer is used to direct requests for subsequent input or output to the correct file.

Access Control Module

The access control module is called to evaluate the access control information for a particular branch, as defined in Section 2. This module is given a pointer to the directory entry for the branch in question and a code indicating the type of operation which is being attempted. The access control module returns a single effective mode to the calling procedure. The effective mode is the mode which governs the use of a file with respect to the current user or process. The calling procedure uses this mode to determine if the requested operation is to be permitted.

If the access control information indicates that a trap is to be effected, the procedure to which the trap is directed is passed the entry for the branch in question and the operation code. The procedure which processes the trap must return to the access control module, specifying the effective mode to be returned by the access control module to its calling procedure. The procedure which processes the trap may choose to strengthen, weaken or leave unchanged the usage attributes which define the effective mode for the branch.

Page Marker Module

The page marker periodically interrupts the current process and takes note of page usage, and resets the page use bits[2] of all pages involved in the current process. Pages which fall below a dynamically set activity threshold are listed in the Page Out Table (POT) as likely candidates for removal when space becomes needed.

Page Management Module

Control passes to the page management module by means of a missing-page fault in a page table in use by the current process. This fault may indicate that a new page should be assigned from free storage or that an existing page should be retrieved from an active file. In either case a free page must be assigned before anything else can happen. If no pages are currently available, the first page listed in the page out table is paged out. If no pages are listed in the page out table, a random page of appropriate size is removed.

If a new page is to be read in, the page table entry for the missing page contains a pointer to the appropriate entry in the segment status table and a flag indicating whether this page is to be read from the original file or the execution file. In either case a pointer to the appropriate active file may be obtained from the segment status table. This pointer is passed as a parameter to the I/ O queue management module with a read request to restore the correct page to core memory.

I/O Queue Management Module

The I/O queue management module processes input and output requests for a particular active file. The calling procedure specifies a read or a write request and a pointer to an entry in the active file table which corresponds to the desired file. This request is placed on the appropriate queue for the particular device interface module which will process the request. The queue management module then calls that device interface module indicating that a new request has been placed on its queue. When this is done, the queue management module returns to the calling procedure which must decide whether or not to block itself until the I/O request or requests are completed.

Device Interface Modules

For each type of secondary storage device used by the basic file system, a device interface module will be provided. A device interface module has the sole responsibility for the strategy to be used in dealing with the particular device for which it was written. Any special considerations pertaining to a particular storage device are invisible to all modules except the interface module for that device.

A device interface module is also responsible for assigning physical storage areas, as needed, on the device for which it was written. To accomplish this function, the interface module must maintain records of all storage already assigned on that device. These records are kept in **storage assignment tables** which reside on the device to which they refer.

4.2 Other File System Modules

The modules described below are not considered part of the basic file system and are not indicated in Fig. 3. However, these modules are considered to be a necessary and integral part of the file system as a whole.

Multilevel Storage Management Module

The multilevel storage management module operates as an independent process within the Multics system. This module collects information concerning the frequency of use of files currently active in the system. In addition, this module collects information concerning overdrawn accounts from the secondary storage accounting module.

The storage management module insures that an adequate amount of secondary storage is available to the basic file system at all times. This is accomplished by moving infrequently used files downward in the multilevel storage complex. This module also moves the most frequently used files to the highest-level secondary storage device available.

Storage Backup System Modules

The storage backup system consists of five modules which operate as independent processes. These modules perform the functions described in Section 3.

1. Incremental Dump Module--The sole responsibility of this module is to prepare incremental dump tapes of all new or recently modified files.
2. Weekly Dump Module--This module is run once a week to prepare the weekly dump tapes.
3. Retrieval Module--This module retrieves files which have been removed from the on-line storage system.
4. Salvage Module--This module is run after a machine or system failure to correct any inconsistencies which may have resulted in the on-line storage system. Since the Multics system cannot safely be run until these inconsistencies are corrected, the salvage module must be capable of running on a raw machine.
5. Catastrophe Reload Module--This module is used to reload the contents of the on-line storage system from the incremental and weekly dump tapes after a machine or system failure. Normally, this module is run only when all attempts to salvage the contents of the on-line storage system have failed. This module must be capable of running on a raw machine or under the control of the Multics system.

Utility and Service Modules

A large library of utility modules is provided as part of the file system. These modules provide all the necessary functions for manipulating links, and branches using the more primitive functions provided by the file coordinator. A special group of utility modules is provided to copy information currently stored as a file to other input or output media, and vice versa. The following functions are provided as a bare minimum:

1. File to printer
2. File to cards
3. Cards to file
4. Tape to file
5. File to tape

Actually these modules merely place the user's request on a queue for subsequent processing by the appropriate service module. The service module executes the requests in its queue as an independent process. As soon as the user's request has been placed on an appropriate queue, control is returned to the calling procedure although the request has not yet been executed.

5. Conclusions

In this paper, a versatile secondary storage file system is presented. Various goals which such a system should attain have been set, and the system designed in such a way as to achieve these goals. Such a system is felt to be an essential part of an effective on-line interactive computing system.

6. Acknowledgment

The file system presented here is the result of a series of contributions by numerous people, beginning with the MIT Computation Center, continuing with Project MAC, and culminating in the present effort.

References

1. F. J. Corbató and V. A. Vyssotsky, "Introduction and Overview of the Multics System," this volume.

2. E. L. Glaser, J. F. Couleur and G. A. Oliver, "System Design of a Computer for Time-Sharing Applications," this volume.

3. V. A. Vyssotsky, F. J. Corbató and R. M. Graham, "Structure of the Multics Supervisor," this volume.

4. J. F. Ossanna, L. E. Mikus and S. D. Dunten, "Communications and

Input-Output Switching in a Multiplex Computing System," this volume.

5. E. E. David, Jr., and R. M. Fano, "Some Thoughts About the Social Implications of Accessible Computing," this volume.

Additional References

- C. W. Bachman and S. B. Williams, "A General Purpose Programming System for Random Access Memories," Proceedings of the Fall Joint Computer Conference 26, Spartan Books, Baltimore, 1964.
- J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *ACM Conference on Programming Languages,* San Dimas, Calif., Aug. 1965. To be published in *Comm. ACM.*
- A. W. Holt, "Program Organization and Record Keeping for Dynamic Storage Allocation," *Comm. ACM* 4, pp. 422-431, Oct. 1961.
- T. H. Nelson, "A File Structure for the Complex, the Changing and the Indeterminate," *ACM National Conference,* Aug. 1965.
- M. V. Wilkes, "A Programmer's Utility Filing System," *Computer Journal 7,* pp. 180-184, Oct.1964.

** Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).*
