DESIGN STRATEGIES FOR FILE SYSTEMS

Stuart E. Madnick

October 1970

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

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 molded.

Many colleagues generously contributed their time, energy, and criticism to help produce this report. In particular, 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, Joseph Alsop, Derek Henderson, Norm Kohn, and Claude Hans.

The author's association with MIT Project MAC as well as the IBM Cambridge Scientific Center provided the environment and influenced many of the ideas formed in this report.

This report was composed and edited, on-line in the CP-67/CMS Time Sharing computer system, with the aid of the SCRIPT manuscript processing system.

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

DESIGN STRATEGIES FOR FILE SYSTEMS*

Abstract

This thesis describes a methodology for the analysis and synthesis of modern general purpose file systems. The two basic concepts developed are (1) establishment of a uniform representation of a file's structure in the form of virtual memory or segmentation and (2) determination of a hierarchy of logical transofrmations within a file system. These concepts are used together to form a strictly hierarchical organization (after Dijkstra) such that each transformation can be described as a function of its lower neighboring transformation. In a sense, the complex file system is built up by the composition of simple functional transformations. To illustrate the sepcifics of the design process, a file system is synthesized for an environment including a multi-computer network, structured file directories, and removable volumes.

*This report reproduces a thesis of the same title submitted to the Alfred P. Sloan School of Management and the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Master of Science, June 1969.

CONTENTS

ILLUSTRATIONS

CHAPTER ONE

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.

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"<Rosen 67>, 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 alsc cf assembly systems."

Mr. Rosen's observations imply that very little attention has been expended in the attempt to generalize the functions of operating systems. File systems have alsc 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<Rosin 69>. 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 cperating systems appeared, each with its own operating procedure and specialization. These systems were typically very clcsely 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 ccmmon 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 important difference among the many tape drives available at an installation, sc 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 cr 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 number and flexibility of I/O facilities demanded by programs are continuously increasing.

A major factor in the rapid growth of file systems is 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 as 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 of programs and data to and from the machine. This information was converted and maintained in a more compact but directly machine processible 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 from 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 accessable 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. Robert Rosin highlights these developments in his recent survey of supervisor and monitor systems<Rosin 69>:

> "A file system is especially necessary in any system which purports to provide realistic time-sharing. However, the advantages of this facility cannot be overlooked in a more conventional environment".

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 facility 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 in terms of the manpower required to develop and implement, and the amcunt of instructions and space used by the file system.

Whereas the early operating systems along with their rudimentary file systems revolved around the need tc 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 cbject prcgrams are totally dependent on the file system.

Scope and Purpose

The development of file systems has suffered from many of the same problems as that of 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 cannct be statically determined since it is a dynamic functicn cf 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 fcr any specific implementation. This issue was highlighted by Robert Rappaport in his thesis "Implementing Multi-Process Primitives in a Multiplexed Computer System"<Rapp 68> 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 tcck 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 cf 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 tc 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".

CHAPTER TWO

Motivation Behind File System Design

There are two basic goals to be satisfied by the file system design. It is necessary to (1) establish a uniform representation of a file's structure and (2) determine the hierarchy of logical transformations that occur in a file system. W. R. Henry's recent paper on hierarchical data management systems<Henry 69> discusses similar notions cf separating logical and physical file control, but differs significantly from the approaches presented in this report in many fundamental ways. It should be a useful reference to a reader interested in other current research in this area.

Uniform Representation of File Structure

A typical computer system is portrayed by Figure 2.1. Such a configuration usually has a varied assortment of secondary storage devices in addition to the primary storage. Programs and data must be in primary storage in order to be executed or operated upon, respectively.

It is generally true that _if_ primary storage size was limitless and very inexpensive, there would be no need for secondary storage (possible exceptions may be backup requirements and transfer of data). In the framework of this report, the file system will be defined as the software mechanism that extends the capacity of primary storage by handling and coordinating the transfer of information to and from the secondary storage devices. This definition is somewhat more restrictive than other common interpretations which include as part of the file system the physical devices or the programs that operate upon the data. In this interpretation the file system merely stores and transfers information but does not operate upon it.

```
                                        +--------------+
                                   +--->| I/O DEVICE 1 |
                                   |    +--------------+
                                   |
                                   |
+---------+         +---------+    |    +------------+
| PRIMARY |<=======>| CENTRAL |<---+--->| I/O DEVICE |
| STORAGE |         |PROCESSOR|    |    |    2       |
+---------+         +---------+    |    +------------+
                                   |         .
                                   |         .
                                   |         .
                                   |         .
                                   |    +--------+
                                   |    |   I/O  |
                                   +--->| DEVICE |
                                        |    n   |
                                        +--------+
```

Figure 2.1
Physical Computer Configuration

Early file systems were usually designed tc cperate with specific applicaticn programs. Since there are potentially a very large number of different secondary storage devices, many of which can be used in more than one way (e.g. sequential or random access, blocked or unblocked, etc.) each file system limited itself specifically to those devices and organizations that were appropriate fcr its interded application. Figure 2.2 depicts the relaticnships between the applications, the devices, and the file systems.

This type of development produced chaotic situations. It is somewhat analogous to assembly language programming without any established standard calling sequences or communication conventions, which makes it difficult, if not impossible, to use arbitrary programs as subrcutines. In particular, it was quite common to find that data files produced by the payroll programs, using their private file system, could not be accessed by the file system used by the personnel programs, and vice versa. As a result, there was much duplication of effort and confusion in the development and use of these early file systems.

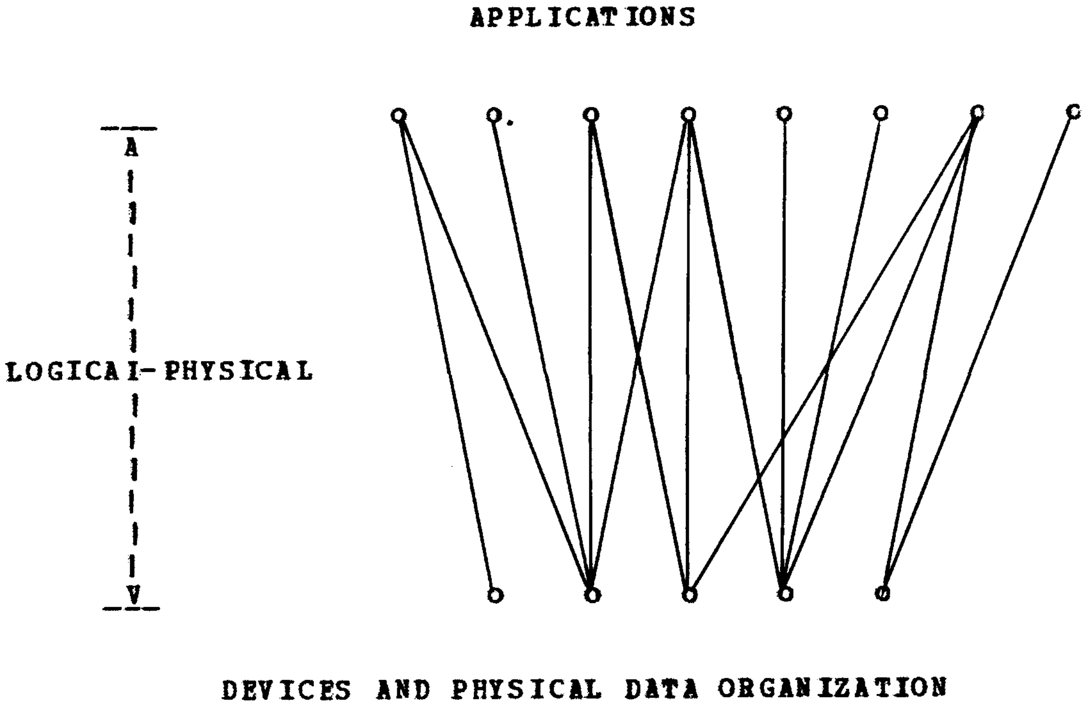

Figure 2.2
Early File Systems

(Analogous to Assembly Language Programming)

More recently, the computer manufacturers and operating system designers realized that it is possible to select a small set of common logical file organizations (or access methods) that can satisfy the needs of most application programs. Furthermore, these access methods could be designed in a flexible manner to operate on a variety of different devices and device organizations. This provided the user with a logically device-independent interface with the file system. Figure 2.3 illustrates this structure.

This approach can be compared with the emergence of Problem Oriented Languages, such as COBOL for business applications and FORTRAN for scientific applications. The access methods file systems suffered the same shortcomings as the programming languages: (1) despite claims, they were not really device independent, (2) occasionally it was necessary to resort to assembly language to overcome or bypass a restriction, and (3) it was not possible to inter-mix access methods (analogy would be to intermix FORTRAN and COBOL subroutines).

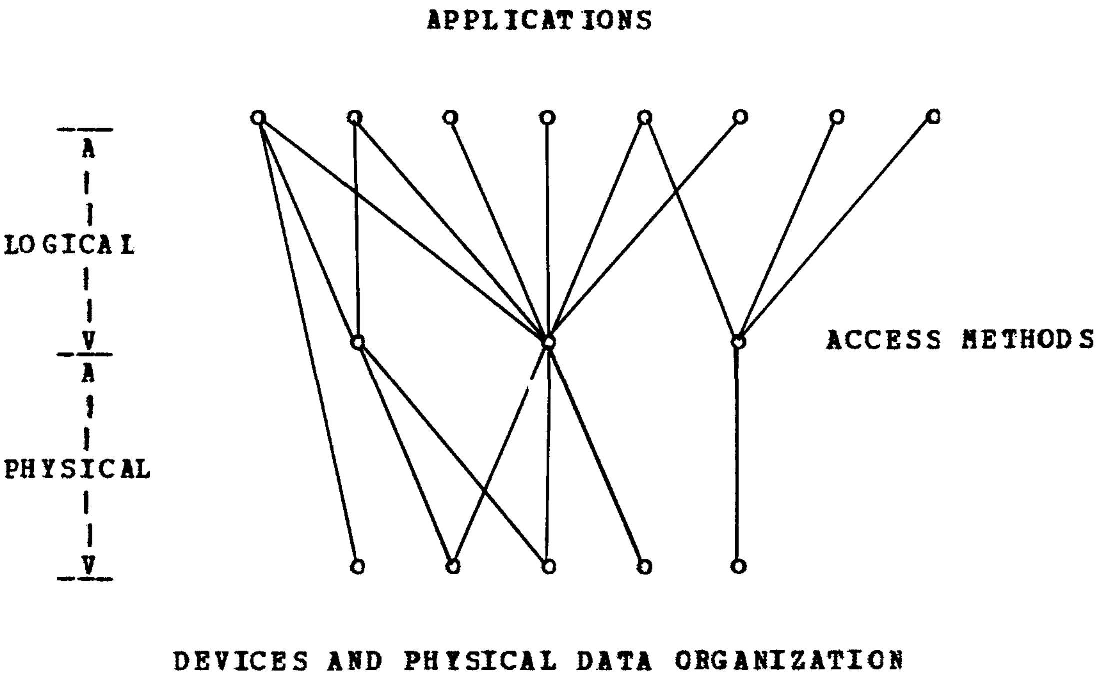

Figure 2.3
Access Methods File Systems

(Analogous to Early Programming Languages,
Such as FORTRAN and COBOL)

In order to overcome the weakness in the access methods approach, it is necessary to design a single uniform file representation that can (1) be used for every application and (2) be device independent. This idealistic goal is analogous to the search for "The" universal programming language, for which PL/I is probably the most ambitious attempt to date.

It is reasonable to expect that such a uniform representation will be so atomic or primitive in form that it will be desirable to construct more powerful specialized access methods for the convenience of the typical user. Since the access methods are built upon the uniform representation, it is much easier to modify or implement new access methods or, if necessary, operate at the atomic level to bypass the restrictions of the access methods. This approach pushes the logical/physical separation of file system structure much further as indicated in Figure 2.4.

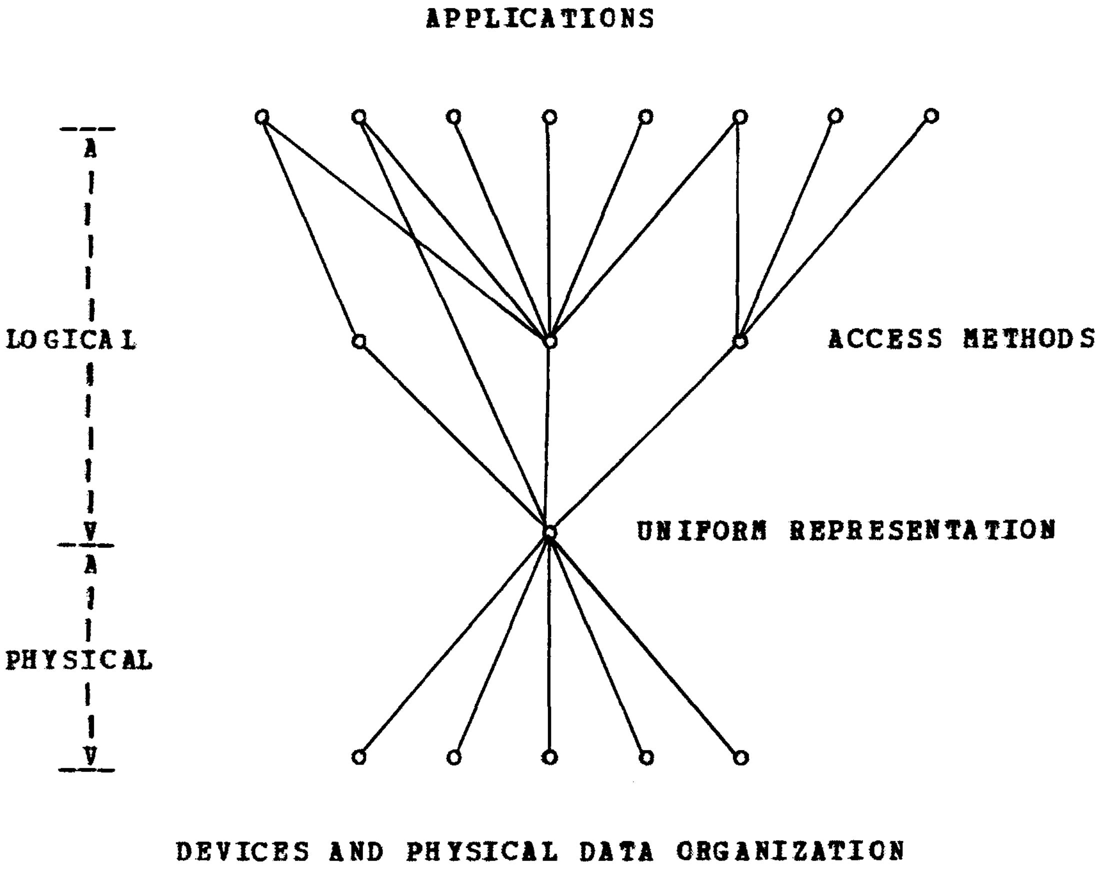

Figure 2.4
Uniform File Representation

(Analogous to Universal Programming Language,
PL/I ?)

The rationale behind the selection of a particular uniform representation is not trivial. For example, there are three broad classes of common uniform representaticns:

1. Stream - every file is treated as a continuous sequential stream of information. It is possible to access only the current position in the stream or reposition to the beginning of the stream. This representation can be implemented conveniently on almost all secondary storage devices, although it does not provide the user with very powerful or efficient features for many applications.
2. Direct-Access - every file is treated as an ordered collection of items. Each item is directly accessable by means of a unique identifier corresponding to its position in the ordering. This representation, which corresponds to primary storage organization, is more powerful than Stream, but is very difficult to implement on intrinsically serial devices, such as magnetic tape.
3. Associative - every file is treated as an unordered collection of items, each item is directly accessible by means of an identifier that has been "associated" with the item. This is a very flexible representation. Unfortunately, except for a small class of sophisticated

secondary storage devices the implementation is very complex and inefficient.

Irregardless of the specific uniform representation chosen, the important concept is that all files can be viewed as being identical in structure independent of the particular physical device on which the file is recorded. This generalization is depicted in Figure 2.5, which should be compared with Figure 2.1.

```
                                          +----------+
                                     +--->| UNIFORM  |
                                     |    | FILE 1   |
                                     |    +----------+
                                     |
                                     |
+----------+         +----------+    |    +----------+
| PRIMARY  |<=======>| CENTRAL  |<---+--->| UNIFORM  |
| STORAGE  |         |PROCESSOR |    |    | FILE 2   |
+----------+         +----------+    |    +----------+
                                     |         .
                                     |         .
                                     |         .
                                     |         .
                                     |    +----------+
                                     +--->| UNIFORM  |
                                          | FILE m   |
                                          +----------+
```

Figure 2.5
Logical Computer Configuration

Hierarchy of Logical Transformations

Although a precise description of a file system will not be presented until later sections, there are several general characteristics of most file systems. In particular, a user specifies his request, such as read or write, by designating a file and an element within the file. Most advanced file systems allow considerable flexibility in the mechanism used to specify a file, it is typically described by means of a symbolic file name. Furthermore, the element within the file is specified in terms of the logical representation of elements in the particular file system which may or may not correspond to a precise physical specification of how and where the element is stored. For example, a typical request might be of the form:

"Read item 23 from file ALPHA into location 1564."

Realizing that information must usually be stored on devices in somewhat obscure ways, there must be some sequence of transformations required to convert the user's request into its final form that physically operates on the secondary storage device. Quite often the transformation is viewed as a single step but that is a gross oversimplification that hides the fundamental mechanisms in use. In Figure 2.6 the conversion process is illustrated in terms of a discrete sequence of logical transformations.

Since the specifics of these transformations may not be

```
              FILE SYSTEM                ANALOGY

          READ ITEM 23 FROM    ||   SEND LETTER TO
          FILE ALPHA INTO      ||   JOHN DOE'S
          LOCATION 1564.       ||   HOME ADDRESS.
                               ||
                               ||
          SYMBOLIC             ||   JOHN DOE
          FILE NAME            ||        |
               |               ||        |
LFS            |               ||        |
               V               ||        V
          NUMERIC              ||   030-34-1234
          FILE IDENTIFIER      ||        |
               |               ||        |
BFS            |               ||        |
               V               ||        V
          FILE DESCRIPTOR      ||   Birth date
               |               ||   Office Address
               |               ||   Home address
               |               ||        etc.
               |               ||        |
FOSM           |               ||        |
               V               ||        V
          LOGICAL              ||   EXTRACT
          I/O COMMANDS         ||   HOME ADDRESS
               |               ||        |
DSM            |               ||        |
               V               ||        V
          PHYSICAL             ||   SEND TO
          I/O COMMANDS         ||   POST OFFICE
               |               ||        |
IOCS           |               ||        |
               V               ||        V
          I/O DEVICE           ||   POSTMAN DELIVERY
                               ||
```

Figure 2.6
Logical Transformations in File System

obvious until the more detailed sections later, a simple analogy is presented in Figure 2.6 that loosely parallels the file system transformations. The analogy is only intended to provide some insight into the rationale behind each stage of the transformation.

The process starts from the user's request to "read item 23 from the file ALPHA into location 1564". The first step is to convert the symbolic file name into a unique numeric file identifier. In the analogy, this corresponds to looking up John Doe's identifier which is a social security in this illustration. The purpose for using an identifier is basically the same in both cases. It is usually more convenient to store information, manually or automatically, by means of a unique numeric "key" rather than a symbolic name which may, under certain circumstances, not even be unique (i.e. there may be more than one John Doe in which case other factors must be considered in order to uniquely identify the person under consideration).

The file identifier can then be used to conveniently access all the information known about a file, this information collectively is known as the file's descriptor. In the analogy, this would correspond to requesting all information in the social security records of 030-34-1234.

Now that everything is known about the file, it is necessary to consider the specific operation to be performed. Using the file descriptor, a sequence of logical

I/O commands can be produced. These are called logical I/O commands because they do not consider the physical characteristics of the secondary storage device to be used. This is analogous to putting an address on a letter which is usually done without considering the physical destination nor the route to be taken.

In order to complete the transformation, the logical I/O commands must be converted into the appropriate sequence of physical I/O commands. This conversion may be trivial or complex depending upon the peculiarities of the device and I/O interfaces to the devices. In the analogy this process is performed at the post office where the address is used to determine the physical routing needed to get the letter to its destination.

The final step in the process is the physical transfer of information. This is usually performed by means of software/hardware interactions to activate the appropriate device and confirm the successful completion of the request. Of course, in the analogy this transfer is accomplished by the postman ("neither rain nor snow nor dark of night...") assisted by trucks, planes, trains and other automation.

CHAPTER THREE

File System Design Model

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<Dijks 67><Dijks 68> and Randell<Rand 68>. The important aspect of 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" or "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.

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

```
          |
          V
  +-------------+
  |  LEVEL K+1  |
  +-------------+
          |
          V
  +-------------+
  |   LEVEL K   |
  +-------------+
          |
          V
  +-------------+
  |  LEVEL K-1  |
  +-------------+
          |
          V
```

Figure 3.1
Hierarchical Levels

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 occuring in each potential "system state". 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 an important situation. In theory, 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"<Denn 65> or "named virtual memory"<Daley 68>. 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.

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<Lett 68><Daley 68><Denn 68><Salt 68> 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

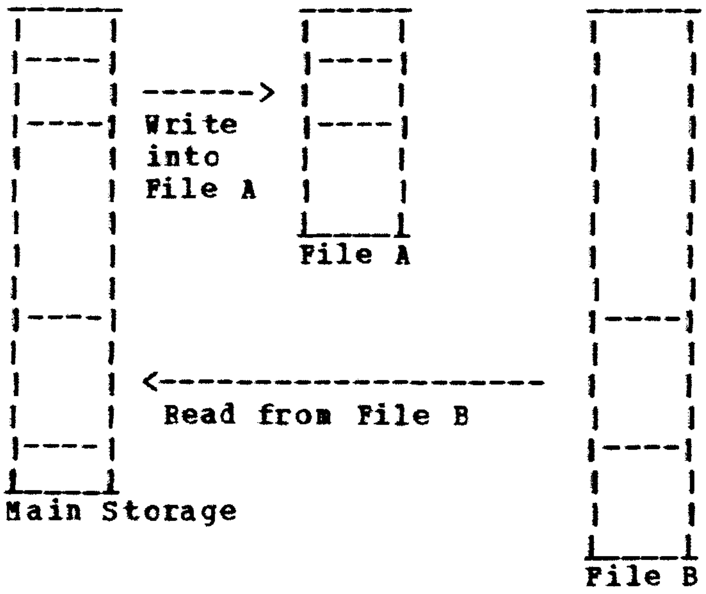

Figure 3.2
"Real" Memory and "Virtual" File Memory

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 tc 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. This strategy does not prevent the user from providing additional information, such as how often the file will be used and in what manner. The important factor is that this information is not necessary and its significance is determined by the file system rather than the user.

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 cther 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 seven 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 seven hierarchical levels are:

1. Input/Output Control System (IOCS)
2. Device Strategy Modules (DSM)
3. Allocation Strategy Modules (ASM)
4. File Organization Strategy Modules (FOSM)
5. Basic File System (BFS)
6. Logical File System (LFS)
7. 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 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.

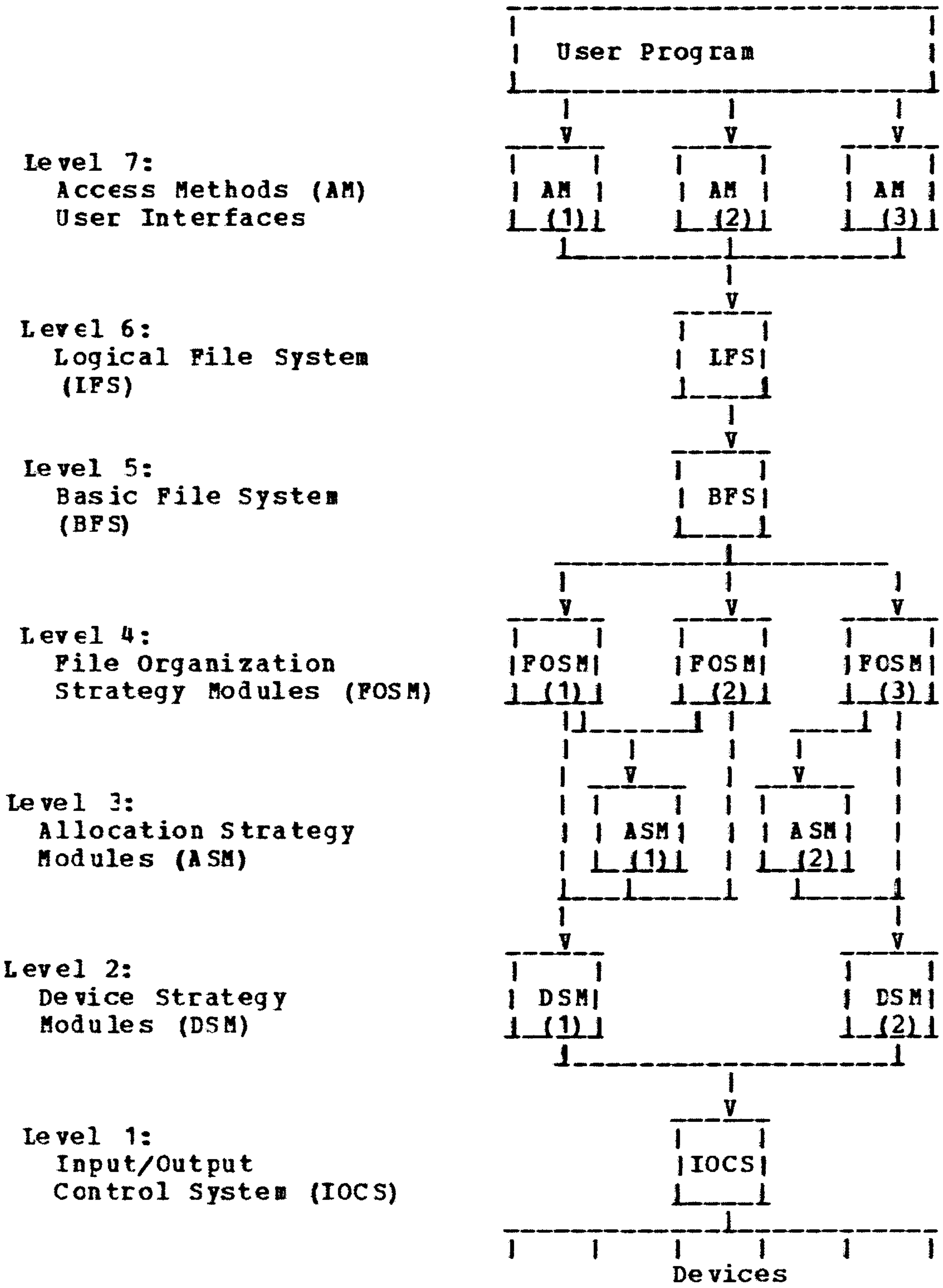

Figure 3.3
Hierarchical File System

```
                            +----------+
 PARAMETERS                 | USER OR  |     DATA BASES
                            | ACCESS   |
                            | METHODS  |
                            +----------+
  _                    _
 | FILE NAME            |        |
 | FILE ADDRESS         |        |
 | CORE ADDRESS         |        |
 | LENGTH               |        |
 |_FUNCTION            _|        V
                            +----------+
                            | L F S    |<===> NAME-->IDENTIFIER
                            +----------+
  _                    _
 | FILE IDENTIFIER      |        |
 | FILE ADDRESS         |        |
 | CORE ADDRESS         |        |
 | LENGTH               |        |
 |_FUNCTION            _|        V
                            +----------+
                            | B F S    |<===> IDENTIFIER-->
                            +----------+                DESCRIPTOR
  _                    _
 | FILE DESCRIPTOR      |        |
 | FILE ADDRESS         |        |
 | CORE ADDRESS         |        |
 | LENGTH               |        |
 |_FUNCTION            _|        V
                            +----------+
                            | F O S M  |<===> ADDRESS-->LOGICAL
                            +----------+                  I/O
  _                    _
 | DEVICE IDENTIFIER    |        |
 | LOGICAL I/O LIST     |        |
 | CORE ADDRESS LIST    |        |
 |_FUNCTION            _|        V
                            +----------+
                            | D S M    |<===> LOGICAL-->PHYSICAL
                            +----------+      I/O        I/O
                                 |
  _                    _
 | DEVICE ADDRESS       |        |
 |_I/O COMMAND LIST    _|        V
                            +----------+
                            | I O C S  |
                            +----------+
                                 |
                                 V
                            +----------+
                            | DEVICE   |
                            +----------+
```

Figure 3.4
Parameters and Data Bases Used by File System

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. If necessary, new records are allocated by the ASM. The list of records to be physically processed is passed on to the appropriate DSM.

Although not necessary, the FOSM is often designed to allocate "hidden" file buffers in order to minimize redundant or unnecessary I/O. 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 allocated 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").

Allocation Strategy Modules (ASM)

The Allocation Strategy Modules keep track of the available records on a device. They are responsible for allocating records for a file that is being created or

```
         _____          _____
        |     |        |     |
        |     | <----> |     | Record 4
        +-----+        |_____|         _____
        |     |                        |     |
        |     | <--------------------> |     | Record 7
        +-----+          _____         |_____|
        |     |         |     |
        |     | <-----> |     | Record 14
        +-----+         |_____|         _____
        |     |                        |     |
        |     | <--------------------> |     | Record 2
        |_____|                        |_____|

File Virtual Memory       Physical Records
```

Figure 3.5
Mapping Virtual Memory Into Physical Records

expanded, and deallocating records for a file that is being erased or truncated. The FOSM requests that a record be allocated when needed, the ASM actually selects the record.

Quite frequently, the ASM functions are incorporated into either the FOSM or DSM. In this paper these functions will be kept as separate as possible by explicitly recognizing the separate ASM level.

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 and ASM requests.

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<Dixon 67>, GIM-1<Nel 67>, and TDMS<Blei 67>. 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 paper 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 length, L.

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.

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

```
 _____________
|           | 0
| Record 1|
+----------+__
|           | L
| Record 2|
+----------+__
|           | 2L
| Record 3|
+----------+__
|           | 3L
| Record 4|
|__________:__
             4L
```

Figure 3.6
Layout of Virtual Memory For Fixed-Length
Record Access Methods

```
 ____________
| L1        | 0
+----------+__
|          | 4
| Record  1|
+----------+__
| L2       | L1+4
+----------+__
|          | L1+8
| Record  2|
+----------+__
| L3       | L1+L2+8
+----------+__
|          | L1+L2+12
| Record  3|
|__________|__
             L1+L2+L3+12
```

Figure 3.7
Layout of Virtual Memory for Variable-Length
Record Access Method

```
     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. For example, Nelson<Nel 65> discusses some flexible and complex file structures that can be used "as an adjunct to creativity".

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 him questionable 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 can be treated as a standard file if its 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 identifers for directory files as well as data files. Each subsequent directory file can contain file identifers for other directory files as well as data files.

Figure 3.8 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 3.8 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 facilities for

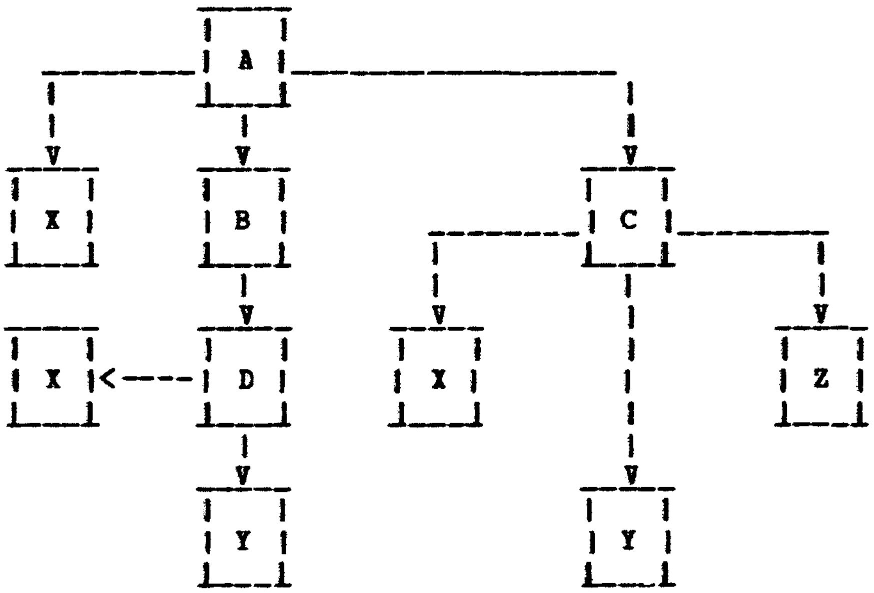

Figure 3.8
Hierarchical File Directory Example

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<Daley 65>. 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 identifer. The precise locations of the file descriptors can differ for different implementations, but uniquely defined by the identifer. 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 tc locate the file descriptor and place it in the Active File Directory unless it has already been "opened". The Basic File System also checks that the action requested upon the file such as read, write, cr 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 file 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 ervironment, 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 reccmmended 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 large, 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.

Allocation Strategy Modules

When the FOSM maps a valid write request onto a logical record for which a physical record has not been allocated, the ASM 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 ASM 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 with the Allocation Strategy Modules to optimize overall performance. The precise nature of meaningful cooperation would be too detailed to discuss in this paper.

Device Strategy Modules

In addition to the obvious "read" and "write" functions, direct access devices often require additional I/O commands, such as "seek" and "search", for proper positioning. The FOSM and ASM deal only with the logical act of of reading and writing. They transfer 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 the 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 record 49.

Input/Cutput 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 toc 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 cr 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.

CHAPTER FOUR

Multi-Computer Network Environment

Background

A general file system design model must, of course, be modified and elaborated to satisfy the needs of any specific desired file system environment. To illustrate the refinement process, a unique file system design will be presented for a multi-computer network.

Multi-computer networks are becoming an increasingly important area of computer technology<Mad 68>. There are several significant reasons behind the growth of multi-computer networks:

1. To increase the power of a computer installation in a modular manner, especially if (a) it is not possible to acquire a larger processor, (b) reliability is important, or (c) there are real-time or time-sharing constraints.
2. To serve the co-ordination requirements of a network of regional computer centers.
3. To support the accessibility to a nation-wide data base.

An example of the environment to be considered for this paper can be illustrated in Figure 4.1. This type of multi-computer network has been in limited use for several years in many configurations. The IBM 7094/7044 Direct-Coupled System<Rosen 69> was probably cne cf the earliest practical examples of such an inter-connected arrangement.

There are several implicit constraints imposed upcn the multi-computer system illustrated in Figure 4.1:

1. Independence of Central Processors.

   Each of the central processors operate independently such that there are no direct processor-to-processor data transfer nor signaling, and furthermcre there is no "master" processor.

2. Non-shared Memory.

   Each central prccessor has its own main storage unit. These units are not shared with nor accessed by another central processor.

3. Inter-locked Device Ccntrollers.

   The device controllers act as "traffic cops" to the actual I/O direct access devices. They control the traffic between a computer's I/O channel and a selected I/O device. A single device contrcller will only accept requests from one channel at a time and will only select one I/O device (among those under its control) at a time. Once a device controller

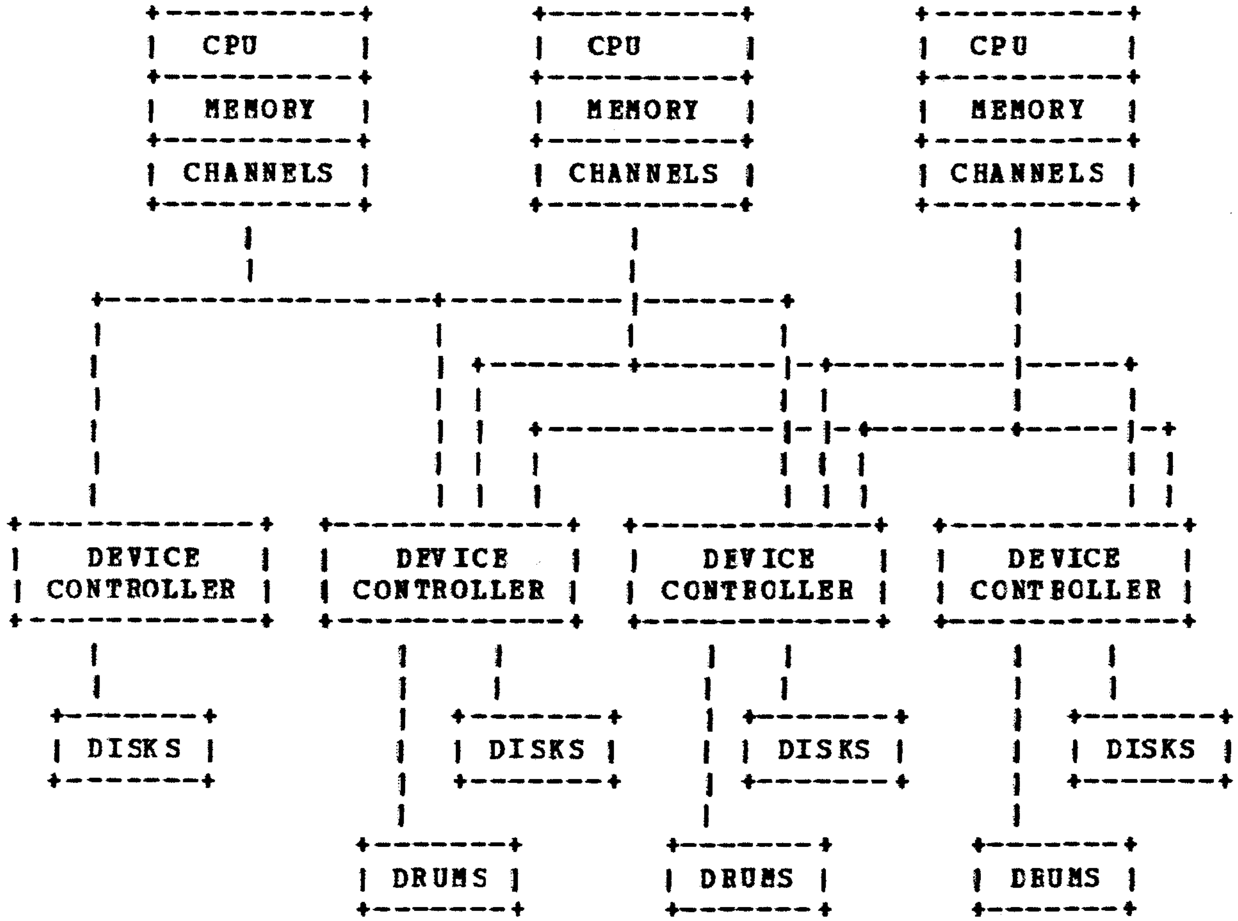

Figure 4.1
Example of Multi-computer File System Network

connects a channel with a device, the connection remains intact until the channel releases the device or an I/O error occurs.

The environment described above, although well within the boundaries of current technology, has not been the subject of much investigation. Such configurations are presently very expensive and, therefore, chosen only for very specialized situations. Even then there are only two or three processors and very specialized software and operational factors. A discussion of the CP-67/CMS Time Sharing System <IBM 68a><Sea 68> will serve to establish the relevance of the multi-computer network environment.

The CP-67/CMS Time Sharing System uses the special hardware features of a single IBM System/360 model 67 processor augmented by software to produce an apparant environment corresponding to the multi-computer network illustrated in Figure 4.1, with many independent central processors, device controllers, and direct access I/O devices. In practice a typical single processor 360/67 configuration would produce the affect of about 30 active processors ("virtual" System/360 model 65 processors each with a 256,000 byte memory) and 50 active device controllers. More detailed descriptions of the CP-67/CMS System can be found in the References. In the traditional sense of time-sharing, each user of the CP-67/CMS System is provided with a "virtual" computer operated from a simulated operatcr console (actually an augmented remote terminal). Most importantly, each "virtual" computer (i.e. user) operates logically independently of all other "virtual" computers except for the specified inter-connected I/O devices and device contrcllers.

Problems Arising In Multi-Computer Networks

There are many problems associated with the multi-computer file system network. Some of these problems are unique to this environment. Other problems have been solved in traditional file systems<Corb 62><Salt 65><Scie 68>, but the solutions require major revisions due to the peculiarities of the environment. The most significant problems are listed briefly below.

1. No shared memory.

   Usually file systems co-ordinate the status of the files and devices by using main storage accessable tables and data areas that describe file status, access rights, interlocks, and allocation. There is no such common communication area in main storage that can be accessed by all the independent processors.

2. No inter-computer communication.

   Multi-computer configurations usually provide a mechanism for sending signals or data transfers between the separate processors. With this capability the non-shared memory problem could be solved by either (a) electing one processor to be the "master" processor that coordinates the other processors, or (b) supply all the processors with enough information such that each processor knows what all the other processors are doing. The concept of a "master" processor opposes the intended homogeneous, independent processor assumption. The possibility of supplying status information to all other processors, although reasonable for a three or four processor configuration, was not considered a feasible solution for a system with hundreds of processors and devices and thousands of files. For these reasons, inter-computer communication, although an available capability, was not included as a required capability of the multi-computer environment described above.

3. No pre-arranged allocations.

   For small specialized multi-computer file networks, each processor can be "assigned" a specific area of a device or set of devices that can be used to write new files, all other processors can only read from this area by convention. This prevents the danger of two independent processors writing files at the same place. Such an "arrangement" is not practical for a large, flexible multi-computer file network since the static assignment of secondary storage space does not take account of the dynamic and unpredictable requirements of the independent processors.

4. Extendable device and file allocation.

The number of devices and sizes of devices as well as the number and sizes of files are, within reason, unlimited. For example, a specific amount of secondary storage equivalent to 100,000 card images could be used to hold 10 files of 10,000 card each or 1,000 files of 100 cards each. This consideration discourages techniques that result in a strong efficiency or main storage capacity dependency on the "size and shape" of the file system. Of course, the magnitude of the file system size will affect the operation, but arbitrary restrictions such as "no more than 64 files on a device" would be discouraged unless essential.

5. Removable volumes.

It has become common to differentiate between the I/O mechanism used to record or read information, called a "device", and the physical medium on which the information is stored, called a "volume". For most drums and many disk units, the device and volume are inseparable. But, for magnetic tape units and many of the smaller disk units the volume, magnetic tape reel and disk pack respectively, are removable. It is intended that the file system include files that are on unmounted volumes (disconnected from an I/O device) as well as mounted volumes. Therefore, a configuration that consists of ten disk units may have a file system that encompasses hundreds of volumes, only ten of which may be actively in use at a time. Since removing and mounting a volume takes several minutes of manual effort, it will be assumed that the "working set" of volumes (volumes that contain files that are actively in use) remains static for reascnable periods of time and is less than or equal tc the number of devices available. The fact that volumes are removable and interchangeable (i.e. may be mounted cn different devices at different times) does affect the organization of the file system. For example, a scheme that involved linking files together by means of pointers (chained addressing) could require mounting volumes just to continue the path of the chain even though little or no "lcgical" information was requested from files on that volume. In the worst case, it might be necessary to mount and unmount all the volumes of the file system to locate a desired file. Such a situation should definitely be avoided if not totally eliminated by the file system.

6. Structured file directories and file sharing.

In a traditional file system, the mapping between the symbolic file name and the corresponding file was accomplished by means of a single Master File

Directory. For modern file systems with thousands of files scattered over hundreds of volumes, it became desirable, if not necessary, to form groupings of files by means of Secondary File Directories<Daley 65>. These groupings are often used by the system to associate users with files they own (User File Directories). This capability is also available to the user to arrange his files into further sub-groups (libraries) or into separate project-related groupings. Occasionally it becomes necessary for a file to be included in two or more groupings (e.g. accessible by more than one User File Directory) with potentially different access privileges (protection) associated with each grouping. Many of these features that are relatively easy to implement in a traditional file system are complicated by the introduction of independent processors and removable volumes.

7. Fail-safe operation.

Reliable operation is a very important requirement of a general purpose file system. There are many known techniques for I/O error and systematic backup and salvage procedures that are applicable to this environment. The important problem associated with the multi-computer network is that potential error conditions exist that are not normally found in traditional single computer file systems. For a single computer system, a processor error (including unexpected processor disconnection, i.e. "turning off") is a rare occurrence. Such a situation is remedied by repairing whatever physical hardware is necessary and then running a special "salvager" program tc bring the file system into a well-defined operational state. In the environment of a multi-computer network, processors may be connected or disconnected at any time without any awareness by the other processors. To prevent any inconsistent file system operation by the other processors and eliminate the need for usually time-consuming salvage techniques, it is necessary to keep the file system in a well-defined consistent state at all times.

A File System Design

The purpose of the remainder of this paper is to apply the organization presented in the File System Design Model section to solve the problems associated with a multi-computer file system network. Discussion of the Access Methods and Input/Output Comtrol System will be omitted. This is necessitated for brevity and consideration of the facts that the Access Methods are highly application oriented, as discussed in a previous section, and that the Input/Output Control System is usually a basic and common component of all Operating Systems. The principal contribution of this model lies in the structure of the five other levels.

Logical File System

To present the goals and requirements of the Logical File System in a brief and demonstrative manner, an example will be used. The reader should refer to Figure 4.2 for the following discussion. It is important that the peculiarities of the example, such as the choice of file names (e.g. "FILE6" and "DIR4"), not be confused with the general characteristics of the Logical File System.

In Figure 4.2, there are 12 files illustrated. Associated with each file is an identifier of the form "VOL1(3)". The usage of this identifier will not be

```
       VOLUME "VOL1"     VOLUME "VOL2"      VOLUME "VOL3"


        (User 1)
       +----------+
       |/VOL1(3) /|
       +----------+   +-------------------------------------+
       | FILE3    |---+   (User 2)                          |
       +----------+      +----------+                       |
       | DIR2     |----->|/VOL2(3) /|<---+   (User 3)       |
       +----------+      +----------+    |  +----------+    |
+------| FILE2    |      | DIR3     |----+->|/VOL3(2) /|    |
|      +----------+      +----------+    |  +----------+    |
| +----| FILE4    | +----| FILE5    |    +--| DIR4     |    |
| |    +----------+ |    +----------+       +----------+    |
| | +--| FILE1    | | +--| FILE6    |    +--| DIR3     |    |
| | |  +----------+ | |  +----------+    |  +----------+    |
| | |               | |                  |  | FILE8    |--+ |
| | |               | |                  |  +----------+  | |
| | |  +----------+ | |  +----------+    |                | |
| | +->|/VOL1(2) /| | +->|/VOL2(4) /|    |                | |
| |    |//////////| |    |//////////|    |  +----------+  | |
| |    +----------+ |    +----------+    +->|/VOL3(5) /|  | |
| |                 |                       +----------+  | |
| |                 |                +------| FILE6    |  | |
| |    +----------+ |    +----------+|      +----------+  | |
| +--->|/VOL1(6) /| +--->|/VOL2(2) /|| +----| FILE2    |  | |
|      |//////////|      |//////////|| |    +----------+  | |
|      +----------+      +----------+| | +--| FILE7    |  | |
|                                    | | |  +----------+  | |
|                                    | | |                | |
|      +----------+                  | | |                | |
+----->|/VOL 1(4)/|                  | | |  +----------+  | |
       |//////////|                  | | +->|/VOL3(4) /|<-+<+
       +----------+                  | |    |//////////|
                                     | |    +----------+
                                     | |
       +----------+                  | |
       |/VOL1(5) /|<-----------------+ |    +----------+
       |//////////|                    +--->|/VOL3(3) /|
       +----------+                         |//////////|
                                            +----------+
```

Figure 4.2
Example of File Directory Structure (to LFS)

discussed until later, in the meanwhile notice that each file's identifier is unique. The 12 files are divided into 2 types, directory files (i.e. VOL1(3), VOL2(3), VOL3(2), and VOL3(5)), and data files (i.e. VOL1(2), VOL1(6), VOL1(4), VOL1(5), VOL2(4), VOL2(2), VOL3(4), and VOL3(3)). The distinction between directory files and data files is only a matter of usage, the Access Methods may operate upcn a directory file in the same manner as a data file, furthermore, all lcwer levels (e.g. Basic File System) treat all files as data files. This factor will be elaborated shortly.

It is the stated function of the Logical File System to map a file name reference into a unique file identifier. This mapping is a function of the requested file name (symbolic file name path) and a starting point (base directory) in the file directory structure. In Figure 4.2, threе example base directories are illustrated by associating VOL1(3) with user 1, VOL2(3) with user 2, and VOL3(2) with user 3. Therefore, user 1 references to the file name FILE2 yields the file VOL1(4).

A more complex example can be illustrated by considering the file VOL3(4). User 3 can refer to this file under the name FILE8. Alternatively, it can be referenced by the name DIR3.FILE7. The file DIR3, which is associated with VOL3(5) from user 3's base directory, is interpreted as a lower level directory. Then from file VOL3(5), the file name

FILE7 is mapped into VOL3(4) as intended. The file VOL3(4) can be referenced from user 2's base directory as DIR3.FILE8 or DIR3.DIR3.FILE7, for example. From user 1's base directory, the file VOL3(4) can be referenced as FILE3, DIR2.DIR3.FILE8, DIR2.DIR3.DIR3.FILE7, or even DIR2.DIR3.DIR4.DIR3.DIR3.FILE7.

Two important side affects of the base file directory and file name path facilities are that (1) a specific file may be referenced by many different names, and (2) the same name may be used to reference many different files.

The headings VOLUME "VOL1", VOLUME "VOL2", and VOLUME "VOL3" are intended to indicate that the 12 files are scattered over 3 separately detachable volumes: VOL1 (containing VOL1(2), VOL1(3), VOL1(4), VOL1(5), and VOL1(6)), VOL2 (containing VOL2(2), VOL2(3), and VOL2(4)), and VOL3 (containing VOL3(2), VOL3(3), VOL3(4), and VOL3(5)). If volume VOL2 were detached from the system, user 1 could still reference VOL1(4) as FILE4 and VOL3(4) as FILE3, but could not reference VOL3(4) as DIR2.DIR3.FILE8 nor VOL1(5) as DIR2.DIR3.DIR3.FILE6 since the path would logically require passing through volume VOL2. Furthermore, user 3 is allowed to erase (i.e. remove from file system structure) the file VOL3(4) under the name FILE8, assuming appropriate protection priviledges, _whether or not_ volume VOL1 is mounted in spite of user 1's reference to file VOL3(4) under the name FILE3.

The Logical File System could be extremely complex if it had to specifically consider the physical addresses of volumes, the device characteristics, and the location of file directories on volumes, in addition to its obvious requirement of searching file directories. These problems are eliminated by introducing the file identifier and the interface with the Basic File System.

The Basic File System processes requests that specify a file in terms of a file identifier consisting of a volume name and index, such as (VOL3,4), rather than a file name. A sample call from the Logical File System to the Basic File System, in PL/I-like notation, is:

```
CALL BFS_READ(VOLUME,INDEX,CORE_ADDR,FILE_ADDR,COUNT);
```

where VOLUME is the name of the volume containing the file, INDEX is the corresponding unique index of the file, CORE_ADDR is the main storage address into which data is to be read, FILE_ADDR is the file virtual memory address from which the data is to be read, and COUNT is the number of bytes to be transmitted. Using these features, the <u>heart</u> of the logical File System (ignoring opening and closing files, file access protection, illegal file names, etc.) reduces to the PL/I-like code presented in Figure 4.3. It is assumed that the file name has been broken down into an array of path element names (e.g. if name is DIR2.DIR3.FILE8, then PATH(1)='DIR2', PATH(2)='DIR3', PATH(3)='FILE8', and PATH_LENGTH=3), that BASE_VOLUME and BASE_INDEX initially specify the (VOLUME,INDEX) identifier of the base directory, and that each entry in a file directory is N bytes long and formatted as indicated in the FILE_ENTRY declaration.

For efficiency, the names of all files that are actively in use (usually a small fraction of all files in the system) are kept in main storage in an Active Name Directory (AND). The AND is searched before accessing the file directories on secondary storage. Entries are deleted from the AND when the corresponding file is "closed" or "deleted".

Of course, the handling of access (protection) rights, errors, and other responsibilities will make the Logical File System much more complex, but it is important to note that the design and implementation of the Logical File System escapes all physical file organization and device characteristic considerations and complexities.

```
DECLARE 1 FILE_ENTRY,
        2 FILENAME CHARACTER (8),
        2 VOLUME   CHARACTER (8),
        2 INDEX    FIXED BINARY,
              . . .
              . . .
DO I = 1 TO PATH_LENGTH;
   DO   J = 0 BY N WHILE (FILE_ENTRY.FILENAME ¬= PATH(I));
   CALL BFS_READ(BASE_VOLUME,BASE_INDEX,FILE_ENTRY,J*N,N);
   END;
   BASE_VOLUME = FILE_ENTRY.VOLUME;
   BASE_INDEX = FILE_ENTRY.INDEX;
END;
              . . .
              . . .
```

Figure 4.3

Example Procedure to Perform Logical File System Search

Basic File System

The Basic File System must convert the file identifier supplied from the Logical File System into a file descriptor than can be processed by the File Organization Strategy Module. A file descriptor contains information such as the volume name, physical location of the file on the volume, and the length of the file. Every file must have an associated file descriptor, but since the number of passive files (i.e. not actively in use) might be very large, the file descriptors are maintained on secondary storage until needed (i.e. file is "opened"). In organizing the secondary storage maintenance of the file descriptors there are several important considerations:

1. There must be a unique file descriptor for each file regardless of how often the file appears in file directories or what symbolic names are used. This is required to maintain consistent interpretation of a file's status.
2. The file descriptor information for a file must reside on the same volume as the file. This is reasonable since if either the file or its descriptor is not accessable at some time by the system (i.e. unmounted) the file cannot be used, this possibility is minimized by placing them on the same volume.

3. In the same manner that the Logical File System was simplified by using the facilities of the lower hierarchical level, the file descriptors should be maintained in a manner that allows the File Organization Strategy Module to process them as normal files.

These problems are solved by the use of the Volume File Descriptor Directory (VFDD). There is a single VFDD for each volume, it contains the file descriptors for all files residing on the volume. The file descriptors are of fixed length and are located within the VFDD positionally according to the corresponding file identifier's index. In order to exploit the facilities provided by the File Organization Strategy Module, the VFDD can be processed by the lower levels as a normal file. It is assigned an unique file identifier consisting of the volume name and an index of 1, in fact the file descriptor for a VFDD is stored (when not in use) as its own first entry. Figure 4.4 presents diagrammatically the logical file structure of Figure 4.2 with the added detail of the Volume File Descriptor Directories and File Directory formats.

For efficiency, the descriptor's of all files that are actively in use are stored in an Active File Directory (AFD). The AFD is searched before accessing the Volume File Descriptor Directory.

```
    +----------------+
    |   +---------+  |    +---------+     +-------------------+
    +-->|/VOL1(1)/|  | +->|/VOL1(2)/|  +->|     /VOL1(3)/     |
        +---------+  | |  |/////////|  |  +-------------------+
VOL1(1) | >>>>>>> |--+ |  +---------+  |  | FILE3   | VOL3(4) |
        +---------+    |             |  +---------+---------+
VOL1(2) | >>>>>>> |----+             |  | DIR2(D) | VOL2(3) |
        +---------+                  |  +---------+---------+
VOL1(3) | >>>>>>> |--------------------+  | FILE2   | VOL1(4) |
        +---------+                     +---------+---------+
VOL1(4) | >>>>>>> |--------------------+  | FILE4   | VOL1(6) |
        +---------+                    |  +---------+---------+
VOL1(5) | >>>>>>> |----+  +---------+  |  | FILE1   | VOL1(2) |
        +---------+    +->|/VOL1(5)/|  |  +---------+---------+
VOL1(6) | >>>>>>> |--+    |/////////|  |
        +---------+  |    +---------+  |
    VFDD for "VOL1"  |                 |
                     |    +---------+  |  +---------+
                     +--->|/VOL1(6)/|  +->|/VOL1(4)/|
                          |/////////|     |/////////|
                          +---------+     +---------+

    +----------------+
    |   +---------+  |    +---------+     +-------------------+
    +-->|/VOL2(1)/|  | +->|/VOL2(3)/|  +->|     /VOL2(3)/     |
        +---------+  | |  |/////////|  |  +-------------------+
VOL2(1) | >>>>>>> |--+ |  +---------+  |  | DIR3(D) | VOL3(2) |
        +---------+    |             |  +---------+---------+
VOL2(2) | >>>>>>> |----+             |  | FILE5   | VOL2(2) |
        +---------+                  |  +---------+---------+
VOL2(3) | >>>>>>> |--------------------+  | FILE6   | VOL2(4) |
        +---------+                     +---------+---------+
VOL2(4) | >>>>>>> |----+  +---------+
        +---------+    +->|/VOL2(4)/|
    VFDD for "VOL2"       |/////////|
                          +---------+

    +----------------+
    |   +---------+  |                    +-------------------+
    +-->|/VOL3(1)/|  | +----------------->|     /VOL3(2)/     |
        +---------+  | |                  +-------------------+
VOL3(1) | >>>>>>> |--+ |                  | DIR4(D) | VOL2(3) |
        +---------+    |                  +---------+---------+
VOL3(2) | >>>>>>> |----+                  | DIR3(D) | VOL3(5) |
        +---------+       +---------+     +---------+---------+
VOL3(3) | >>>>>>> |------>|/VOL3(3)/|     | FILE8   | VOL3(4) |
        +---------+       |/////////|     +---------+---------+
VOL3(4) | >>>>>>> |----+  +---------+
        +---------+    |
VOL3(5) | >>>>>>> |--+ |                  +-------------------+
        +---------+  | |               +->|     /VOL3(5)/     |
    VFDD for "VOL3"  | |  +---------+  |  +-------------------+
                     | +->|/VOL3(4)/|  |  | FILE6   | VOL1(5) |
                     |    |/////////|  |  +---------+---------+
                     |    +---------+  |  | FILE2   | VOL3(3) |
                     |                 |  +---------+---------+
                     |                 |  | FILE7   | VOL3(4) |
                     +-----------------+  +---------+---------+
```

Figure 4.4
Example of File Directory Structure (to BFS)

The File Organization Strategy Module processes requests that specify a file in terms of a file descriptor (the entry extracted from the VFDD) rather than a file name or file identifier. A sample call from the Basic File System to the File Organization Strategy Module, in PL/I-like notation, is:

```
CALL FOSM_READ(DESCRIPTOR,CORE_ADDR,FILE_ADDR,COUNT);
```

where CORE_ADDR, FILE_ADDR, and COUNT have the same interpretation as discussed above.

The primary function of the Basic File System reduces to the single request:

```
CALL FOSM_READ(VFDD_DESCRIPTOR,DESCRIPTOR,M*(INDEX-1),M);
```

where VFDD_DESCRIPTOR is the descriptor of the VFDD associated with the volume name supplied by the Logical File System as part of the file identifier, INDEX is from the specified file identifier, M is the standard length of a VFDD entry, and DESCRIPTOR is the desired file descriptor.

The Basic File System performs several other tasks, such as protection validation and maintenance of the core-resident Active File Directory that enables efficient association between a file's identifier and descriptor for files that are in use (i.e. "open"). But, as in the Logical File System, the domain of the Basic File System is sufficiently small and narrow that it remains a conceptually simple level in the hierarchy.

File Organization Strategy Modules

The Logical File System and Basic File System are, to a great extent, application and device independent. The File Organization Strategy Modules are usually the most critical area of the file system in terms of overall performance, for this reason it is expected that more than one strategy may be used in a large system. Only one strategy will be discussed in this section, the reader may refer to the papers listed in the References<Corb 62><Mad 68b><Salt 65><Scie 68> for other possible alternatives.

The FOSM must map the logical file address onto a physical record address or hidden buffer based upon the supplied file descriptor information. In the simplest case, the mapping could be performed by including a two-part table in the file descriptor. The first part of each entry would indicate a contiguous range of virtual file addresses, the second part of each entry would designate the corresponding physical record address. It has been assumed, however, that all file descriptors have a specific length, whereas the mapping table is a function of the file's length and is potentially quite large. Therefore, it is not feasible to include the entire mapping table as part of the file descriptor. One of the most powerful file organization strategies utilizes file maps, Figure 4.5 illustrates such an arrangement.

```
                 0
+-----------+    +-----------+
|      |>>>>|--->|           |
|      +----|    +-----------+
|           |    |     .     |
+-----------+    |     .     |
Descriptor       |     .     |
                 +-----------+
              999|           |
                 +-----------+
                                                   0
+-----------+    +-----------+                      +-----------+
|      |>>>>|--->| >>>>>>>   |--------------------->|           |
|      +----|    +-----------+                      +-----------+
|           |    |     .     | 499,000              |     .     |
+-----------+    |     .     |    +-----------+     |     .     |
Descriptor       |     .     | +->|           |     |     .     |
                 +-----------+ |  +-----------+     +-----------+
                 | >>>>>>>   |-+  |     .     |  999|           |
                 +-----------+    |     .     |     +-----------+
                                  +-----------+
                           499,999|           |
                                  +-----------+
+-----------+    +-----------+                      +-----------+
|      |>>>>|--->| >>>>>>>   |--------------------->| >>>>>>>   |--+
|      +----|    +-----------+                      +-----------+  |
|           |    |     .     |    +-----------+     |     .     |  |
+-----------+    |     .     | +->| >>>>>>>   |     |     .     |  |
Descriptor       |     .     | |  +-----------+     |     .     |  |
                 +-----------+ |  |     .     |     +-----------+  |
                 | >>>>>>>   |-+  |     .     |     | >>>>>>>   |  |
                 +-----------+    |     .     |     +-----------+  |
                                  +-----------+                    |
                                  | >>>>>>>   |-+                  |
                                  +-----------+ |                  |
                                                |                  |
                               +----------------+                  |
                               |                                   |
                               |                +------------------+
                               | 249,999,000    | 0
                               |  +-----------+ |  +-----------+
                               +->|           | +->|           |
                                  +-----------+    +-----------+
                                  |     .     |    |     .     |
                                  |     .     |    |     .     |
                                  +-----------+    +-----------+
                       249,999,999|           | 999|           |
                                  +-----------+    +-----------+
```

Figure 4.5
Example of File Organization Strategy

In this example it is assumed that each file is divided into 1000 byte physical records. A file can be in one of several states depending upon its current length. If the file's length is in the range 1 to 999 bytes, the file descriptor contains the address of the corresponding physical record. If the file is between 1000 and 499,999 bytes long, the file descriptor specifies the address of a file map located on secondary storage. Each entry of the file map (assumed to require 2 bytes) designates the physical address of a block of the file (blocks are ordered by virtual file addresses: 0-999, 1000-1999, 2000-2999, etc.). Furthermore, for files greater than 500,000 bytes, but less than 250,000,000 bytes, there are 2 levels of file maps as illustrated.

This strategy has several advantages. Under the worst conditions of random access file processing only from one to three I/O operations need to be performed. By utilizing several hidden buffers for blocks of the file as well as file maps, the number of I/O operations required for file accesses can be drastically reduced.

Allocation Strategy Modules

The function of allocation and deallocation of blocks involves several separate factors. Before describing the implementation of the mechanisms, it is wise to review the desired characteristics:

1. A file is allowed to grow in size, the FOSM will request additional blocks from the ASM for the data portions of a file or its index tables, as needed.
2. Common direct access devices contain from 8000 to 32000 separately allocatable blocks, thus it is not feasible to store all allocation information in main storage.
3. Since two independent processors may be writing new files on the same volume at the same time, it is necessary to provide interlocks such that they do not accidently allocate the same block to more than one file, yet not require one processor to wait until the other processor finishes.

These problems can be solved by use of a special Volume Allocation Table (VAT) on each volume. In this scheme, a volume must be subdivided into arbitrary contiguous areas. For direct access devices with movable read/write heads, each discrete position (known as a "cylinder") covers an area of about 40 to 160 blocks. A cylinder is a reasonable unit of subdivision. For each cylinder on the volume, there is a corresponding entry in the VAT. Each entry contains a "bit map" that indicates which blocks on that cylinder have not been allocated. For example, if a cylinder consists of 40 blocks, the bit map in the corresponding VAT entry would be 40 bits long. If the first bit is a "0", the first block has not been allocated; if the bit is a "1", the block has already been allocated. Likewise for the second, third, and remaining bits.

When the FOSM first requests allocation of a block on a volume, the ASM selects a cylinder and requests that the DSM read the corresponding VAT entry into main storage. An available block, indicated by a "0" bit, is located and then marked as allocated. As long as the volume remains in use, the VAT entry will be kept in main storage and blocks will be allocated on that cylinder. When all the blocks on that cylinder have been allocated, the updated VAT entry is written out and a new cylinder selected. With this technique the amount of main storage required for allocation information is kept to a minimum (about 40 to 160 bits per volume), at the same time the number of extra I/O operations is minimized (about one per 40 to 160 blocks of allocation).

The problem of interlocking the independent processors still remains. As long as the processors are allocating blocks on different cylinders using separate VAT entries, they may both proceed uninterrupted. This condition can be accomplished by utilizing a hardware feature kncwn as "keyed records" available on several computers including the IBM System/360. Each of the VAT entries is a separate record consisting of a physical key area and a data area. The data area contains the allocaticn information described above. The key area is divided into two parts: the identification number of the processor currently allocating blocks on that cylinder and an indication if all blocks on that cylinder have been allocated. A VAT entry with a key of all zerces would identify a cylinder that was not currently in use and had blocks available for allocation.

There are I/O instructions that can be used by the DSM that will automatically search for a record with a specified key, such as zerc. Since the device controller will not switch processors in the midst of a continuous stream of I/O operations from a processor (i.e. "chained I/C commands"), it is possible to generate an uninterruptible sequence of I/O commands that will (1) find an available cylinder by searching the VAT for a entry with a key of zero and (2) change the key to indicate the cylinder is in use. This thus solves the multi-processor allocation interlock prcblem.

Device Strategy Modules

The Device Strategy Modules convert "logical I/O requests" from the File Organization Strategy Modules and Allocation Strategy Modules into actual computer I/O command sequences that are forwarded to the Input/Output Control System for execution.

When a request to transfer a large portion of a file (10,000 bytes for example) is issued, it is unlikely that a significant amount of the needed blocks are in hidden buffers. It will, therefore, be necessary to request I/O transfer for several blocks (e.g. about 10 blocks if each block 1000 bytes long). The FOSM will generate logical I/O requests of the form: "read block 227 into location 12930, read block 211 into location 13930, etc." The DSM must consider the physical characteristics of the device such as rotational delay and "seek" position for movable heads. It then decides upon an optimal sequence to read the blocks and generate the necessary physical I/O command sequence including positioning commands. The Input/Output Control System actually issues the physical I/O request, error retry, and other housekeeping as discussed earlier. The detailed strategy for choosing the optimal I/O sequence is, of course, very device dependent and will not be elaborated here.

Other Considerations

The preceeding sections have highlighted the framework of a file system. There are, of course, many other important decisions to be made in such a system, such as the format and organization of tables, error conditions<Lock 68>, measurement and accounting mechanisms, etc. One of the subtle points will be discussed in this section.

The Basic File System is intended to deal with files represented by unique identifiers. In the specific system presented, the identifier is designated as the tuple, <volume, index in VFDD>. This representation resulted in a very efficient mechanism for accessing a file's descriptor that avoided much of the time-consuming table lock-up. Unfortunately, this representation is not temporally unique. It has been assumed that when a file is deleted, the VFDD index position used for that file's descriptor is available for use by new files that may be created. This would not be a problem if all instances of the deleted file's identifier were removed from the system at the same time, but there may be more than one path to the file due to links from other symbolic file directories. The strategy used by the Basic File System did not provide any convenient means to locate all references (i.e. links) to a specific file. Furthermore, even if such a mechanism existed, it would not solve the problem since the reference may exist in a file directory that is located on a volume that is not physically mcunted or accessable by the system at the time of deleticn. Therefore, in such an environment, it is possible to have links in directories that identify files that have been deleted. The danger exists that the following sequence of events may occur: (1) a file is created and assigned identifier, <ALPHA,5>, (2) a link is made to that file, (3) the file is deleted by its creator, (4) a new file is created and coincidently assigned the identifier <ALPHA,5>, and (5) the link previously created is used not realizing that the intended file has been deleted and replaced by scme other arbitrary file!

Fortunately, this dilemna is not irrevocable, there is a multitude of solutions. Two simple variations would be (1) never reuse VFDD entries but allow the file to ccntinually grow but become "sparse" or (2) maintain ccunt of the number of links to a file and reuse the VFDD entry only when all links have been removed. A better solution can be formulated by attacking the original goal of generating truly unique file identifiers. The Multics Operating System has similar requirements, it forms unique identifiers by concatenating the central processor's unique serial number with the chronolog clock time with accuracy in the range of microseconds. A much simpler scheme can be incorporated into the file system by associating a separate counter with each volume. Whenever a new file is created on a vclume and assigned a VFDD entry, the value of the corresponding counter is incremented by one. For the purpose of the file system, the tuple, <volume, counter value>, is a unique identification of a file.

The counter value, which monotonically increases, cannot be efficiently used as a direct index into a finite size file descriptor directory. A minor modification to the Basic File System design can incorporate the ideas of the above discussion. The file identifier can be constructed from the triple, <volume, VFDD index, counter value>. In this context the counter value will be called a "key", since its sole purpose is to verify that the accessed VFDD entry is correct by attempting to "unlock" the entry (i.e. comparing the key from the VFDD entry with the key from the symbolic file directory which was copied from the VFDD when the link was initially established).

The above problems are typical of the factors that must be considered by file system designers. The general file system model will very seldom be a complete description of a specific implementation and it certainly will not replace the need for systems analysts, but it can save many months of the initial design!

CONCLUDING COMMENTS

Tc 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 primative 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 cffer "do-it-yourself FORTRAN kits". Since modern file systems often outweigh all cther 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 encugh to encompass most present file systems of advanced design and file systems presently planned, yet basic enough to be applicable tc 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.

REFERENCES

Bar 67 Barrow, D.W., Fraser, A.G., Hartley, D.F., Landy, B., and Needham, R.M., File Handling at Cambridge University, *Proceedings Spring Joint Computer Conference*, pp. 163-167, 1967.

Blei 67 Bleier, R. E., Treating hierarchical data structures in the SDC time-shared data management system (TDMS), *ACM National Conference Proceedings*, 1967.

Corb 62 Corbato, F. J., *et al*, *The Compatible Time-Sharing System*, MIT Press, Cambridge, 1962.

Daley 68 Daley, R. C., and Dennis, J. B., Virtual memory, processes and sharing in Multics, *Communications of the ACM*, May 1968.

Daley 65 Daley, R. C., and Neumann, P. G., A general purpose file system for secondary storage, *Proceedings Fall Joint Computer Conference*, 1965.

Denn 65 Dennis, J. B., Segmentation and the design of multi-programmed computer systems, *Journal of the ACM*, October 1965.

Dijks 67 Dijkstra, E. W., The structure of the 'THE' multiprogramming system, ACM Symposium on Operating Systems Principles, Gatlinburg Tennessee, October 1967.

Dijks 68 Dijkstra, E. W., 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.

Dixon 67 Dixon, P. J., and Sable, D. J., DM-1 - A generalized data management system, *Proceedings Spring Joint Computer Conference*, 1967.

Henry 69 Henry, W.R., Hierarchical structure for data management, *IBM Systems Journal*, Volume 8, No. 1, 1969.

IBM 68a IBM Cambridge Scientific Center, *CP-67/CMS Program Logic Manual*, Cambridge Massachusetts, April 1968.

IBM 68b IBM Corporation, *IBM System/360 Time Sharing System Access Methods*, Form Y28-2016-1, 1968.

Lett 68 Lett, Alexander S., and Konigsford, William L., TSS/360: a time-shared operating system, *Proceedings Fall Joint Computer Conference*, 1968.

Lock 68 Lockemann, Peter C., and Knutsen, W. Dale, Recovery of disk contents after system failure, *Communications of the ACM*, 1968.

Mad 68a Madnick, Stuart E., Multi-processor software lockout, *ACM National Conference Proceedings*, August 1968.

Mad 68b Madnick, Stuart E., Design strategies for file systems: a working model, FILE/68 International Seminar on File Organization, Helsingør Denmark, November 1968.

Mad 69 Madnick, Stuart E., Modular approach to file system design, *Proceedings Spring Joint Computer Conference*, 1969.

Nel 65 Nelson, T.H., A file structure for the complex, the changing and the indeterminate, *ACM National Conference Proceedings*, August 1965.

Nel 67 Nelson, D. B., Pick, R. A., and Andrews, K. B., GIM-1 - A generalized information management language and computer system, *Proceedings Spring Joint Computer Conference*, 1967.

O'N 67 O'Neill, R.W., Experience using a time-shared multi-programming system with dynamic address relocation hardware, *Proceedings Spring Joint Computer Conference*, 1967.

Rand 68 Randell, B., Towards a methodology of computer system design, Working Paper for the NATO Conference on Computer Software Engineering, Garmisch Germany, October 7-11 1968.

Rapp 68 Rappaport, R. L., Implementing multi-process primitives in a multiplexed computer system, S.M. Thesis, MIT Department of Electrical Engineering, August 1968.

Rosen 67 Rosen, Saul, *Programming Systems and Languages*, McGraw-Hill, New York, 1967.

Rosen 69 Rosen, Saul, Electronic computers: a historical survey, *ACM Computing Surveys*, Volume 1, No. 1, p. 24, March 1969.

Rosin 69 Rosin, Robert F., Supervisory and monitor systems, *ACM Computing Surveys*, Volume 1, No. 1, pp. 37-54, March 1969.

Salt 65 Saltzer, J. H., CTSS technical notes, MIT Project MAC Report MAC-TR-16, August 1965.

Salt 68 Saltzer, J. H., Traffic control in a multiplexed computer system, Sc.D Thesis, MIT Department of Electrical Engineering, August 1968.

Scher 65 Scherr, A. L., An analysis of time-shared computer systems, MIT Project MAC Report MAC-tr-18, June 1965.

Schw 64 Schwartz, Jules I., Coffman, Edward G., and Weissman, Clark, A general-purpose time-sharing system, Proceedings Spring Joint Computer Conference, 1964.

Schw 67 Schwartz, Jules I., and Weissman, Clark, The SDC time-sharing system revisited, ACM National Conference Proceedings, 1967.

Scien 68 Scientific Data Systems, SDS 940 Time-Sharing System Technical Manual, Santa Monica California, August 1968.

Sea 68 Seawright, L. H., and Kelch, J. A., An introduction to CP-67/CMS, IBM Cambridge Scientific Center Report 320-2032, Cambridge Massachusetts, September 1968.

Wilk 69 Wilkes, M.V., Time-Sharing Computer Systems, pp. 75-90, American Elsevier Publishing Company, Inc., New York, 1968.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
Design Strategies for File Systems

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
M.S. Thesis, Alfred P. Sloan School of Management and Dept. of Electrical Engineering

5. AUTHOR(S) *(Last name, first name, initial)*
Madnick, Stuart E.

6. REPORT DATE
October 1970

7a. TOTAL NO. OF PAGES
114

7b. NO. OF REFS
33

8a. CONTRACT OR GRANT NO.
Nonr-4102(01)

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
MAC TR-78 (THESIS)

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
3D-200 Pentagon
Washington, D.C. 20301

13. ABSTRACT

This thesis describes a methodology for the analysis and synthesis of modern general purpose file systems. The two basic concepts developed are (1) establishment of a uniform representation of a file's structure in the form of virtual memory or segmentation and (2) determination of a hierarchy of logical transformations within a file system. These concepts are used together to form a strictly hierarchical organization (after Dijkstra) such that each transformation can be described as a function of its lower neighboring transformation. In a sense, the complex file system is built up by the composition of simple functional transformations. To illustrate the specifics of the design process, a file system is synthesized for an environment including a multi-computer network, structured file directories, and removable volumes.

14. KEY WORDS

File Systems   Operating Systems   Modularity   Virtual Memory
Data Management   Programming