DESIGN STRATEGIES FOR FILE SYSTEMS: A WORKING MODEL

S. E. Madnick
Massachusetts Institute of Technology
U.S.A.

INTRODUCTION

Evolution of File Systems

The evolution of general purpose file systems parallels very closely the evolution of operating systems. This is not surprising since the concept of file systems grew out of the embryonic input-output control (IOC) functions of early operating systems and now represents the most significant component of most modern operating systems.
To a large extent there has been very little attention formally directed to the specific problem of analyzing operating systems. In 1967, Saul Rosen collected together material for a book, "Programming Systems and Languages" <16>, which was to be a distinctive selection of previously published and unpublished reports describing the most important programming languages and discussing many of the most important operating system concepts. He was forced to conclude:

"The paper on Operating Systems was prepared for presentation at the University of Michigan Engineering Summer Conference, June 18–29, 1962. It has had fairly wide circulation as Rand Report P–2584. The material covered has been of vital importance in the development of the "classical" operating system, yet it is difficult to find an adequate treatment outside of very long and usually dry system manuals. George Mealy was one of the few working experts in the field who took the time to write down some of the basic principles of operating systems and also of assembly systems."

Mr. Rosen's observation would imply that indeed very little attention has been expended in the attempt to generalize the functions of operating systems, and file systems have also been severely neglected.
In the early years of computing (roughly 1952–1962), programmers slowly moved away from the practice of approaching a bare machine with card decks and sharpened pencils, fighting with the console for more or less extended periods of time, and leaving triumphantly with final results or in defeat with a ream of machine dump. Operating systems have evolved, not so much as a blessing, but as a practical necessity. As computers became faster and more complex, it was no longer possible for an individual programmer to be an expert in every phase of the programming and machine usage; he now must rely on the operating staff and system programmers to provide the necessities of life.
These operating systems were often ill-designed and usually specialized around a single goal. One of the first truly successful operating systems was FMS (FORTRAN Monitor System) for the IBM 709/7090/7094 family. Its name implies its specialization. As a result a large number of operating systems appeared, each with its own operating procedure and specialization. To a large extent these systems were very closely tied to a programming language (e.g. FORTRAN, COBOL, Assembler).
Input Output Control Systems (IOCS) emerged as a part of the Operating System based on the simple observation that all programs perform some amount of input and/or

output. Therefore, rather than requiring each programmer to write a new set of input/output routines for each program, a common and sufficiently flexible collection of routines were supplied with the Operating System. This situation became especially critical as computer I/O capabilities were extended to include high-speed, buffered, asynchronous channels which required complex program logic to efficiently perform input-output.

From the crude beginnings of IOCS, file systems followed a logical, though often slow, evolution. Once all physical input/output functions were localized in the IOCS, many generalizations became possible. Usually, there is no specific difference among the many tape drives available at an installation, so that any arbitrary tape unit may be used for input or output to a program. Furthermore, later runs at the same or different installations need not use the same unit as long as unique correspondences can be maintained. At first it was considered that the best practice in handling the choice of input-output units by the object program was to include unit assignments as an assembly parameter or to read in unit assignments as data and initialize the program appropriately. This practice worked well when it was followed, which was seldom. With the advent of the near-universal use of IOCS, a more foolproof and flexible manner of operating was to establish the correspondences as part of the IOCS. The object programs dealt strictly in symbolic unit assignments.

Since the object programs no longer interacted directly with the I/O units nor were even aware of unit assignments, additional degrees of freedom became available to the operating system, providing a more efficient and convenient environment. For example, the system could determine unit assignments automatically and dynamically, based upon complex criteria such as availability and performance (e.g. I/O interference, buffering, etc.). The actual technique of I/O (unbuffered, single-buffered, double-buffered, etc.) could be removed from programmer concern.

The proliferation of I/O device types, such as low-speed, medium-speed, high-speed and hyper-tapes, as well as drums and disks of all shapes and sizes, resulted in the expansion of IOCS to include capabilities that are now called data management or file system facilities. The basic notion exploited is that just as the programmer had little concern as to what tapes were to be used, he really does not care what device is used nor what method of I/O is employed within broad logical constraints. For example, if a programmer wishes logically to treat his I/O data as 80 column cards, the file system could physically utilize unit-record equipment, tapes, disks, drums, data cells, or a host of other devices in various manners logically to simulate the effect of input-output using 80 column cards.

This trend became irreversible with the advent of multi-tasking operating systems, since the availability of devices was continuously and dynamically changing. In such an environment, it becomes impractical and probably impossible to designate specific I/O units statically and arbitrarily in the program.

The importance of these data management and file systems cannot be overly emphasized. Just as the assumption that programs perform input-output was a basic fact, it appears that the amount and flexibility of I/O requirements demanded by programs are continuously increasing.

A major factor in the rapid growth of file systems can be directly and indirectly attributed to the introduction of low cost, high capacity, high-speed, direct access devices such as disks, drums, and data cells. A description of direct access devices would emphasize the fact that they have two degrees of freedom rather than only one as with tape-like devices. Since these devices can be used for both sequential and direct access applications, the total amount of usage increases. Of course, the extra degrees of freedom necessitate more complex I/O routines and further tighten the reliance on file systems to perform these functions.

Direct access devices are usually as flexible or more flexible than tape devices. Card-image or printer-image fixed record data types can be handled as well as variable-length or structured data forms. Although these capabilities could be performed by the object program, the vast majority of these functions have been subsumed as by-products of the file system.

The second major factor contributing to the rising importance of file systems, as in early operating systems, was necessity. This time it was due to the "information explosion". As the number of users, uses, and sophistication of use increased, the amount of information in the forms of programs and data rose correspondingly. It was no longer convenient nor usually physically possible to haul the required boxes and boxes of programs and data to and from the machine. This information was converted and maintained in a more compact but directly machine processable form, such as magnetic tape or disk pack. Not only were the individual programs and data collections large, but the total number of distinct and unique files (i.e. programs and data collections) was very large. It is not uncommon for a single programmer to have to use 10 to 100 separate programs and a roughly equivalent number of data collections. This situation became especially acute with the increased use of online systems. A user at a remote teletype terminal could not be expected to re-type and enter all his programs and data from the terminal. They must be permanently maintained and stored at the central computer facility, although accessible and alterable under remote terminal control. Quite obviously, it would be uneconomic and unmanageable to store each unique file on a separate tape or disk pack. Thus, people were faced with the problem of using the I/O devices to store thousands of permanent files in addition to the traditional use for input, output and "scratch" storage. Direct access devices provide the capability of storing hundreds or thousands of unique files and accessing them in any order conveniently. This type of direct access device usage results in many side effects. The first problem, of course, involves a complex storage organization to locate "empty" space on the device and a directory-like mechanism to keep track of the individual files. Many other facilities are usually required, such as a security system to prevent unauthorized access to restricted files, and procedures to recover from hardware or software failures. Of course, each installation or group develops additional elegant file system capabilities to meet special requirements or to provide extensive flexibility.

For the same reasons that programmers utilize and rely on the file system, the operating system uses the facilities of the file system. For example, user identification (e.g. passwords, account numbers, etc.), accounting and charge information as well as system self-measurement data must be maintained dynamically using the facilities of the file system. The previously mentioned directories of "empty" space on direct access devices and the symbolic file directory and access control information are usually handled as system files. The operating system uses the file system capabilities to store the various processing programs (e.g. FORTRAN, COBOL, Assemblers, etc.) as well as many infrequently used supervisor routines. Furthermore, advanced operating systems perform "spooling", roll-in/roll-out, and paging in conjunction with the file system. It is not hard to realize that the file system is usually the most important component of an operating system on the basis of amount of manpower required to develop and implement, and the amount of instructions and space used by the file system.

Whereas the early operating systems along with their rudimentary file systems revolved around the need to support miscellaneous I/O functions for programming languages, modern file systems are at the very center of the operating system. The supervisor, programming systems, and object programs are totally dependent on the file system.

Scope and Purpose

The development of file systems has suffered many of the same problems as programming languages. Probably the single most important problem was the excessive concern with efficiency. Of course efficiency is important, but in most current-day programming situations other factors, such as productivity and flexibility, are finally receiving their long-deserved attention. The question of efficiency can be put into proper perspective from recent studies of real programming groups, where it has been found that the "best" programmer was up to 15 times more "efficient" than the least proficient programmer. It is not the function of this paper to get deeply involved in programming language controversies, but to illustrate the trends and changing attitudes. For example, if the original designers of FORTRAN had not felt that its acceptance depended on the utmost attention to efficiency and, therefore, had not defined the language in terms of the hardware capabilities of a specific machine, IBM 704, it is possible that the evolution of languages such as FORTRAN-IV, COBOL, ALGOL, and PL/I and generalized compiler techniques might have proceeded in a more organized fashion. The entire field of generalized approaches to programming languages and compiler techniques has only recently emerged as a major factor in the computing profession.

File systems have followed a similar development. In the name of "efficiency", each new file system was specially tailored to the original needs and environment of its intended use and very seldom could benefit from the experience or techniques of preceeding systems. As the demands on a given file system increased, new features and facilities were added, often with a "crowbar". Each of these piece-meal file systems drove us further and further from an organized, generalized file system structure.

Most literature in this area has appeared in one of two forms. The typical system manuals describe the "clever" techniques used to implement a specific file system, but provide very little assistance for comparisons with other current systems or in the design of new file systems. The other type of reference deals with discussions of desirable characteristics for future file systems, usually emphasizing user facilities, but adds little insight into the problems of designing and implementing such a system.

To a certain extent, generalized approaches have begun to evolve in "time-sharing" systems. In this paper such systems will be called conversational resource-sharing, since time is only one of many resources that are shared and it is the conversational or interactive nature of these systems that is most easily distinguished from batch-oriented operating systems.

These generalized file systems for conversational resource-sharing operating systems developed both by design and necessity. In order to provide all the features required by user programs and the supervisor, a flexible design was essential. Furthermore, owing to the complexity of the environment and its dynamically changing aspects, it would be impossible to devise an "optimally efficient" strategy. The implementers were thus forced to abandon any attempt to make the system more efficient and were free to develop a flexible system with a clear conscience.

The goals of flexibility and efficiency need not be contradictory. In any multi-tasking system, which includes most modern, non-conversational, batch-oriented operating systems as well as conversational systems, I/O operations can be performed asynchronously by channels, and the central processor time can be utilized by executing other tasks while I/O is in progress. In this environment file system efficiency ceases to be of paramount concern. Furthermore, individual user attempts to optimize performance could result in unnecessary inefficiencies due to conflicts with other tasks, such as excessive I/O interference from overloading the channels. The file system, aware of the total requirements, could provide a strategy that results in a more harmonious arrangement, increasing system throughput far more than individual user optimization could.

Even in single-task or application-oriented operating systems, there is definite value to an organized, generalized file system. For most large, complex user programs as well as compilers and assemblers, the program action including precise file system requirements cannot be statically determined since it is a dynamic function of the input data supplied. Therefore, a dynamically flexible file system could often outperform a specialized, but inflexible, file system.
It is the purpose of this paper to present a general file system design. It is extremely important to start with a flexible but precise model although this design will probably need to be modified and made more detailed for any specific implementation. This issue was highlighted by Robert Rappaport in his thesis "Implementing Multi-Process Primitives in a Multiplexed Computer System" <15> which describes the development of the Traffic Controller for the MIT Project MAC Multics System:

"After having found acceptable solutions for the problems at hand, one asks oneself why it took so long to arrive at these solutions and was there any way to have done it more quickly? One might further ask if the arrived-at solutions are in any sense optimum?
After being involved in designing a large system involving the work of many people, one gets the feeling that such problems as were encountered here are bound to crop up. The development of any large system can only remain manageable if distinct parts of the system remain modular and independent.
Without a theory of computing systems to fall back on, designing such complex systems becomes an art, rather than a science, in which it is impossible to prove the degree to which working solutions to problems are in any sense optimum solutions. In much the same way as authors write books, large computer systems go through several drafts before they begin to take shape. In the absence of a theory one can only cope with the complexity of the situation by proceeding in an orderly fashion to first produce an initial working model of the desired system. This part of the work represents the major effort of the design and implementation project. Once having arrived at this benchmark, many of the problems may then be seen in a clearer light and revisions to the working model are implemented much more quickly than were the original modules. As to the development of a theory, one gets the impression that it will be a long time in coming."

Therefore, while we await THE general theory of computer science, the file system model presented in this paper will hopefully serve the need for an "initial working model" from which "problems may be seen in a clearer light".

BASIC CONCEPTS USED IN FILE SYSTEM DESIGN

Two concepts are basic to the general file system model to be introduced. These concepts have been described by the terms "hierarchical modularity" and "virtual memory". They will be discussed briefly below.

Hierarchical Modularity
The term "modularity" means many different things to different people. In the context of this paper we will be concerned with an organization similar to that proposed by Dijkstra <6><7> and Randell <14>. The important aspect ot this organization is that all activities are divided into sequential processes. A hierarchical structure of these sequential processes results in a level or ring organization wherein each level only communicates with its immediately superior and inferior levels.
The notions of "levels of abstraction" of "hierarchical modularity" can best be presented briefly by an example. Consider an aeronautical engineer using a matrix inversion package

to solve space flight problems. At his level of abstraction, the computer is viewed as a matrix inverter that accepts the matrix and control information as input and provides the inverted matrix as output. The application programmer who wrote the matrix inversion package need not have had any knowledge of its intended usage (superior levels of abstraction). He might view the computer as a "FORTRAN machine", for example, at his level of abstraction. He need not have any specific knowledge of the internal operation of the FORTRAN system (inferior level of abstraction), but only of the way in which he can interact with it. Finally, the FORTRAN compiler implementer operates at a different (lower) level of abstraction. In the above example the interaction between the 3 levels of abstraction is static since after the matrix inversion program is completed, the engineer need not interact, even indirectly, with the applications programmer or compiler implementer. In the form of hierarchical modularity used in the file system design model, the multi-level interaction is continual and basic to the file system operation.

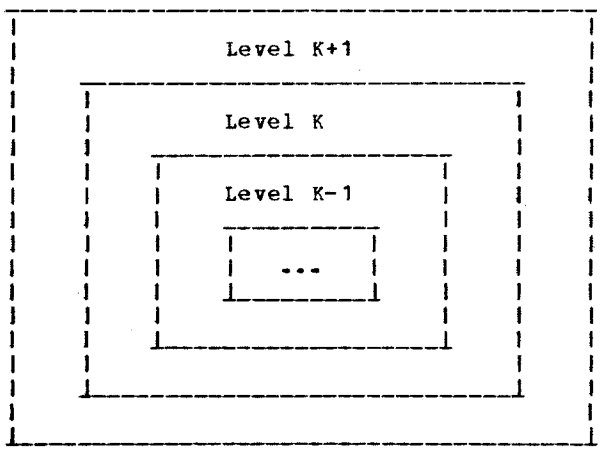

Figure 1.
Level, Ring-like Modular Organization

There are several advantages to such an modular organization. Possibly the most important is the logical completeness of each level. It is easier for the system designers and implementers to understand the functions and interactions of each level and thus the entire system. This is often a very difficult problem in very complex file systems with tens or hundreds of thousands of instructions and hundreds of inter-dependent routines. Another by-product of this structure is "debugging" assistance. For example, when an error occurs it can usually be localized at a level and identified easily. The complete verification (reliability checkout) of a file system is usually an impossible task since it would require tests using all possible data input and system requests. In order to construct a finite set of relevant tests, it is necessary to consider the internal structure of the mechanism to be tested. Therefore, an important goal is to design the internal structure so that at each level, the number of test cases is sufficiently small that they can all be tried without overlooking a situation. In practice, level 0 would be checked-out and verified, then level 1, level 2, etc., each level being more powerful, but because of the abstractions introduced, the number of "special cases" remains within bounds.

Virtual Memory

There are four very important and difficult file system objectives: (1) a flexible and versatile format, (2) as much of the mechanism as possible should be invisible, (3) a degree of machine and device independence, and (4) dynamic and automatic allocation of secondary storage. There have been several techniques developed to satisfy these objectives in an organized manner; the concept exploited in this generalized file system has been called "segmentation" <5> or "named virtual memory" <3>. Under this system each file is treated as an ordered sequence of addressable elements, where each element is normally the same size unit as the main storage, a byte or word. Therefore, each individual file has the form of a "virtual" core memory, from whence the name of the technique came. The size of each file is allowed to be arbitrary and can dynamically grow and shrink. There is no explicit data format associated with the file; the basic operations of the file system move a specified number of elements between designated addresses in "real" memory and the "virtual" memory of the file system.

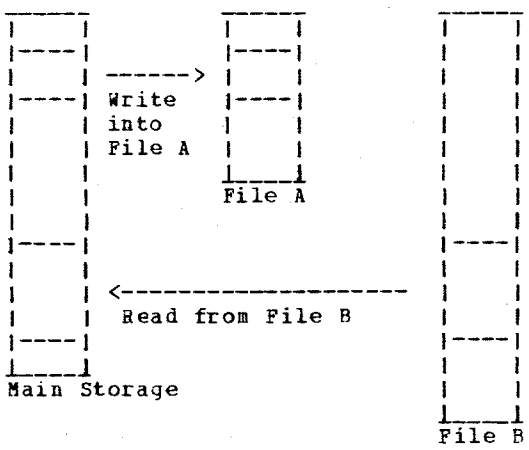

Figure 2.
"Real" Memory and "Virtual" File Memory

There are several reasons for choosing such a file concept. In some systems the similarity between files and main storage is used to establish a single mechanism that serves as both a file system for static data and program storage and a paging system <3><5><18> for dynamic storage management. "Virtual memory" provides a very flexible and versatile format. When specific formatting is desired, it can be accomplished by the outermost file system level or by the user program. For example, if a file is to be treated as a collection of card-image records, it is merely necessary to establish a routine to access 80 characters at a time starting at byte locations 0, 80, 160, ... Almost all other possible formats can be realized by similar procedures.

Except for the formatting modules, the entire file system mechanism, including allocations, buffering, and physical location, is completely hidden and invisible to the user. This relates closely to the objective of device independence. In many file systems

the user must specify which device should be used, its record size (if it is a hardware formatable device), blocking and buffering factors, and sometimes even the physical addresses. Although the parameters and algorithms chosen might, in some sense, be optimal, many changes might be necessary if the program is required to run with a different configuration or environment.

There are very serious questions of efficiency raised by this file system strategy. Most of these fears can be eased by the following considerations. First, if a file is to be used very seldom as in program development, efficiency is not of paramount importance; if, on the other hand, it is for long-term use as in a commercial production program, the device-independence and flexibility for change and upkeep will be very important. Second, by relieving the programmer of the complexities of the formats, devices, and allocations, he is able to utilize his energy more constructively and creatively to develop clever algorithms relating to the logical structuring of his problem rather than clever "tricks" to overcome the shortcomings or peculiarities of the file system. Third, in view of the complexity of current direct-access devices, it is quite possible that the file system will be better able to coordinate the files than the average user attempting to specify critical parameters.

OVERVIEW OF FILE SYSTEM DESIGN MODEL

The file system design model to be presented in this paper can be viewed as a hierarchy of six levels. In a specific implementation certain levels may be further sub-divided or combined as required. A recent study of several modern file systems, which will be published in a separate report, attempts to analyze the systems in the framework of this basic model. In general all of the systems studied fit into the model, although certain levels in the model are occasionally reduced to trivial form or are incorporated into other parts of the operating system.

The six hierarchical levels are:

1. Input/Output Control System (IOCS)
2. Device Strategy Modules (DSM)
3. File Organization Strategy Modules (FOSM)
4. Basic File System (BFS)
5. Logical File System (LFS)
6. Access Methods and User Interface

The hierarchical organization can be described from the "top" down or from the "bottom" up. The file system would ordinarily be implemented by starting at the lowest level, the Input/Output Control System, and working up. It appears more meaningful, however, to present the file system organization starting at the most abstract level, the access routines, and removing the abstractions as the levels are "peeled away".

In the following presentation the terms "file name", "file identifier", and "file descriptor" will be introduced. Detailed explanations cannot be provided until later sections, the following analogy *may* be used for the reader's assistance. A person's name (file name), due to the somewhat haphazard process of assignment, is not necessarily unique or manageable for computer processing. A unique identifier (file identifier) is usually assigned to each person, such as a Social Security number. This identifier can then be used to locate efficiently the information (file descriptor) known about that person.

Access Methods (AM)

This level consists of the set of routines that superimpose a format on the file. In general there will probably be routines to simulate sequential fixed-length record files, sequential variable-length record files, and direct-access fixed-length record files, for example. Many

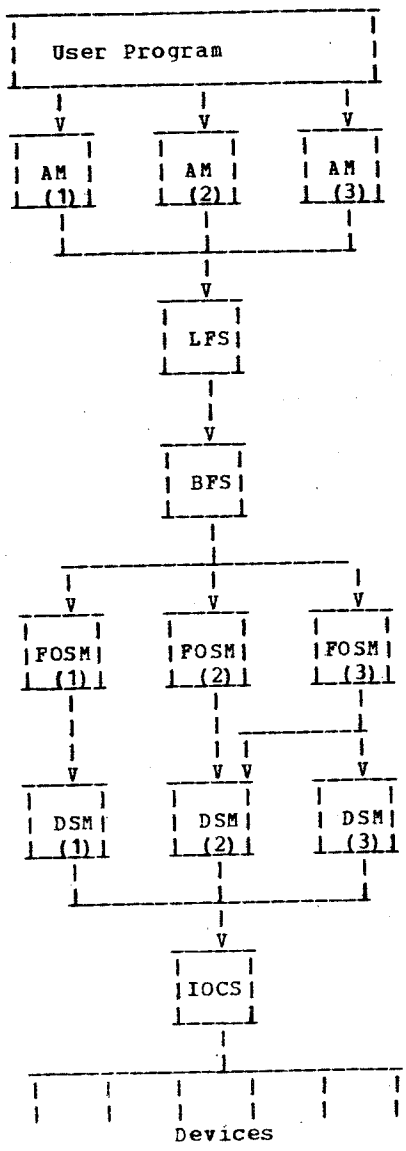

Figure 3.
Hierarchical File System

more elaborate and specialized format routines, also called access methods or data management, can be supplied as part of the file system. Obviously, a user may write his own access methods to augment this level.

Logical File System (LFS)
Routines above this level of abstraction associate a symbolic name with a file. It is the function of the Logical File System to use the symbolic file name to find the corresponding unique "file identifier". Below this level the symbolic file name abstraction is eliminated.

Basic File System (BFS)
The Basic File System must convert the file identifier into a file descriptor. In an abstract sense, the file descriptor provides all information needed to physically locate the file, such as the "length" and "location" of the file. The file descriptor is also used to verify access rights (read-only, write-only, etc.), check read/write interlocks, and set up system-wide data bases. The Basic File System performs many of the functions ordinarily associated with "opening" or "closing" a file. Finally, based upon the file descriptor, the appropriate FOSM for the file is selected.

File Organization Strategy Modules (FOSM)
Direct-access devices physically do not resemble a virtual memory. A file must be split into many separate physical records. Each record has a unique address associated with it. The File Organization Strategy Module maps a logical virtual memory address into the corresponding physical record address and offset within the record.
To read or write a portion of a file, it is necessary for the FOSM to translate the logically contiguous virtual memory area into the correct collection of physical records or portion thereof. The list of records to be processed is passed on to the appropriate DSM.

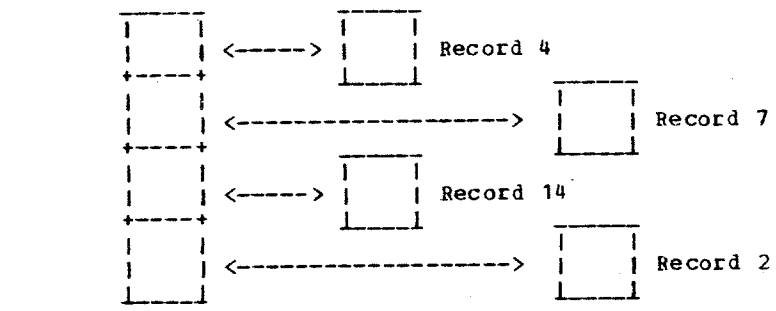

Figure 4.
Mapping Virtual Memory Into Physical Records

To minimize redundant or unnecessary I/O, the FOSM allocates "hidden" file buffers as needed. If the requested portion of virtual memory is contained in a currently buffered

record, the data can be transferred to the designated user main storage area without intervening I/O. Conversely output to the file may be buffered. If a sufficiently large number of buffer areas are located to a file, it is possible that all read and write requests can be performed by merely moving data in and out of the buffers. When a file is "closed", the buffers are emptied by updating the physical records on the secondary storage device and released for use by other files. Buffers are only allocated to files that are actively in use (i.e. "open").

Device Strategy Modules (DSM)
When a large portion of a file is to be read or written, many records must be processed. The Device Strategy Module considers the device characteristics such as latency and access time to produce an optimal I/O sequence from the FOSM requests.
The DSM also keeps track of the available records on the device. It is responsible for allocating records for a file that is being created or expanded, and deallocating records for a file that is being erased or truncated. The FOSM requests that a record be allocated when needed, the DSM selects the record.

Input/Output Control System (IOCS)
The Input/Output Control System coordinates all physical I/O on the computer. Status of all outstanding I/O in process is maintained, new I/O requests are issued directly if the device and channel are available, otherwise the request is queued and automatically issued as soon as possible. Automatic error recovery is attempted when possible. Interrupts from devices and unrecoverable error conditions are directed to the appropriate routine. Almost all modern operating systems have an IOCS.

File Systems versus Data Management Systems
In the literature there is often confusion between systems as described above, which this paper calls "file systems" and systems which will be called "data management systems", such as DM–1<8>, GIM–1<13>, and TDMS<17>. The confusion is to be expected since both types of systems contain all of the functional levels described above. The systems differ primarily on the emphasis placed on certain levels.
In general file systems, the file is considered the most important item and emphasis is placed on the directory organization (Logical File System) and the lower hierarchical levels. It is expected that specialized access methods will be written by users or supplied with the system as needed.
In most data management systems, the individual data items are considered the most important aspect, therefore emphasis is placed on elaborate access methods with minimal emphasis on the lower levels of abstraction. Because of the heavy emphasis on a single level, data management systems tend to appear less hierarchical than file systems since the lower levels are often absorbed into the access methods.

Access Methods

The virtual memory interface provided by the Logical File System allows for very flexible user applications and access methods. In a PL/1–like notation, calls to the Logical File System are of the form:

LFS–Read/Write (Filename, Addr1, Addr2, Number);

where Addr1 is the main storage address, Addr2 is the file virtual memory address, and Number is the number of elements to be moved.
In this report elements will be assumed to be 8-bit bytes. For example, a request to read 100 bytes from location 200 within the file named ALPHA into main storage location

1234 could be expressed:

LFS–Read ('ALPHA', 1234, 200, 100);

Sequential fixed-length records, sequential variable-length records, and direct-access fixed-length records are common access methods. All of these organizations and many more can be realized using a file's virtual memory. Note that the records processed by the access methods are "software" records and have no relation to the physical/logical records processed by the FOSM and DSM.

Sequential and Direct-Access Fixed-Length Record Access Methods

To simulate these access methods, the file's virtual memory is treated as a sequence of records of the desired lengths, L.

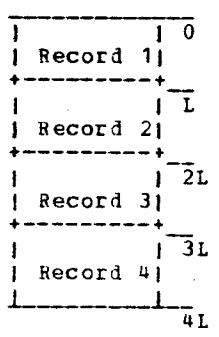

Figure 5.
Layout of Virtual Memory For Fixed-Length Record Access Methods

To access these records sequentially, a position counter, PC, is set aside that starts at 0 and is incremented by L after each read or write. The position counter therefore finds the location of the next sequential record. The routine could be written as:

LFS_Read (Filename, Location, PC, L);
PC = PC + L;

To access these records by direct-access there is no need for a position counter since the desired record, r, can be found at location (r–1)*L in the file's virtual memory. This routine could be written as:

LFS_Read (Filename, Location, (r–1)*L, L);

Sequential Variable-Length Record Access Method

The Sequential Variable-Length Record Access Method treats the file as an ordered sequence of records, each record may be a different length. This method can be implemented by preceeding each record with a "hidden" length field.

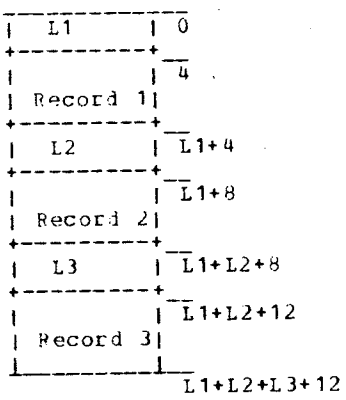

Figure 6.
Layout of Virtual Memory for Variable-Length Record Access Method

These records can be accessed using a variation of the Sequential Fixed-Length scheme. For example:

```
LFS_Read (Filename, L, PC, 4);/* Get 4 byte length */
LFS_Read (Filename, Location, PC+4, L);/* Get data */
PC = PC + L + 4;/* Update position counter */
```

Other Access Methods

The above examples were presented to illustrate the ease with which conventional access methods can be supported under this file system design. The real importance of the virtual memory concept is not its ability to provide traditional access methods, but the ease and flexibility with which problem-oriented access methods can be developed. The programmer is able to design access methods based on the needs of his problem rather than forcing his problem solution to be constrained by a small set of limited access methods.

The power of a computer reaches its peak when it is capable of amplifying the creativity of the programmer. A system that restricts the programmer's ability to express his ideas provides a minimal service.

Logical File System

A user's program references each file by means of a unique symbolic name. It is the function of the Logical File System to convert the symbolic name reference into its corresponding unique file identifier. The Logical File System performs the mapping using a "file directory organization".

In the simplest case the file directory is entirely stored in main storage as a two-entry

table. The two entries are the symbolic file name and its corresponding file identifier. A look-up routine is all that is needed to serve the function of the Logical File System. This approach is used by several file systems because of its simplicity and efficiency. Unfortunately, the number of files that are allowed in the file system is restricted by the amount of main storage available for the file directory.

To remove the above limitation, many file systems keep the file directory on secondary storage. The file directory is often treated as a standard file whose file descriptor is always known. This allows the file directory to be processed, expanded, and truncated using the normal file system mechanisms. The Logical File System mapping still involves a table look-up, only this time the table is contained in a file's virtual memory rather than main storage. The calls to the Basic File System are essentially the same as the calls to the Logical File System, only a file identifier is specified rather than a symbolic file name.

A few of the advanced file systems have introduced the concept of the hierarchical file directory. From a simple point of view, a file directory hierarchy resembles and serves a similar purpose to a PL/1 data structure. In practice, certain files are classified as "directories" in addition to their normal attributes. The earlier model of the Logical File System implied that there was only one directory file. This file contained the file identifiers for all the other files, called "data files". This has been extended to allow the base directory, often called the "root directory", to contain file identifiers for directory files as well as data files. Each subsequent directory file can contain file identifiers for other directory files as well as data files.

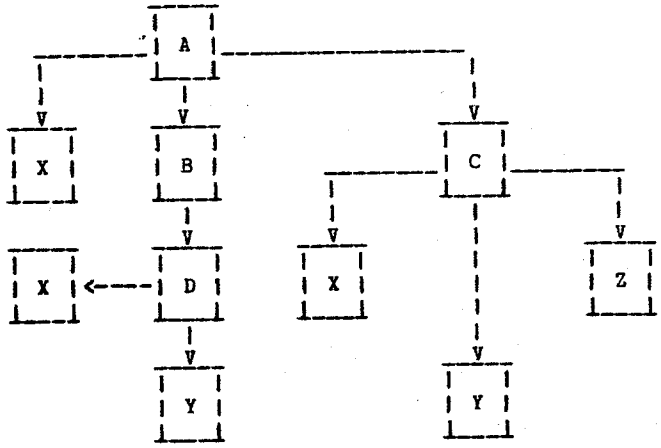

Figure 7.
Hierarchical File Directory Example

Figure 7 illustrates a file directory hierarchy. The files A, B, C, and D are directory files, all the others are data files. The data files, as well as directory files, do not necessarily have unique symbolic names. There are 3 data files in Figure 7 named "X", as in PL/1 this ambiguity is solved by using qualified names such as "A.X", "A.B.D.X", and "A.C.X".

The file directory hierarchy serves many purposes in addition to providing flexible and versatile programmer usage. "File sharing" and "controlled access" among users are very closely tied to the hierarchical directories. Certain of these features are discussed in the paper by Daley and Neumann <4>. A more detailed treatment of this topic will be presented in a subsequent paper by this author.
The implementation of the Logical File System for a file directory hierarchy is a simple extension of the single directory technique. After finding the correct file identifier in the root directory, it is either the data file desired or, if a secondary directory file, is used in exactly the same manner as the root directory identifier to advance one more level in the hierarchy.

Basic File System

As explained in the Overview section, a file is physically located on secondary storage as an ordered collection of distinct records. The information that describes a file's size, access rights, device address or addresses, and the mapping algorithm must be maintained by the file system.
In a simple file system this information can be incorporated into the file directory as long as there is a unique one-to-one mapping of file name onto file. In a sophisticated file system with features such as (1) hierarchical file directory, (2) aliases that allow a single file to be referenced by different names, (3) links that allow a file to be referenced from various directories in the file hierarchy or from different users, and (4) removable or detachable "volumes" or devices, the unique mapping cannot be guaranteed.
To produce an unambiguous file system, the file directory information is divided into three parts, the file name, identifier and the descriptor. The file name directories are the mappings between a symbolic file name and the corresponding identifier. The precise locations of the file descriptors can differ for different implementations, but uniquely defined by the identifier. In fact, since the file descriptors usually need not be searched, they need not be contiguous. Usually they are collected in either (1) a special system wide file, (2) a collection of files, each located on a separate device or volume, or (3) hidden within the symbolic file name directories.
Although it is usually not possible to keep the symbolic file directories in main storage, the number of files actively in use is sufficiently small that the corresponding file descriptors can be placed in a core-resident table called the Active File Directory or Open File Directory.
It is the function of the Basic File System to use the unique file identifier to locate the file descriptor and place it in the Active File Directory unless it has already been "opened". The Basis File System also checks that the action requested upon the file such as read, write, or delete does not violate the restrictions specified in the file descriptor.
After verifying legal access to the file, the Basic File System passes control to the appropriate File Organization Strategy Module as specified in the final descriptor entry.

File Organization Strategy Modules

The primary function of the File Organization Strategy Module is to map a file's virtual memory address onto a corresponding physical record number. There are at least three common physical file organization strategies: sequential, linked, and indexed.

Sequential File Organization Strategy
The Sequential File Organization Strategy is used by most of the older, simpler, and

non-dynamic file systems. Under this technique logically consecutive records are physically consecutive. For example, if each record is 1000 bytes long, virtual address 3214 would be located in the fourth logical record. If the first logical record (i.e., the one containing virtual address 0) is physical record 120, the record containing virtual address 3214 would be physical record 123.

There are two notable advantages claimed for this technique. Firstly, the mapping is very simple and efficient. The only information needed is the fixed record size and the address of the first record. Secondly, if the file is to be processed in a sequential manner, the consecutive organization allows for minimizing device latency and access time.

Although the first point is indisputable, the second claimed advantage is open to question. If there is more than one file on the same device that is actively in use, as is common in a multi-tasking environment, then the device read/write positioning will be switching rapidly among the active files, defeating the assumed sequential accessing.

The major disadvantage of this sequential organization is that the maximum size of the file must be assumed statically before creating the file. By specifying too small a size, the task will be forced to terminate if more space is needed. If too large a size is assumed, as is common, there is much wasted space and fragmentation.

This technique may be recommended for single-tasking systems with few permanent files and very few files simultaneously in use. It might be useful for a large information utility system which is based on a large number of independent, low cost, low usage, high capacity devices such as data cells where wasted space is not a significant problem.

Linked File Organization Strategy

The Linked and Indexed File Organization Strategies allow for files to dynamically grow and shrink. The linked technique was probably developed first since it is simpler and emphasizes sequential characteristics which were primarily used in early file systems.

The linked organization requires each record of a file to specify the location of the next logical record, analogous to the "links" on a chain. The file descriptor specifies only the location of the first record. It tells nothing about the locations of the other records. As the file grows, new records are dynamically allocated and linked onto the file.

For sequentially processed files, the linked technique provides a very simple and efficient mechanism. A few bytes are used in each record to record the link, and since record sizes are usually in the range of 1000 bytes the overhead is minimal. Unfortunately, random or direct-access file usage poses serious problems. If, for example, the last access was to a data area in logical record 5, a reference to an area in logical record 15 will require 9 intermediate I/O accesses to find the links before reaching the desired record. The Linked File Organization Strategy has been used satisfactorily on systems where the vast majority of files are accessed sequentially.

Indexed File Organization Strategy

The Indexed File Organization Strategy is a significant variation to the linked technique. Records are dynamically allocated as needed, but rather than distributing the record addresses throughout the file as links, they are collected together as a table. The logical record number is used as an "index" in the table to find the corresponding physical record number.

If files are limited to small or medium sizes, the index table can be stored as part of the file descriptor. If files are allowed to be arbitrarily, the index table must itself be treated as a file and is broken into separate records. In the former case, sequential and random access processing proceed easily and efficiently. In the latter case, sequential processing is very efficient, except for intermittent accesses for the next portion of the index table. Random processing may be very efficient if localized to a simple index table block; in any case it will never exceed a small number of intermediate accesses, usually one or two, for

totally random processing.
The Indexed File Organization Strategy has the advantage of allowing the concept of a "sparsely filled" file. If we assume that each physical record is 1000 bytes and each index (record number) is 4 bytes, then the index table for a file that is 250,000 bytes long would require 250 indexes or 1000 bytes. By designating a special code, such as 0, to indicate an index for a non-allocated record, a file can be created with specific contents at locations 10,000, 40,000, and 247,000 but with unspecified contents elsewhere. By convention, unspecified contents are usually initialized as zero by the file system. The above sparse file would only require four physical records, three records for the specified portions of the file and one record for the index table. As more information is written into a sparse file, more physical records will be allocated as needed.
The indexed organization provides a simple and efficient way to use programming techniques, such as "hash coding" or "random entry" tables, that require a large though sparse virtual memory.
Many of the most recent file systems have adopted techniques similar to the Indexed File Organization Strategy.

Device Strategy Modules

The Device Strategy Module serves two separate functions: building physical I/O lists from FOSM logical I/O requests, and performing allocation and deallocation of physical records.

Building I/O Lists
In addition to the obvious "read" and "write" functions, direct access devices often require I/O commands, such as "seek" or "search" for proper positioning. The FOSM deals only in the logical acts of reading and writing. It transfers a set of requests to the DSM of the form: "read record 24 into location 5400, read record 49 into location 6400, and write record 27 from location 9324". The DSM must translate these requests into the obscure I/O list format required for the particular device.
Furthermore, due to device characteristics such as latency and access time, the order in which the requests are performed affects the total amount of time that the device is kept "busy". For example, if records 24 and 27 are "closer", in some sense, to each other than record 49, it might be more efficient to read record 24, write record 27, and then position to read 49.

Allocation And Deallocation Of Physical Records
When the FOSM maps a valid write request onto a logical record for which a physical record has not been allocated, the DSM is called to find an available record for use. There are two common techniques used to keep track of available records. The first technique links all available records together. This method is often used in conjunction with a Linked File Organization Strategy Module. The second technique uses a "bit map" for each device. A bit map is a function which operates on a bit string and describes the relationship between a bit position and a physical record on the device. For example a convenient bit map might be: bit 0 corresponds to physical record 0, bit 1 to physical record 1, etc. If a bit is set to 0, the corresponding record is available for allocation, otherwise it has already been allocated to a file. The bit map provides a very compact representation of the allocation information. The allocation states of a device with a capacity of 8,000,000 bytes divided into 8000 1000-byte records can be stored in a 1000 byte bit map. In a file system with a large number of high-capacity direct-access devices, it may be impossible to keep all the bit maps in main storage. The bit map may be

subdivided into sections, such as a separate bit map for each group of 800 records. Only one section of the bit map for a device is kept in main storage at a time, the remaining sections are left stored on the device.
Since sequential processing is a very common file usage, the DSM may attempt to allocate records to take advantage of this fact. Of course, any specific File Organization Strategy Module and Device Strategy Module group are expected to be cooperative to optimize overall performance. The precise nature of meaningful cooperation would be too detailed to discuss in this paper.

Input/Output Control System

The Input/Output Control System coordinates all the physical I/O on the computer. On most modern computers there are complex interdependencies among the physically independent I/O devices. Usually this dependency occurs due to the dedicated nature of "selector" channels and device control units that can switch to any device but can only service one device at a time. For very high-speed devices, such as drums, the main storage access time can be an important factor. If too many simultaneous memory requests occur, "overrun" can occur resulting in erroneous data transmission. The IOCS keeps track of the status of all devices, control units, and channels. When an I/O operation is requested, the IOCS checks to insure a clear path to the device through the channels and control units and that no I/O capacity limits will be exceeded. If it is not possible to issue the requested I/O operation, the IOCS stores the request on a queue. The I/O will be issued at a later time when all conditions are satisfied. Since the I/O interdependencies may exist among all devices, every I/O operation whether for the file system or dedicated special purpose device must be funnelled through the IOCS.
Although most modern I/O devices are very reliable, spurious errors do occur. Usually the retry or recovery procedure is very simple, in such a case the IOCS will attempt corrective measures.
The caller to the IOCS is informed of the status of his I/O request, for example (1) successful completion, (2) unrecoverable error condition, or (3) asynchronous interrupt. The sophistication and scope of the IOCS depends upon the devices to be handled and the goals of the file system and operating system.

CONCLUDING COMMENTS

To a large extent file systems are currently developed and implemented in much the same manner as early "horse-less carriages", that is, each totally unique and "hand-made" rather than "mass produced". Compilers, such as FORTRAN, were once developed in this primitive manner; but due to careful analysis of operation (e.g., lexical, syntax, and semantic analysis, etc.), compilers are sufficiently well understood that certain software companies actually offer "do-it-yourself FORTRAN kits". Since modern file systems often outweigh all other operating system components such as compilers, loaders, and supervisors, in terms of programmer effort and number of instructions, it is important that a generally applicable methodology be found for file system development.
This paper presents a modular approach to the design of general purpose file systems. Its scope is broad enough to encompass most present file systems of advanced design and file systems presently planned, yet basic enough to be applicable to more modest file systems. The file system strategy presented is intended to serve two purposes: (1) to assist in the design of new file systems and (2) to provide a structure by which existing file systems may be analyzed and compared.

ACKNOWLEDGEMENTS

The author acknowledges the many long and often heated discussions with his colleague, Mr. Allen Moulton, from which many of the basic ideas for this file system design were moulded.
Many colleagues generously contributed their time, energy, and criticism to help produce this final document. Special thanks are due to Prof. John J. Donovan, Prof. David Ness, and Prof. Robert M. Graham, as well as, Stephen Zilles, Ben Ashton, Hoo-min Toong, Michael Mark, Derek Henderson, Norm Kohn, and Claude Hans.
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).

REFERENCES

1. R. E. ELEIER
   Treating hierarchical data structures in the SDC time-shared data management system (TDMS)
   ACM national conference proceedings 1967

2. F. J. CORBATO et al
   The compatible time-sharing system
   MIT press Cambridge 1962

3. R. C. DALEY and J. B. DENNIS
   Virtual memory, processes and sharing in multics
   Communications of the ACM May 1968

4. R. C. DALEY and P. G. NEUMANN
   A general purpose file system for secondary storage
   Proceedings fall joint computer conference 1965

5. J. B. DENNIS
   Segmentation and the design of multi-programmed computer systems
   Journal of the ACM October 1965

6. E. W. DIJKSTRA
   The structure of the 'THE' multiprogramming system
   ACM symposium on operating systems principles Gatlinburg Tennessee October 1967

7. E. W. DIJKSTRA
   Complexity controlled by hierarchical ordering of function and variability
   Working paper for the NATO conference on computer software engineering Garmisch Germany October 7–11 1968

8. P. J. DIXON and DR. J. SABLE
   DM–1 – a generalized data management system
   Proceedings of the 1967 spring joint computer conference

9. IBM CAMBRIDGE SCIENTIFIC CENTER
   CP–67/CMS program logic manual
   Cambridge Massachusetts April 1968

10. IBM CORPORATION
IBM System/360 time sharing system access methods
Form Y28–2016–1 1968

11. S. E. MADNICK
Multi-processor software lockout
ACM national conference proceedings August 1968

12. S. E. MADNICK
Design strategies for file systems: a working model
File/68 international seminar on file organization Helsingør Denmark November 1968

13. D. B. NELSON, R. A. PICK and K. B. ANDREWS
GIM–1 – a generalized information management language and computer system
Proceedings of 1967 spring joint computer conference

14. B. RANDELL
Towards a methodology of computer system design
Working paper for the NATO conference on computer software engineering Garmisch Germany October 7–11 1968

15. R. L. RAPPORT
Implementing multi-process primitives in a multiplexed computer system
S. M. thesis MIT department of electrical engineering August 1968

16. S. ROSEN
Programming systems and languages
McGraw-Hill New York 1967

17. J. H. SALTZER
CTSS technical notes
MIT project MAC MAC-TR-16 August 1965

18. J. H. SALTZER
Traffic control in a multiplexed computer system
Sc. D thesis MIT department of electrical engineering August 1968

19. A. L. SCHERR
An analysis of time-shared computer systems
MIT project MAC MAC-TR-18 June 1965

20. SCIENTIFIC DATA SYSTEMS
SDS 940 time-sharing system technical manual
Santa Monica, California August 1968

21. L. H. SEAWRIGHT and J. A. KELCH
An introduction to CP--67/CMS
IBM cambridge scientific center report 320–2032 Cambridge Massachusetts September 1968