UNIVERSITY OF MANITOBA

THE DESIGN AND IMPLEMENTATION OF A STRUCTURED INDEXED FILE SYSTEM

by

Howard John Ferch

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

May, 1978

THE DESIGN AND IMPLEMENTATION OF A STRUCTURED INDEXED FILE SYSTEM

BY

HOWARD JOHN FERCH

A dissertation submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

This thesis discusses a file system specifically intended to manage source data files. The motivation for this file system was dissatisfaction with the facilities provided by existing file editing systems; the new file system is intended to provide a powerful set of file management facilities to allow for the construction of a more flexible editing system.

The two most important features of this file system are the ability to perform direct updating of data, and the management of hierarchies of files. Direct updating of data eliminates the copying of files required to perform editing functions on sequential files. Three major aspects to the hierarchical support are discussed. The use of a hierarchy allows for a subdivision of data in a way that matches the user's logical subdivision. In addition, the hierarchy is managed in such a way that an entire subsection of the hierarchy may be processed as a unit. The hierarchy provides a logical base for inherited attributes and password protection.

After the limitations of existing editors, and the facilities to be provided, are discussed, an actual implementation of such a file system is presented in three parts. First, the storage structures for providing the direct updating and file hierarchy capabilities are discussed. Secondly, the techniques used for maintaining integrity of the user data are presented. Finally, the performance level attained by such a file system is discussed and shown to be comparible to that of existing editors.

ACKNOWLEDGEMENTS

The author would like to express his gratitude to his advisor, Dr. C.R. Zarnke, for his many comments and suggestions during the production of this thesis and the software product upon which it is founded. Thanks are also due to the other examiners of this thesis, Dr. D.D. Cowan, Dr. R. J. Collens, and Dr. S. A. Bukhari, for the time they spent in reading the thesis, and for their suggestions for improvement.

Special thanks must be given to the many people who used the NAM file system during its development, and whose comments have helped to make it the successful product that it is.

The financial assistance of the National Research Council of Canada, through a postgraduate scholarship, is gratefully acknowledged.

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

With the advent of more powerful and cheaper computer hardware, there has been a correspondingly greater use of general purpose time-sharing systems. Such systems provide each user with a computer terminal from which he can interactively and in real time enter commands and data to the computer system and can interrogate the computer system. It is now common for many manufacturers to supply such a general time-sharing system as part of their major software offerings with a batch stream as an additional more minor component. This is even true for small minicomputer systems. For example, in Digital Equipment Corporation's RSX-11 [DEC 1] series of operating systems for the PDP-11 [DEC 3] computer system, which is a minicomputer system, only the largest version of the system even provides a batch capability at all.

One of the most widely used components of such a time-sharing system is the source editor, which provides support for the entry and editing of source data. The term "source data" is used to mean data in external character form produced directly by human effort, as opposed to

machine generated and manipulated data. Such data usually consists of program text, data for a program, or documentation (this thesis was entered and edited using such an editing system).

This source data is stored in files on secondary storage media such as magnetic disk units. The purpose of the editor is to provide a facility for editing these data files. Appropriate parts of the user's files are brought into the processor's main storage where they may be manipulated; the user employs editing commands to alter his data and the changes he makes are then stored back on the secondary storage media. Some of the best known of these editors include IBM's TSO [IBM 2], Digital Equipment Corporation's TECO [DEC 4], and Stanford's WYLBUR [Faiman 1973]. All general purpose time-sharing systems provide a built-in editor.

Most of these editing systems operate as a standard user program using the operating system facilities and are not actually built-in to the operating system. (i.e. they are treated simply as ordinary application programs). Consequently, they normally allow updating only of the types of files, or of some of the types of files, supported directly

by the operating system. Since the simplest type of file is a sequential series of records, the editor invariably supports the user's data only as a sequential file to which random editing operations cannot be applied directly. Thus, while editing, the editor normally copies the user's file to a specially organized (i.e., non-sequential) work file, edits the copy, and then rewrites the user's file when he is finished. If the computer system fails for some reason (for example, due to a power failure), the user may lose the changes performed during that session.

The editors in use provide a fairly standard set of editing operations. Records may be listed, inserted, deleted, altered, copied, moved and renumbered. Operations on entire files, such as listing, deleting, copying, and moving are also allowed, although these operations are often provided directly by the time-sharing system rather than by the editor itself, since they are of general application, and are not specifically used for editing purposes.

Unfortunately, while a great deal of study and effort has been devoted to operating systems design and implementation, particularly in the areas of process management, scheduling, and storage management, almost no mention is ever even made

of the editing system, although it is that part of the system that is most visible to the user. For example, an examination of the great amount of documentation on the MULTICS system [Organick 1972], which is probably the best known operating system implemented by someone other than a computer manufacturer, yields almost no information about the editing system's capabilities. As another example of the little attention given to editing, Ritchie and Thompson, in their description of the otherwise interesting UNIX system [Ritchie 1974], fail even to mention the editor even though (using their own figures in the same paper) the editor was the most heavily used command and also was one of the largest consumers of system resources.

The goal of this thesis, then, is to study the design, implementation, and performance of a file system providing much more sophisticated file manipulation facilities for the editing system than those currently found in use. The "file system" includes all that support software which provides data management services for the editor, in contradistinction to the editing function itself which includes the editor command language analysis and the actual altering and acquisition of data.

The major thrust of this thesis will be to demonstrate the practicality and cost of a more powerful file system by the construction of an actual file system implementing the new ideas. An actual file system, called the Network Access Method (NAM), has been implemented by the author and is in use providing facilities for the MANTES editing system [Zarnke 1978]. The entire system has not been implemented just to demonstrate the workability of the new ideas but is in production as a major editing system for a current user population of approximately 600 users at the University of Manitoba. The advanced features of the NAM file system have allowed the construction of an editing system possessing many new features over existing editors; the construction of NAM was motivated by the fact that the desired features of the editor could not be implemented using the existing operating system. Note, however, that it is not the intention of this thesis to discuss the actual features or commands supplied by the MANTES editor.

The thesis will begin in chapter 2 by discussing those features supplied for existing editors, and those additional features which are provided by NAM, as well as justification for the new facilities. Chapter 3 will discuss the external interface to NAM: that is, those features supplied to the editor by the file system. This is not to say that NAM

exists solely for use by editors; in fact it should be accessible to programs generally. However, most examples and justifications are based on the existence of an editor as the invoking program. In chapter 4 the storage structures used to implement the features provided will be discussed, while chapter 5 discusses a special aspect of file systems providing a direct updating capability; that of maintaining integrity of the user data across system failures. Chapter 6 discusses the performance and practicality of the NAM file system, using experimentally gathered data. Chapter 7 summarizes the results.

CHAPTER 2 - FILE SYSTEM FUNCTIONS

The first step in the construction of any software system is to acquire a knowledge of what the system is intended to do. This chapter, then, presents a general discussion of the actions of the file system. More specific details will follow in the next chapter but here is presented the background and motivation for the ideas which form its basis. The functions which are discussed here are crucial to the entire file system - the combination of functions discussed here provides the need and justification of the file system since the desired collection of features does not exist elsewhere. The remainder of the thesis will present the difficulties inherent in the implementation of the given functions.

Before starting the discussion proper it is important to distinguish the file system supplying the data storage facilities from the actual editor. The important concerns for this thesis are not the syntax and use of editor commands, but rather the facilities provided by the file system to the editor, used to manage the storage and retrieval of files and records. The function of the editor

is to interact with the user, usually through a series of commands. These commands are translated into requests that the file system perform certain data management functions. The actual editor is important only in that the functions it provides to its users dictate and motivate the functions provided by the file system - it is the desire to provide a sophisticated editor that forces the development of a similarly sophisticated file system to support it. It is not important for these discussions whether the file system is in fact part of, and unique to, the editor, or whether it is a general function supplied by the operating system, although it makes sense to separate it clearly. In general, to make the distinction in function between the editor and file system clear it can be said that the editor accepts commands from its users and performs operations upon sets of records - the actual location of the records on the secondary storage media and the transferring of data to and from the editor's work areas is the responsibility of the file system. The editor should have no knowledge of the techniques used to store the data - it only knows how to invoke the file system to retrieve and insert the particular data items it is concerned with.

2.1 BACKGROUND

A brief history of editing systems is useful to bring together those functions supplied by existing file systems. Source file editing systems have been in regular use since the early 1960's. In collecting a list of features desired in designing a new file system it thus would be expected to find a large variety of ideas and techniques in use by the fairly large number of timesharing systems in use or previously in use. Unfortunately, however, this is not the case. For example, there is a remarkable similarity between the features supported by the Compatible Time-Sharing System [Corbato 1962] in use in the early 1960's and those provided by modern day systems such as IBM's TSO [IBM 1,2], the MULTICS system [Organick 1972], or UNIX [Ritchie 1974]. While a substantial amount of work has been done in other areas of operating systems architecture, almost no original ideas have been presented in the areas of file systems design or source file editing.

Thus it is fairly easy to discern the important features supplied by current editing systems. Since the files operated upon by editors are read by other programs such as compilers, the files used are those generally provided by

the operating system as a whole. In fact, the features provided by editing systems are invariably linked to those of the host operating system; for example, in an operating system not providing sequence-numbered records in files it would be very unlikely to find an editor with sequence numbered records. Thus a discussion of editor features is to a large degree a discussion of the file systems provided by operating systems.

The most common types of files are sequential files: files containing an ordered set of records. Consequently, most editors are designed to operate on such files. The user may select a file and then perform a standard set of operations upon records in the file. These operations include listing, insertion, deletion, altering, moving and copying of records. Sometimes the records of a file are identified by unique sequence numbers; in other editors the records are not explicitly numbered. In either case, the sequential ordering of the records is important. No other structuring of the records is provided; the user must locate a record by its sequence number or by scanning a set of records in order to find one based on content. Only one file may be edited at a time, thus making quite ditficult inter-file operations such as making the same change to several related files.

Since sequential files as used by the editor are not suitable for the addition and deletion of records some effort must be made by the editor and its file system to make changes. Consequently, the file must first be copied to a specially organized internal work file maintained by the editor's file system. This internal work file is organized in such a way that insertions and deletions can be performed upon it. The editing changes are thus made against the work file; when all desired changes have been made the work file is copied back to the permanent copy kept in sequential order that can be accessed by all the other programs in the system. This copying and saving back of the file is slow and costly in machine resources and is especially annoying if the user wishes to make only a few changes (or even just to list parts of the file). This copying to a work file is also the reason that the user may edit only one file at a time - all files to be edited would have to be copied to internal work files first. Thus the user is discouraged (by the time taken) from switching from editing one file to editing another. If, for example, while editing one file he notices he needs to make some minor change to another file he cannot easily do this. In addition, if some problem such as a system failure or power failure occurs he may lose some or all of his changes since

the editor work file may be lost or may be difficult to retrieve.

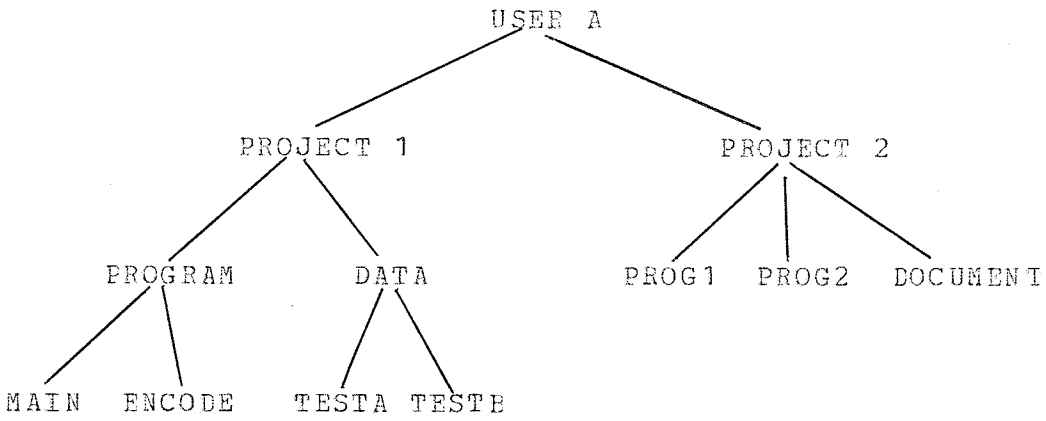
One feature very useful for editing systems is the ability to associate attributes with files. A system which supports, potentially, a diversified set of files, needs to allow files to differ in some respects; that is, the files are not necessarily completely uniform. For example, text files would contain both upper and lower case text while program source files might normally contain text in only one case. Some editors have the ability to associate such attributes with individual files in order that the attributes be applied automatically without extra user effort. In this way, the editor can make any adjustments to its operation (such as converting the user's input to a single case) automatically. These attributes have no intrinsic meaning for the file system, but since the attributes must be permanently associated with the file, the file system must be able to accomodate their addition and deletion.

Another facility that is sometimes offered is that to create several versions of a file. Quite often it is important to restore a file to its state at some earlier time. For example, a user may try some changes to an

algorithm, only to discover that the old one was better, and then he may wish to return to the old form. Having several versions of a file also allows the user to keep a production working version of a program and a current version being altered. This facility can easily (but not efficiently) be obtained by making a copy of a file before altering it. Editing systems which support the facility rely upon operating system facilities for keeping multiple copies of a file. For example, a user of IBM's OS/VS series of operating systems can use generation data groups [IBM 3] to keep track of versions of a file, while users of DEC operating systems such as RSX-11 [DEC 1] are given a direct version facility for all of their files. A more efficient technique for implementing versions will be described later in this thesis.

One of the most important concepts in file management systems, although one which is usually external to editing systems, is the ability to create hierarchical structures to relate files. A user of a computer system normally has sets of files which are related to each other; for example, a program source file, an input data file, and a documentation file, all pertaining to one single project. Virtually all operating systems allow the construction of a file hierarchy to express the natural relations between files. For

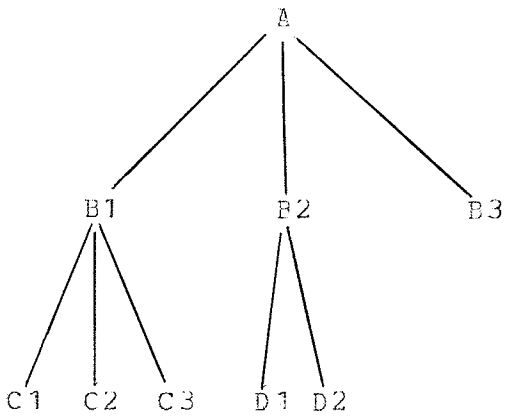
example, a user may create a hierarchical structure such as;



which defines files related to two projects. A hierarchical structure such as this has many advantages; it provides a logical structuring of data; it provides a natural mechanism for implementing naming conventions and protection rights; and it provides a means for referring to a related group of files as a unit. Unfortunately, existing editing systems are not capable of dealing with such a hierarchy and consequently permit the editing of only the bottom files. If a user wishes to alter all occurrences of a text string in several files, he must edit each individual file separately.

Before continuing on with the limitations of existing editors, however, a few definitions and descriptions of hierarchies in general are in order. Given a hierarchical

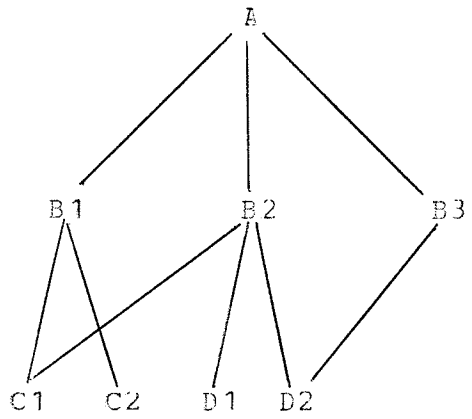
set of files, such as that depicted here, each participant in the tree structure is called a node.



Those nodes appearing at the bottom level and possessing no subdivisions are called leaves, (C1, C2, C3, D1, D2, B3) while the very top node is called the root (A). Starting with any node in the hierarchy and travelling upwards, all nodes passed on the way are called the ancestors of the given node. (A and B1 are the ancestors of C1), while the immediate ancestor of a node is called its father. (B1 is the father of C1). In a similar fashion, the nodes immediately under a given node are called its sons. (C1 is a son of B1). Different sons of the same father will be called brothers. (C1 and C3 are the brothers of C2).

In addition, hierarchies in which any node (except the

root node) may have several fathers will be discussed, as in;



Such structures will be referred to as networks; the nodes C1 and D2 in this example will be referred to as network nodes and they possess multiple fathers.

While the user is generally given the ability to create such hierarchies (although some systems do not allow the creation of networks), only the leaves themselves contain any data. The ancestors of the leaves exist only for the purpose of relating the leaves and assist in naming them. Each node in the hierarchy is named (by the user); the name by which a node may be referred to is that generated by concatenating the names of all its ancestors, beginning with the root node, down to the given node. Some delimiter, such

as a ".", is generally chosen to separate the individual names in the overall name. Thus in the last example, node C1 could be named

A.B1.C1 or A.B2.C1

depending upon which path is used to reach the node. The names A and B1 are said to be qualifiers of C1, while the complete name (A.B1.C1) is called a qualified name, or totally qualified name; the name of each individual node is called its unqualified name; and a qualified name in which some of the ancestor's names are omitted is called a partially qualified name. For example, the unqualified name of node D2 is simply D2; the totally qualified names are A.B2.D2 and A.B3.D2, and it has partially qualified names A.D2, B2.D2, or B3.D2.

Almost exclusively, existing operating systems provide two choices for referring to nodes of the hierarchy. A totally qualified name may be provided, or the user is allowed to have a "current directory" which identifies a node in the hierarchy; he may refer to any sons using their unqualified names. For example, if a user has a current directory of A.B1 he may refer to C1 and C2 directly using their unqualified names, or he may specify their full names A.B1.C1 or A.B1.C2. Generally two nodes in the hierarchy

may have the same (unqualified) name as long as the names referring to them are distinct. For example, using the last hierarchy, node C2 could have been called D1 since A.B1.D1 is distinct from A.B2.D1. However, node C2 could not have been called C1 for then A.B1 would have two sons called C1.

2.2 LIMITATIONS OF EXISTING EDITORS

This section discusses limitations of the facilities just mentioned in existing editing systems. Since this thesis is concerned with file management for editing systems, no discussion of editor command language or syntax limitations will be given; only those facilities connected with the storage of data are relevant.

One of the major limitations in existing editing systems is the fact that they work on a temporary copy of a file. One disadvantage of this technique is that the copying and saving back of the file is slow and costly in machine resources, and is especially annoying if the user wishes to make only a few changes. As mentioned previously, this cost also discourages the user from switching from one file to another; he cannot simply save his current position, make a few changes to another file, and then come back to where he

left off. In addition, many of the copying editors provide no recovery procedures after a system failure. After a failure, the temporary copy of the file is often lost and hence the user must reenter all his changes up to the time of failure. For all these reasons the file system described in this thesis is a direct updating editor; no copies of files are made. Changes are made directly to the actual file.

While most operating systems do provide facilities to manage hierarchies of files, it is unfortunate to note that existing editors do not make use of such a feature. Editing systems only permit the editing of one leaf file of the hierarchy at a time; the creation and manipulation of the hierarchy is entirely external to the editor. Only the leaf nodes contain data; to the editor these leaf nodes are completely independent. This limits the user in several important ways; he cannot perform global editing across a set of related files, he cannot structure his data to match its logical structuring (for example, to subdivide program text into functional blocks), and common items such as file attributes cannot be attached to groups of files, rather than to individual files.

Having an editor which can only operate upon the leaf files of a hierarchy is somewhat restrictive to the user. The user is then unable to group his data in such a way that the physical structuring matches the way in which the user logically views it. The user views text, for example, as being composed of chapters, sections, and paragraphs, etc., rather than simply as a set of records in sequence. Similarly, a program is composed of procedures, sections, etc. With the advent of such programming techniques as structured programming, it only makes sense to physically represent data according to its logical structure. Since structured programs are generally thought of as hierarchical, it seems natural to use a hierarchy to represent such data.

In order to provide such a substructuring facility to the user, it is not sufficient merely to provide the ability to manage hierarchies of files (as other systems do); the editor itself must handle such constructs. The user must be able to treat subportions of the hierarchy as a unit so that he can perform editing operations upon them. He also wants to be able to treat each individual subsection as a unit when it is more convenient. (for example, to reorder the subsections). This ability to utilize the hierarchy is not present in current editing systems.

A second limitation of the failure to make use of the hierarchy has to do with the efficiency of editing. When the user is initially developing a program, he does a great deal of editing on it. Consequently, he wishes to make this as easy as he can for himself. If he were to break his program up into a number of separate files (assuming that the program is large enough to warrant this), then he incurs a serious penalty. This is because of the relatively high cost of switching from the editing of one file to editing another, especially in a copying editor, which must make a copy of the file before editing can commence. To make the same changes to separate sections of data would entail the user reentering his editing command sequence for each file to which he wished to make these changes. Thus, for example, to change the name of a variable used in a program, he either would have to store the entire program in one file to edit as a unit, or else repetitively edit each subsection, going through the procedure of copying each subsection to a temporary file, reentering the editing command sequence, and copying back the temporary file for each subsection. Needless to say, this forces the user into using large files for greatest convenience.

The third limitation of editors with respect to file

hierarchies is the editors' inability to associate editor-specific attributes with groups of related files rather than with each individual data unit. These attributes are used to assign special characteristics to files, such as the tab positions used for inputting data to the file, or the maximum length that is allowed for records in the file. While many editors supply such attributes through one means or another, they cannot be applied to groups of files, but must be respecified for each individual file. The user may wish to associate an attribute (such as that allowing the use of lower and upper case data), with the entire group of files containing the text, rather than respecifying the attribute for each component file. The file hierarchy could provide a convenient scaffold for attaching and applying attributes; an attribute could be attached to a node in the hierarchy, and then be automatically applied to all sub-nodes. Since existing editors do not provide hierarchy support, this facility is lost to the user; editor-specific attributes can only be handled at the individual leaf file level. (and even there it is not done by attributes but rather by some other implicit means).

2.3 PROPOSED FILE SYSTEM

As a solution to the limitations just mentioned, and to provide new functions not previously discussed, this section describes in a fairly brief manner a file system to allow the construction of a much more powerful editing system. The two major distinctions this file system has are the ability to manage hierarchies of files, and the ability to perform direct updating of data without first copying it.

2.3.1 DIRECT UPDATING

The major advantage of direct updating is the time saved by not having to copy each file to and from a temporary work file each time it is edited. While not only saving the user time, direct updating also allows the user to more freely switch from file to file since the time involved in such a switch is much less. However, direct updating has serious implications to the internal design of the file system itself. The data can no longer be stored as a physically sequential file since no provision exists in such a file for adding or deleting records in the middle. Thus a more complicated storage structure using linked data items, or

using indexes to locate data records, is required. Two other problems arise; achieving acceptable performance in terms of the amount of data transferred to and from main memory, and providing integrity for the data.

Discussing the matter of performance first, a problem arises in keeping the file on secondary storage always up to date. Since data is actually stored in blocks, it is necessary to transmit an entire block each time a piece of data is to be updated. In particular, for every new record added, or every record changed, such a data transfer would be needed. This is a considerable expense; some means must be found for reducing the number of transfers. In a copying editor, the temporary work file is kept as much as possible in main storage; transfers to secondary storage are required for the initial copying of the file, the copying back after all changes are made, and whenever the work file exceeds the amount of main memory available. Since one of the major reasons for using a direct updating file system is to have an up to date file in case of a system failure, such as a power failure, keeping the changes only in main memory, as is done in a copying editor, or even just keeping the changes in a temporary copy of the file, is not acceptable. However, groups of changes can be collected together and then be written out as a unit in order to keep the file

reasonably up to date, so that in case of a system failure the disk copy of the file will be close to the main storage copy - only the last few changes may not have been applied. In addition, the file system may provide a means to force the file to be updated at any given time. Thus the editor may force the file to be updated, for example, after every tenth new record is inserted, thereby keeping the disk and main storage copies of the file close to one another. In this way, the disk copy is not altered for every change to the user's data, at the discretion of the editor. The number of transfers is reduced by not keeping the disk copy completely current. However, the transfers for the initial copy to a temporary file, and the final copy back, required for copying editors, are eliminated. A major emphasis of this thesis is a demonstration that a direct updating file system is practical in a performance sense. An entire chapter will be devoted to discussing performance.

As well as a possible performance problem, direct updating makes preserving the integrity of the user's files difficult. The term integrity is used to mean the consistency and correctness of the data. A loss of integrity occurs when some of the user's data becomes lost, or damaged, or is altered in any way not intended by the user. Such a loss of integrity usually results from a failure in

the system during an update to a user's file.

The reason there exists an integrity problem in a direct updating file system is that data is not just related via physical positioning; pointers are also needed. In a sequential file, the only action that can be performed that modifies the file is to add records to the end; insertions elsewhere and deletions cannot be performed. Thus in the event of a system failure, the only possible problem can be that some records did not get added to the end; the records remaining are left in a consistent manner. However when records are linked via pointers some problems arise. Each addition or deletion may involve several changes to data. For example, adding a record into the middle of a file may require altering the link from the previous record as well as that of the new record. Whenever two or more items must be altered to effect an operation a possible integrity problem exists, for if one change is made and the system fails before the second change is made, inconsistent data may be left in the file. Pointers may exist to invalid locations, or some data items may be pointed to from several places instead of from just one place. In a copying editor, the actual file is changed only at the end of an editing session when it is rewritten in its entirety. The only system failure that could affect the user's data is one

occurring during the copy back to the file. In a direct updating file system the file is constantly being modified; a system failure is much more likely to occur at a time when the file is being modified. Thus another major emphasis of this thesis will be to demonstrate the prevention of integrity problems. A later chapter will be devoted to the topic.

2.3.2 HIERARCHY SUPPORT

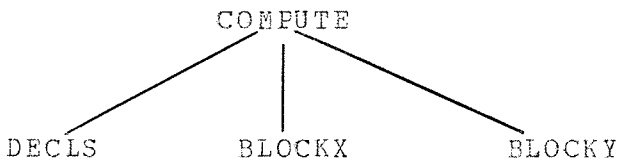
As mentioned previously, one of the two major distinctions the new file system possesses is support for managing entire hierarchies of files. While existing operating systems do support hierarchies of files, current editing systems do not make use of them. The file system described in this thesis permits the use of hierarchies in a way most useful to the editor, as well as providing some new facilities not found in existing operating systems. The editor will permit operations such as: operating upon an entire subportion of the hierarchy as an individual unit, referring to files using only partially qualified or unqualified names, associating attributes with entire parts of the hierarchy, and protecting entire hierarchies and subparts of the hierarchy.

It is useful to be able to treat an entire subportion of a hierarchy as a single unit for editing purposes. This enables editing across sets of related files, which in turn allows the user to subdivide his data in a way that matches the logical structure. The editor should be able to locate any node in the hierarchy and then process it as a simple (unstructured sequential) file with the file system performing any "flattening" required to make the node appear to the editor as a simple file. This is not to say, however, that the hierarchy should always appear in a "flattened" manner. The editor should be able to manipulate the hierarchy as such whenever the user wishes.

Going hand in hand with the automatic flattening of files is the ability to have each node possess both its own data records as well as having sons in the hierarchy. To allow the user to structure his data in the same way as he logically structures it, having a node contain both data records and son nodes can be useful. For example, a procedure in a programming language usually consists of a header statement of some form, followed by the text, followed by a trailer statement, as in;

```
COMPUTE; PROCEDURE(A, B);
     -declarations
     -internal section X
     -internal section Y
END COMPUTE;
```

Since the text is usually composed of several sections, as depicted above, it is useful to represent the entire procedure by the hierarchy;



where, of course, each of DECLS, BLOCKX, and BLOCKY may be further subdivided. However, the header and trailer statements logically belong at the level of node COMPUTE in the hierarchy. Thus the ability to have a node possess both data records and sons is a useful one, and fits in with the automatic flattening mentioned above. When the user wants to operate upon the procedure as a unit, for example to compile it, the file system has the ability to flatten it. When the user only wishes to operate upon the data records of a specific node (for example to change the header and trailer statements of procedure COMPUTE above), the file system can be asked to work only with the records directly possessed by a node.

While many operating systems support file hierarchies, they allow reference to individual nodes in the hierarchy via only totally qualified names, or using the concept of a current directory, in which names must still be qualified, but the root of the hierarchy is a different point. This is not a major drawback in systems in which the editor can only operate upon one leaf file. However, when a much higher degree of substructuring is provided and the user is expected to switch from one file to another much more frequently, using totally qualified names can become quite a nuisance, owing to the length of names involved. Thus the file system described here permits both partially qualified and unqualified names. In order to handle possible ambiguity from having several files with the same name, some means for resolving ambiguity is needed; this will be discussed in a later chapter.

One facility specially required by editing systems is the ability to associate attributes with files. In a file system supporting a hierarchy of files, attributes may be associated with entire groups of files by having an attribute pertain to all subfiles which do not explicitly respecify the value of the attribute. In this way, for example, an attribute specifying the use of both upper and

lower case text can be applied to a document, even though the document is substructured. This inheriting of the attributes is performed automatically by the file system, including telling the editor of changes in attributes due to passing from one file to another, while flattening out an entire hierarchy.

In a similar manner, protection normally supplied to files by the operating system may be extended to allow more editor-specific protection features. Protection of files is normally a feature which is related to the file hierarchy [Saltzer 1974]; a password may be required to descend from one file to a subfile; that is, access to a file does not necessarily grant access to its subfiles. As will be detailed later, this is not necessarily desirable in an editing system; when structuring is used to subdivide text rather than to relate files, it is more reasonable to have a password granting access to a file to also grant access to its subfiles. In addition, since existing editing systems access only the leaf nodes, protection is left to the operating system to implement. The operating system decides before the editing session begins whether a user can access the file being edited, either through explicit passwords or through the user's name. In an editing system supporting the entire hierarchy, then, a different level of password

support may be provided. Access rights other than the usual read access, or write access, but instead specific to sets of editing functions, may be established. For example, a specific password may allow a user to add records, but not delete existing ones. Passwords may be inherited in a similar manner to attributes; a right conferred by a password may be given for all lower level files; any lower level file may possess another password granting a wider range of rights. In this way, protection may be integrated into the editor; the user may specify all passwords he knows (either explicitly, or implicitly through his user identification); the file system then grants or denies each request depending upon the particular file and type of request.

2.4 SUMMARY

In summary, this thesis proposes a new file system designed specifically for use in source file editing. The major features of this file system are the ability to perform direct updating of files, and the ability to handle a hierarchical file structure with entire sections of the hierarchy able to be handled as units, with the use of partially qualified and unqualified names, and with inherited attributes and passwords down to lower levels of

the hierarchy. This file system will also provide some other features which have not yet been mentioned, such as versions of files without creating entire new copies of files, and the networks of files. All of these features will be discussed in greater detail later in this thesis.

This combination of features does not exist in any current file system, including those not intended for use by editing systems. It is not obvious that the construction of such a file system is feasible; the questions of performance and integrity have already been raised. The emphasis of this thesis, then, will be to demonstrate, via an actual implementation of such a file system, the feasibility of the design. The remainder of this thesis will discuss an actual file system, the Network Access Method (NAM) [Ferch 1978], implemented by the author and in production use at the University of Manitoba. This file system is used to provide file support for an advanced editor, called MANTES [Zarnke 1978]. Rather than the MANTES editor, although it provides many features advancing the state of the art in editing, but rather the level of support provided by its file system, NAM is of interest here. Discussion of NAM will be done in four major parts. Chapter 3 gives a discussion of the interface to NAM, which delimits exactly what features are supported. Chapter 4 discusses the storage structures used to permit

direct updating, and Chapters 5 and 6 discuss the questions of integrity of the data, and the performance of the file system.

CHAPTER 3 - EXTERNAL INTERFACE

This chapter describes in greater detail the features provided by the file system to the editor. The interface between the file system and the editor determines to a large extent what operations the editor may perform and the complexity of effort required to perform file management. The layout of the chapter is a discussion first of those functions provided by existing file systems, then of the logical subdivision of file management operations to match the hierarchies of files supported, and finally of the actual operations provided for each subdivision.

3.1 INTRODUCTION

Before designing the interface for a new file system, it is useful to examine those file systems already in use. All operating systems supply one or more basic file systems, or "access methods" to supply file management services. However, these file systems usually are designed to operate only upon a single file at a time. The file hierarchy, if such is supported, exists external to the access methods and is managed via a separate set of operating system routines from the access methods. Thus it is not possible to request an access method, or file system, to read a consecutive set of records from a group of related files. Thus existing file systems are not adequate for supporting the features mentioned in the previous chapter; however, some facets of their design are still useful.

All operating systems supply a basic sequential file system to allow users to read from and write to sequentially ordered files. Generally, no random access to records is allowed. (Many storage media such as punched cards, paper tape, magnetic tape, and so on allow only sequential storage and retrieval). The records in such files are identified only by their position in the file and not by any key

values. The interface with the user involves four functions or operations; the open function identifies the file and sets processing options, the read or write operation allows for reading from or writing to the file, and the close operation disconnects from the file. The file system handles details such as buffering, blocking, and device dependent details so that the user program need not concern itself with anything except the data records.

One important concept present even in such basic file systems is that of a current position, or "record position". At any given time, the user is positioned at a specific record in the file; the next operation is performed relative to this record position. Some sequential file systems allow the user to modify this record position anywhere within the file. The desired position may be selected using either a relative record number within the file, or using the device address of the desired record. However the only use for this random positioning is to establish the record position for subsequent sequential operation; records may not be inserted or deleted from the middle of the file.

A more sophisticated type of file system is very often supplied. Such a file system operates upon records, each of

which is identified by a unique, or primary, key [Knuth 1973]. Of these file systems most provide for both sequential operations and direct, indexed operations, through the record key, and hence are commonly called indexed sequential access methods. Using IBM's VSAM [IBM 4] as an example, such file systems provide the ability to read, rewrite, insert, and delete a record by specifying the value of the record's primary key. Such file systems also support the concept of a record position; a record key may be used to establish a record position. Subsequent sequential operations may then be performed relative to this record position. Several such record positions may often be in effect; the user may then choose the active record position relative to which he wishes each operation to be performed. The ability to have several such record positions active is important to an editor for such operations as copying or moving a set of records to a new position.

These two types of file systems generally are the only ones provided that support sequential sets of records. Even very sophisticated data base systems supporting very complex hierarchical or relational sets of data are built using these two types of file system [Martin 1975]; any special file management functions, such as handling secondary keys for records, are provided internal to the data base systems

and not by the operating system and hence are not usable by the editor.

However, these file systems are inadequate for performing the functions mentioned in the previous chapter. They do not provide any support for the file hierarchy, thus making operations upon groups of files very complicated. A second major shortcoming is their inability to work directly with sets of records. While the user may establish a starting file position, he cannot also establish an ending file position, to delimit a range of records. This is an especially important consideration when the desired range of records spans several files in the hierarchy; the ending record in the range may not be identifiable simply by its primary key since there may be several records from different files with the same key value within the given range. These ranges of records are important in editing systems - very often a user wishes to apply an editing operation over a given subset of his records. Other shortcomings, related to the lack of support of the file hierarchy, include the lack of support for inheriting of attributes, partially qualified and unqualified names, and other desirable features mentioned in the preceding chapter.

Keeping the desired features from the previous chapter in mind then, the interface provided by the NAM file system is now discussed.

3.2 COMMON ASPECTS

The most important difference between NAM and other access methods is the ability possessed by NAM to manage a hierarchy of files. The user may create and manipulate entire sets of files. However, this hierarchy does not pervade the entire set of files managed by the operating system. In order to delimit the space occupied by one user's files from that occupied by another user, the file hierarchy for each user is collected together to form one entity, which is called an aggregate. An aggregate, then, is simply the physical realization of a hierarchy of files with a single root file and a set of subordinate files, or subfiles. Each aggregate is totally independent from any other aggregate, and thus supplies a means for providing management of storage space and accounting of space by clearly delimiting the files for one user from another. This is not to deny any user of the system from possessing more than one aggregate, but a single aggregate is the unit that NAM may operate upon at one time. Just as a user of a

regular file system may operate upon one single file at a time, so a user of NAM may operate upon one single aggregate, although that aggregate may contain an entire hierarchy or network of files.

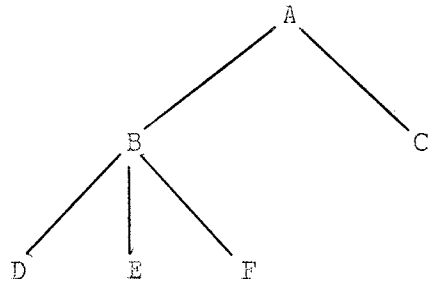
NAM thus provides a set of operations to manage the data contained in a single aggregate. Before going on to discuss the individual operations, however, some general comments will be given. These include comments on the means by which NAM supports record positions to identify the current position in the hierarchy; the subdivision of operations into three levels; aggregate, file, and record; the handling of ranges of records; and support for passwords. Before continuing, however, a few clarifications are in order. In the context of discussions of NAM, the term "file" will be used to mean any node in the hierarchy, and the subtree of that node. A "subfile" will refer to any subordinate node, at any level, under a particular node (file). A "subfile" will also denote a file in the hierarchy, but subfile will emphasize that this file appears as a subnode of another file.

In a standard file system, once a file has been opened, the user is given some means of identifying the file which

he has opened to other calls to the file system. This file identification usually consists of a unique number, identifying the file, or the address of some control block in memory which identifies the file. This file identification is passed by the user as a parameter to other operations provided by the file system, such as the read or write operations, in order to identify which currently open file is to be processed. In file systems supporting the concept of having several current record positions, the user must also indicate to each operation which current record position the operation is to be performed relative to, either through a second parameter or via a special file system operation which is used to indicate which record position is to be used for future operations. In either case, for each data management operation, the user supplies a pair of identifications; one defining which file he is using, and the other subdividing that file to indicate which position within the file to use.

In NAM, this idea is extended through another level. Corresponding to the file identification is an aggregate identification, which is used as a parameter to all NAM operations to indicate the aggregate being processed. Since NAM can operate upon entire file hierarchies, however, the current record position concept must be extended. In a

standard file system, the record position identifies which record within the file the user is positioned to. In NAM, the record position must be extended to indicate two pieces of information; which file in the hierarchy is positioned to, and then which record within the file is positioned to. Since it is desired to have NAM automatically "flatten" files for its users, the record position concept has been extended into two separate positions; a file position, and a record position. Naively, the file position indicates a particular file within the hierarchy, and the record position identifies a particular record within that file. However, there are additional complications as evidenced by the following example.



If the user is operating upon a leaf file, such as D, E, F, or C, there are no complications; the file position identifies the particular file, say E, and the record position identifies a particular record within file E. However, suppose that the user wishes to work with file B as a unit,

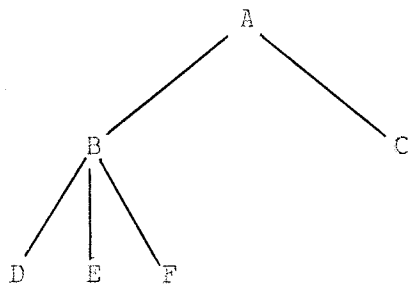
and wishes NAM to automatically flatten it. Then the file position identifies file B, while the record position must identify not only a particular record, but also which son of B contains that record. Thus the current file position may indicate file B, while the record position may identify record 20 within file E.

This concept of subdividing the file position into two parts, one identifying a file, and another a particular record under that file, including the subfile containing the record, is a most important one, for this is the means by which entire subsections of the file hierarchy may be treated as single files in a way transparent to the user. The user may establish a file position to identify any file in the hierarchy, regardless of the level of that file in the hierarchy, and may then use a record position to identify the current record under that file. On subsequent operations, the record position may move from one record to the next, including jumps from one subfile of the file position to another in a manner transparent to the user. The record position may at one time identify the last record of file E above; if the user then asks to read the next record, it will then automatically identify the first record of file F.

Of course, the record position must always identify a record which is contained in a subfile of the file position. A hierarchy thus exists; the aggregate identification identifies which aggregate of files is to be used; as a subdivision of this several file positions may exist, identifying several files within the aggregate hierarchy; each of these may be further subdivided with several record positions. Corresponding to these three subdivisions, NAM operations are divided into three levels; the aggregate operations perform operations pertaining to an entire aggregate and are provided with an aggregate identification as a parameter. Operations to manage the file hierarchy itself are grouped into a set called the file operations, and are provided with both an aggregate identification and a file position as parameters. Operations to manage records are called record operations and are provided with an aggregate identification, a file position, and a record position as parameters. A user who wishes only to work with data records need only open the aggregate, establish a file position using one of the file operations and then use record operations from then on, while another user wishing to examine the file hierarchy itself might use several of the file operations but none of the record operations. Together the three sets of operations provide the editor

with a very flexible file system to enable it to handle hierarchies of data.

Earlier in the chapter it was mentioned that it was desirable to be able to delimit both ends of a range of records, to simplify editing of sets of records. An example will help to explain.



The file that the user is currently editing might be file A, using the above portion of a file hierarchy. Working on file A as a unit, the user may wish to change all records beginning with record 30 in file E, and ending with record 3 in file C. In order to simplify such an operation for the editor, NAM provides a special facility, not present in standard file systems. While editing file A, the editor will have established a file position identifying file A. To delimit such a range of records, the editor may establish two record positions under the file position; one identifying record 30 in file E, the other record 3 in file C. The

editor may then, using either one of the record positions proceed from record to record either starting with record 30 in E and proceeding forward using the first record position, or starting with record 3 in file C and proceeding backwards using the second record position. As the editor moves from record to record, reading each record, NAM has a special ability to compare the current record position with the other record position and to indicate when the two identify the same record. Thus to work with a range of records, the editor simply establishes each end of the range and proceeds from one end to the other. NAM will then indicate when the end of the range has been reached automatically. This facility exists on any operation which can move from one record to another.

Turning now to the last of the general comments, it is appropriate to discuss password protection in NAM before proceeding to the specific operations. Since the hierarchy allowed is intended to a large extent to permit the substructuring of files into manageable parts more so than as a means of building groups of related files, it seems that passwords should control access in a different way from standard usage. In file hierarchies supplied by operating systems, a password is usually required to descend from one file to a subfile; that is, access to a file does not

necessarily grant access to subfiles. In NAM, the reverse is true; if one is granted access to a particular file, then if the hierarchy is used to permit substructuring of that file, one should evidently be given access to any subfile. Furthermore, in NAM, access to a subfile does not mean that access to an ancestor was even necessary; this allows a small part of a hierarchy to be made accessible via certain passwords without the user having access to higher levels. For example, a user may be given access to a set of files containing a subprogram without his being allowed to access the higher level files containing the main program. This of course does not prevent the addition of more passwords on lower level files that provide greater access to those who know them, thus providing similar password access to standard file hierarchies.

In the access levels granted by the passwords, as much flexibility as possible is desired. While most systems provide at least two levels of access, read and write, the need for several others can be visualized. Examining the basic operations provided by the file system, eight levels of access can be distinguished. These are (in order from least restrictive to most restrictive):

1) altering passwords
2) modifying the file structure
3) creating new subfiles
4) adding file attributes
5) altering records
6) adding records
7) listing records
8) displaying the file structure

Each of these levels permits all functions that more restrictive levels allow. Thus a user who has permission to alter records also automatically has permission to add, or list records, and to display the file structure. When an operation is performed, passwords on all ancestors of a file are checked; the level of access permitted is the least restrictive associated with any of these passwords. This check is performed automatically by NAM during each operation; the user supplies all passwords he knows to a special NAM aggregate operation provided for this purpose; NAM retains all these passwords and checks all of them for each operation.

3.3 AGGREGATE OPERATIONS

Turning now to the individual operations themselves, the highest level of operations are those dealing with an aggregate as a whole. Corresponding to the open and close functions of standard file systems are equivalent open and

close operations for aggregates. Given the operating system name of an aggregate, the open operation locates it and prepares for processing. To identify the aggregate for later processing, the open routine returns the previously mentioned aggregate identification. This aggregate identification is passed as a parameter to all other operations as a means to identify which aggregate is to be operated upon. Since it is always required, no further mention of the aggregate identification will be made.

Providing the opposite operation to open, the close function disconnects from the aggregate and frees up any main storage used.

In order that the editor may be guaranteed at any time that the copy of the aggregate on secondary storage is up to date, a special aggregate operation, called the purge operation, is provided. This operation merely writes all changes that have been performed in the main memory copy of the aggregate back into the aggregate itself, and is used by the editor to keep the aggregate reasonably up to date, so as to protect the user against losing too many changes in the event of a system failure.

As has been mentioned, an operation is needed to accept and keep track of the passwords supplied by the user. This operation, since it is not specific to any file or record, is classed with the aggregate operations. The supplied passwords are retained as long as the aggregate is open and are automatically used on every operation to check if access is to be granted.

Another aggregate operation exists to provide the caller with statistics on his aggregate usage. These statistics indicate how much space has been used within the aggregate, how many records and files exist, and how many data transfers have occurred. These statistics may be used to verify the correct functioning of the file system, and to determine the cost of using the file system. They will be referred to again in a later chapter.

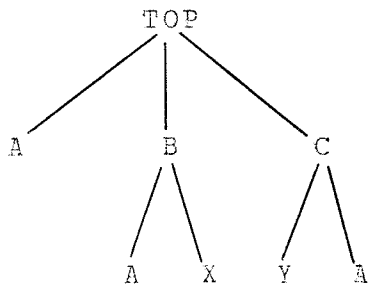
3.4 FILE OPERATIONS

The second level of operations in NAM are those which manage the file hierarchy. These operations include locating a file, creating a file, linking a file under a second father, copying a file, moving a file, and adding or

removing attributes and passwords from a file. All of these operations accept as a parameter a file position; some of them also create new file positions. In a similar manner to a record position in a standard file system moving from identifying one record to the next record as a result of a read operation, so a file position in NAM may identify a new file as a result of an operation. For example, to perform a tree traversal, used to pass through all of the nodes of the hierarchy, the editor may wish to pass from one file to its brother; the file position indicates which file is currently identified. In a similar manner to having record positions delimit the end or ranges of records, file positions may also be used to delimit ranges of files. Operations which are capable of moving a file position from identifying one file to identifying another are also capable of comparing two file positions, so that the editor may easily handle sets of files, if it so wishes.

Corresponding to the open and close operations for aggregates there exists two operations for establishing and removing file positions, called the locate and unlocate operations. The locate operation can establish a file position to a file identified by a name, or identified by position; the unlocate operation simply removes file positions that are no longer active, thus saving memory space.

The most common way in which a user can establish a file position is to supply a name to the locate operation. This name may be an unqualified name, or it may be totally or partially qualified. In order to allow for having several files with the same name, NAM will accept any name which can be resolved unambiguously. However, simply stating that any unambiguous reference is acceptable is an oversimplification. For example, using the hierarchy



it is evident that the leftmost file called "A" cannot be named unambiguously, since TOP.A is an acceptable partially qualified name for nodes TOP.B.A and TOP.C.A. The solution in this case is the same as that used in the language PL/I [Hughes 1973]. If a name is specified which is the completely qualified name of some file, then it is taken to refer to that file. Thus, in the above example, TOP.A refers to the leftmost file "A", while TOP.B.A, B.A, TOP.C.A, and C.A are all unambiguous references to the other

two files named "A". It was mentioned previously, but is worthwhile mentioning again, that two brothers may not have the same name (i.e. neither file "X" nor file "Y" above could have been called "A"), since then no unambiguous references by name would be possible.

One other comment applies to referencing files by name. As mentioned, many systems supporting file hierarchies provide the ability to have a "current directory", with names referring only to subfiles of that current position. Thus, in the above example, if the user had a current directory at file "B", then the name "A" would refer unambiguously to file TOP.B.A. In NAM, to supply an equivalent facility, the user may supply a file position, along with a name when a file is accessed. In that case, the file position restricts the search to all subfiles of the given file position. Unlike the support in other systems, however, the search by name includes the node identified by the file position. In other systems the hierarchy relates files; in NAM a hierarchy is used to substructure files. Because of this, it is expected that the file identified by the file position will be the highest level of a substructured item; as in the example of procedure COMPUTE, in the preceding chapter, it may contain some of the data records of the data which the entire

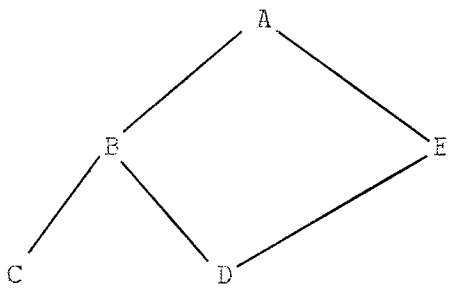
substructure contains. Thus it is desirable to be able to group the highest level node in with the rest; a file position identifying file "B" above allows the user to access files B, A, and X directly by unqualified name.

For the purposes of performing tree traversal, it is necessary to be able to refer to nodes of the hierarchy by position. Thus NAM allows the user to establish a file position at the root node of his aggregate, or to a father, brother, or son of a given node. To locate a father, brother, or son, the user provides a file position identifying the given node to the locate operation; NAM will then alter the file position to identify the new node, or create a new second file position identifying it, depending upon the user's wish. At the same time, the function of comparing the new position to a given position, that was earlier mentioned, is carried out, in order to facilitate operations upon a range of files.

Other file operations include the create operation, to create a new file, which has as its arguments the name of the new file, and a position relative to which the new file is to be created. This position to which the creation is relative, is normally a file position; however, since any

node may possess both data records and sons, it may also be a record position, indicating which record of the file the new son is to be created relative to. The new node may then be created as a left or right brother of the given file position, as the first or last son of the given position, or as a son right or left of the given record position. For obvious reasons, the create operation must check if any files of the same name would exist as brothers of the new node. As an option, the user may have a new file position created to identify the new node so he may immediately process with it; for example, to start inserting records.

A similar operation to the create operation is the link operation, used to include a node under a second father, thus creating a network structure. The link operation accepts the same parameters as the create operation except that in place of a name for the new node, the user supplies a file position identifying the existing file which is to be linked under a second father. As well as checking for two brothers with the same name, the link operation must check for the creation of an "infinite loop", in which a file is linked as an ancestor of itself. For example, in



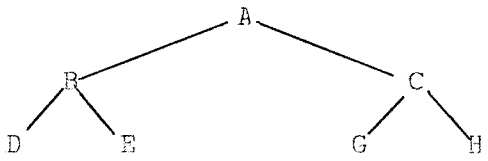
it would be illegal to link either node B or node E under D since then node B or E (as well as node D) would then become an ancestor of itself. Such an occurrence is undoubtedly a user error and would surely cause the file system or editor to loop indefinitely while performing some operations. Such an infinite loop can always be prevented by not allowing any node to become its own ancestor, at any level in the hierarchy.

The opposite of the create and link operations is of course file deletion. Since protection against inadvertent deletion is desirable, NAM allows only the deletion of one leaf node at a time; an entire hierarchy cannot be deleted in one operation. Thus, NAM forces the editor to be aware of user attempts to delete entire hierarchies; techniques for protection against inadvertent deletion, such as maintaining an audit trail of deletions are left up to the editor to implement. The delete operation is passed a file

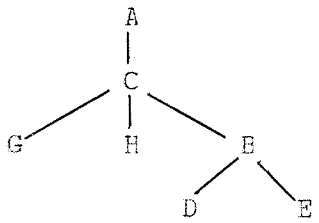
position to the leaf node to be deleted. There are several unusual aspects of deletion that must be recognized, however. For a node in a network with several fathers, only the link from the father specified by the file position is removed; the file itself remains as long as it appears under at least one father. Another unusual aspect occurs because multiple current file positions are permitted. It is possible that another current file position may exist which points to the file being deleted; NAM either must disallow the delete or alter the other file positions to indicate that they are invalid. The particular solution adopted in NAM was chosen due to the design of the MANTIS editor, which frequently keeps two file positions pointing to the same file. Thus NAM normally cannot reject the request for deletion and consequently simply marks other file positions identifying the file as invalid; when the editor tries any operation on one of these, except to unlocate it, it is treated as an error.

A very simple but required operation is that of renaming a file. The rename operation requires, of course, the new name, along with the file position of the node to be renamed. As for creation and linking of nodes, the new name must be checked to make sure that two brothers with the same name do not exist.

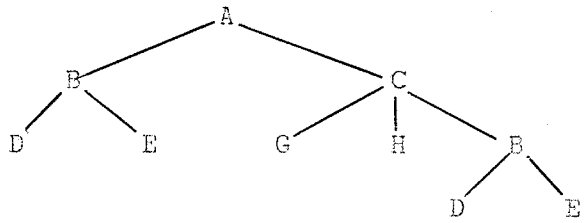
Two other operations are somewhat more complicated. These are the operations of moving and copying files from one location to another. Both operations require two file positions; one describing the node to be moved or copied, the other defining the new position to which the file is to be moved or copied. The move operation moves the entire substructure of the hierarchy whose ancestor is the given node; copy creates a new copy of the substructure (retaining the same names). For example, in the hierarchy



if node B is moved to a position as the last son of C, the resultant hierarchy is



whereas a copy of node B to a position as the last son of C results in the hierarchy



It should be noted that, unlike the copy operation, the move operation does not cause any duplication of data. Only the pointers relating the nodes are altered.

Support of attributes is performed by two special attribute operations; one to add or delete attributes from a given node in the hierarchy, the other to extract the attributes. Each attribute field is identified by a unique number. Addition or deletion of an attribute from a node is straightforward; the operation requires a file position and the attribute field. Reading of an attribute, however, is somewhat more complicated, since attributes are inherited down from a file to all of its subfiles. A particular attribute applies to the file to which it has been added, and to all subfiles in which it has not been given a new value. Thus, if an attribute does not exist specifically on a given node, then reading the value of the attribute requires the examination of the ancestors of the file in turn, until one possessing a value for the attribute is

found. Each attribute field is inherited on its own; one attribute may be inherited from a node's father, another may be inherited from the root file of the entire hierarchy. Thus general attributes may be given by adding them at a high level in the hierarchy; they may then be overridden for certain substructures by changing their value on a lower level node.

In a similar manner to attributes, passwords may also be added to nodes in the hierarchy. However, passwords are identified by their names, rather than by number. In addition, each password has one of the access rights described earlier associated with it. Checking the access rights on each operation is automatically done by NAM; however, a file operation is needed to add and delete passwords from a node; this lock operation is passed a file position and the password to be added or deleted.

3.5 RECORD OPERATIONS

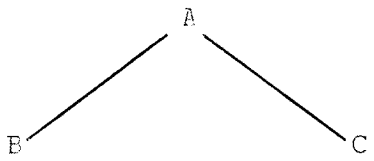
The bottom level of NAM operations are the record operations, which perform the actual management of data records. These operations include locating a record, inserting a new record, reading a record, modifying a

record, deleting a record, and renumbering a record. Of course all of these operations require a file position to identify the file being processed, and a record position identifying the current record. In a similar fashion to standard file systems, the record position moves from record to record with each operation; the ability previously mentioned to compare two positions exists for all operations which can alter a record position.

A few general comments on record handling are required. For the purposes of providing direct access to records, each record in NAM is identified by a record key. To give the greatest flexibility in record handling, NAM does not dictate the form of the key, but accepts simply a variable length string as the record key. In any given file, each record must have a unique key and the records are stored in order by key, although the same key may be used for records appearing in different files.

Because different files are allowed to contain records having identical keys, some difficulties occur. For

example, in



if both files B and C contain records with key values from 1 to 5, then a listing of the higher level file A will list record 1 from C following record 5 from B, giving the appearance that the records are out of sequence. If the user were to ask for the record numbered 1 from file A, then a problem occurs since there are two such records. One very impractical solution would be to require all records in the aggregate to be uniquely numbered; such a restriction is almost equivalent to requiring the user to place all his records in one file. Since it may be costly to look for such multiple occurrences for every record search, NAM simply picks the record from the first subfile containing a record with that or a larger key value. If this causes confusion for the user, he is advised to renumber some of his records appropriately. (In this case he could renumber the records of C, starting with value 6).

It was mentioned in the preceding chapter that a method for providing several versions of a file without maintaining

separate copies of the file exists. Since in most cases two versions of a file will differ in only a few places, maintaining a separate copy of the file is quite wasteful of storage. A more efficient technique duplicates in the file the individual records that have been modified. When a record is changed in one version, both the old value of the record, and the new value, must be retained, and each must have an indication of which version(s) it belongs to. When a record is deleted in a new version, it must be retained for the older versions but must be marked as deleted in the new version. To implement this, each version is identified by an integer; subsequent versions have higher values 1,2,3,... As well, each record has two "version numbers". The first "version number" identifies the version in which the record was created; the second the version from which the record was deleted. Thus, to select from a file those records with a specific version number, it is sufficient to compare the creation and deletion version numbers of the records with that of the desired version. Records whose creation version number is less than or equal to the desired version and whose deletion version number is greater than the desired version are known to exist in that version. For most records, the two version numbers will not be required since most records will exist in all versions; thus by interpreting the omission of the version fields as indicat-

ing that the record exists in all versions, these fields need not be present on all records.

Turning to the actual record operations, in order to establish and remove record positions, two operations are provided. These operations, called "point" and "unpoint", are analagous to the locate and unlocate file operations. The point operation can locate a record absolutely by key, or relative to a previous record position. The unpoint operation simply removes unneeded record positions.

To point to a record by key, the user simply provides the point operation with the desired key, along with an appropriate version number; the point operation returns a record position for use in subsequent references to the record. It is useful, when the specified record does not exist, to return a pointer to the next record in sequence, along with some indication of this occurrence.

It is also necessary to be able to find a record by position. The point operation allows either the first or last record of a file to be located, or the one following or preceding a given record position.

The most commonly used record operation is the read operation, which retrieves a given data record. The read operation naturally accepts a record position to indicate which record to read and, like standard file systems, automatically changes the record position to the preceding or following record. (NAM allows operations in both directions for greater flexibility). The read operation also accepts a comparison record position to simplify handling of ranges of records. Thus the sequence to read a range of records is;

```
rpos1 = POINT(first record in range)
rpos2 = POINT(last record in range)
READ(rpos1, rpos2)
while  not at rpos2  do
   READ(rpos1, rpos2)
end
```

To insert a record, the key of the new record is provided to the insertion routine, which then inserts the record into the proper order. Since most insertions are logically relative to already existing records, the insertion routine can be given a record position which the new record is to be inserted adjacent to. Of course, the new record must have a key between the keys of the two adjacent records. Thus to insert a set of records the editor may insert the first one

by key only, or use the position of an already existing record; subsequent insertions are done using the record position of the previous insertion.

The delete operation is similar to the read operation. It accepts a record position, eliminates the record, and may move the record position from record to record in a similar manner to reading records. As for the file deletion operation, if any other record positions point to the deleted records, they are marked as invalid so that only an unpoint can be performed with them.

To alter an existing record, the rewrite operation is provided. It rewrites the changed contents of the record indicated by the record position. No option of moving to the next record is provided since a record is always read first before rewriting it; the read operation performs all necessary moving from record to record.

One operation not usually associated with keyed files is a renumber operation which allows the key of a record to be changed. This operation is provided specifically for source file editing; after many alterations a user often wishes to renumber his data in order to obtain an even progression of

record key values. For integrity reasons a record is not allowed to be renumbered to a new key value which would force the ordering of records to be altered. This complicates the renumbering process for the editor somewhat; in many cases it will have to renumber a file twice, in order to preserve the proper record order during the renumbering. Like the read routine, the renumber operation may move automatically from record to record with a second record position provided for comparison purposes to automatically stop the sequence.

Two other special operations exist. One counts the records between two record positions; the other indicates the relation between two record positions (i.e. which one identifies an earlier record in the hierarchy). Both have been provided only because they are useful to the editor.

3.6 SUMMARY

To summarize the interface to the NAM file system, NAM operates upon a hierarchy or network of files stored in collections called aggregates. A generalization of the concept of a file position is provided to indicate current

positions within the hierarchy. Means are provided for comparison of two positions to delimit ranges of files or records. Three levels of operations are provided to manage aggregates, files within aggregates, and records within files.

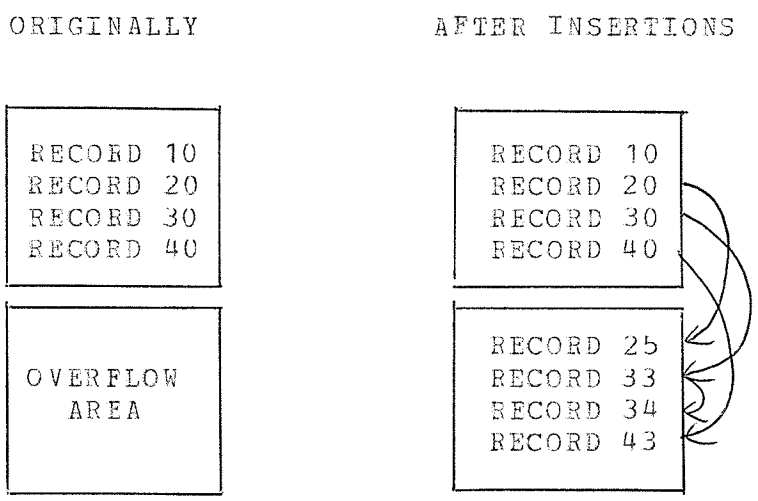
A more detailed discussion of the actual calling sequences by which the editor interfaces to NAM can be found in [FERCH 1978]. These details are too numerous to be included here and would tend to obscure the important details.

CHAPTER 4 - STORAGE STRUCTURE

Now that the facilities provided by NAM have been discussed, the implementation can be described. Most of the implementation details of such a file system are to a large extent mechanical and not worth mentioning here. This is not to say they are trivial, or that the implementation is a minor procedure. The implementation of NAM occupied two and a half years of the author's time, and required 30,000 lines of program text. However, much of the effort is of a fairly straightforward technical nature.

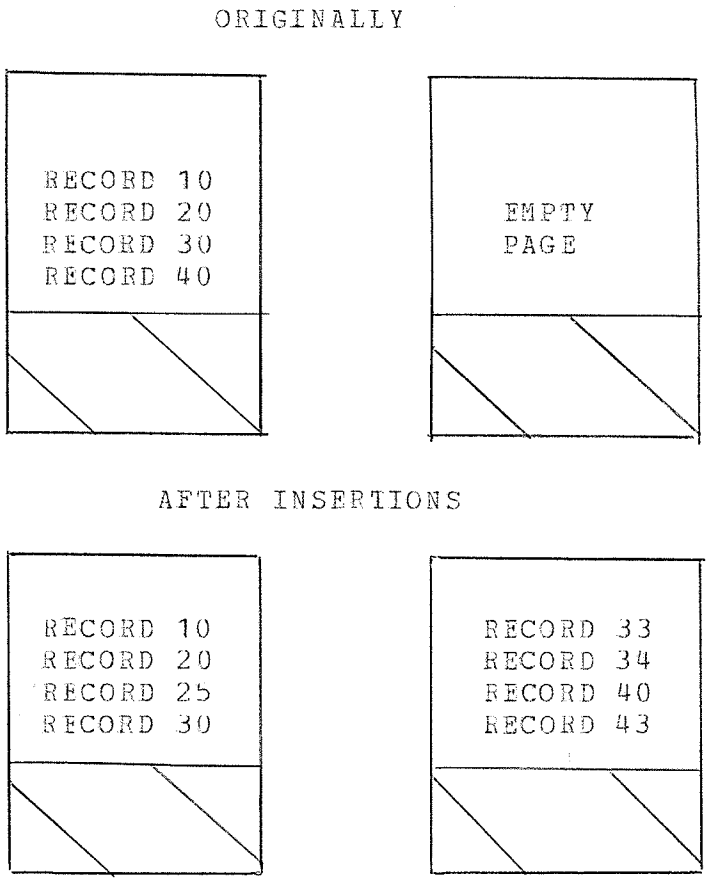
One major detail of the implementation, however, is the design of a storage structure for the data, and will be discussed in this chapter. There are many factors affecting the storage structures used. Chief among these is the ability to perform direct updating. Because of direct updating, regular sequentially stored files cannot be used, since insertions and deletions cannot be performed in such files. In order to accommodate such insertions and deletions, some free areas of space must exist. Then when additions are made, these free areas can be used to store the new data. Similarly, when records are deleted, the

space they previously occupied may become free, useable space. There are two basic techniques for managing such additions and deletions. In the first, the free space is kept in a separate area from the records themselves, called the overflow area. As additions are made to a file, records are inserted in the overflow area when there is insufficient room in the primary area. These records are then linked via pointers in the correct sequence, as depicted by the diagram:



As many changes are made to such a file, a large number of records must be placed in the overflow area, causing accessing to take longer since much jumping from the primary area to the overflow area and back is required to follow the chains of overflow records. Thus such an organization relies on the user to reorganize, or rebuild, his files

periodically to keep the number of overflow records to a minimum. Such a practice is often acceptable in a controlled data processing environment, but for source file editing, it is desirable to have the files remain well-organized without the need for explicit reorganization. A second technique divides a file into a set of pages. Each such page contains a set of consecutive records, along with some free space. In addition, some totally empty pages may exist. As additions are made, the free space within a page is used until the page becomes full. At this point, the page is split into two, by obtaining an empty page and spreading the contents of the original page over the original and the empty one so that each is then left partially full. This is shown in the diagram:



This latter technique is used in NAM to provide for direct updating, since periodic reorganization is not required.

Another major factor in the design is the wide diversity expected in the data to be stored. The degree of structuring employed depends to a large degree on the individual user, and causes a wide variation in the number of nodes in a hierarchy; some users may have only a few files, while

others may have a large number, on the order of a thousand. More importantly, the number of records in a file will also vary a great deal; a file containing job control language statements might have only a few records while a file containing text might have many thousands. Some users will take advantage of the substructuring allowed; others will stay with habit, keeping a small number of large files. Consequently NAM must be able to accomodate this diversity with little penalty to the user, either in access time or storage. Users will refer to files using simple unqualified names, so an organization is needed which allows an efficient search for a file; that is, which does not involve accessing all the nodes of the hierarchy. While keeping these considerations in mind, a major objective is to implement the operations so as to use the minimum number of input/output operations from auxiliary storage.

This chapter is divided into four parts. As a background to the design considerations, the nature of storage media is discussed. Then the storage structures are discussed, subdivided into aggregate related considerations, file storage considerations, and record storage considerations.

4.1 BACKGROUND

It is important to possess a familiarity with the characteristics of the storage media before beginning. The storage media used are magnetic disk units as exemplified by IBM's 3350 disk drive [IBM 5], and DEC's RP04 disk [DEC 2]. Disk storage devices are divided into regions called cylinders, which are further divided into areas called tracks, which are themselves divided into the blocks of data. There are two costs involved in accessing a block of data. The major cost is involved in physically moving the read/write head to the desired cylinder. This time is known as the seek time and typically takes on the order of 25 milliseconds. The other major time is that taken while waiting for the desired block of data to rotate to a position under the read/write head. This time is called the latency time and typically might be another 10 milliseconds. Thus the time taken to read or write data to and from a disk unit is measured in milliseconds, a very large unit of time when compared to the central processor speed, which is measured in microseconds or even nanoseconds per instruction. Thus the overwhelming consideration in achieving the best performance is to minimize the number of input/output operations, or data transfers as they are often called.

One aspect in minimizing the time taken for data transfers is to minimize the seek time for consecutive data transfers by organizing the data blocks to be physically close to each other on the disk. If two blocks are stored on the same cylinder and are read consecutively, then the second will require no seek time; the read/write head will already be positioned at the correct cylinder. However, placement of data on a disk is a function of the operating system; some systems, like UNIX [Ritchie 1974] allow the user no control of placement whatsoever, while others, such as IBM's OS/VS [IBM 1], require the user to specify in advance for each file the amount of space to be used, and then will attempt to allocate that space contiguously on disk. Another factor also affects this placement. In a multiprogramming system there is no guarantee that two consecutive data transfer requests by one user will be executed consecutively on the disk drive; another user's request may be executed between them, thus losing some of the advantage of contiguous placement, since the other user's request may cause the read/write head to be moved. For these reasons, placement of data on the disk is not a factor in the design of NAM; the user's data is logically stored in a contiguous fashion inside his aggregate - the actual placement of the aggregate on the disk drive is a

function of the operating system.

Another consideration of the storage of data on a disk unit is the size of data blocks used. Some disk drives, like DEC's RP04 disk, only allow data blocks which are multiples of a fixed sector size, in this case, 512 bytes. Other disks, such as IBM's 3350, allow the user to select a size. In either case, using fixed-size blocks (pages) across an entire aggregate is more efficient since then a straightforward addressing mechanism for locating and assigning blocks is possible. Most modern file systems employ fixed length blocks, as in IBM's VSAM [IBM 4], SYSTEM R [Astrahan et al 1976], and DEMOS [Powell 1977], to name but a few. The choice of a block size for these blocks is to a large extent operating system dependent. NAM operates in a virtual memory environment in which only portions of a user's storage area actually occupy main storage at a time. The choice of a block size is very much a function of the operating system's virtual memory scheme since block sizes just slightly higher than the virtual memory page size are very inefficient. Therefore NAM, which can operate within a wide range of block sizes, generally uses the largest size under the system's page size (4096) that fits evenly on the disk unit used. On a 3350 disk drive, this blocksize is 3648 bytes and all statistics for NAM will be relative to

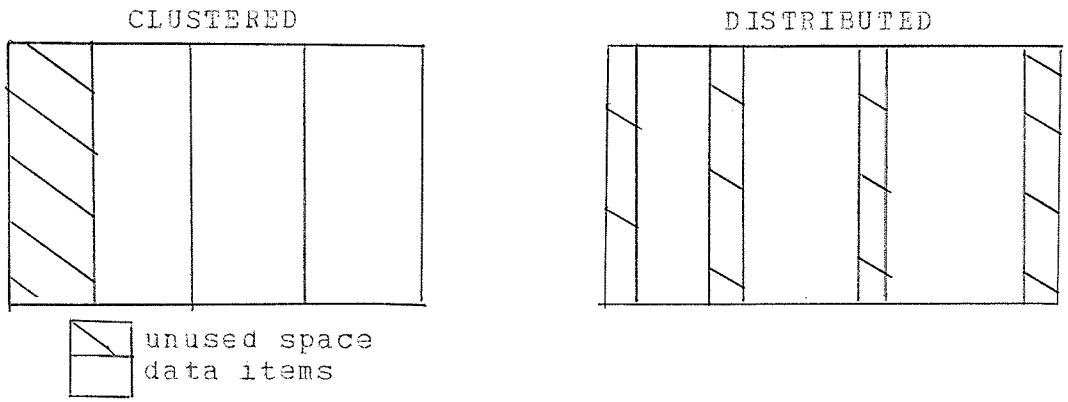
this choice of block size. This value will henceforth simply be called the NAM pagesize.

4.2 AGGREGATE STORAGE

An aggregate in NAM consists of two sets of information; directory information to describe the hierarchy, and the data records. These two sets of information are of course related; the data records belong to nodes in the hierarchy. However, to simplify the storage of both types of data, a common structure for storing them and managing space is utilized. An aggregate, in NAM, consists of a set of fixed-length pages, allocated on disk by the operating system. This set of pages is managed as a unit by NAM, and contains both the directory and data information.

Since the items to be stored in the pages are all of variable lengths and the pages are all of a fixed length, there exists some unused, or free space within each page. As mentioned, this free space is useful for accomodating future additions to the aggregate, but its presence requires a space management scheme.

There are two ways in which the free space within a page may be managed. Within each page are normally stored several items of data; there may be several directory nodes, or many data records, for example. As each such data item is modified it may expand its length or contract. When it expands, extra space is obtained from the unused space; when it contracts, the extra portion then becomes unused space. The two ways in which the unused space may be managed are either to collect it all together in one place within the page, or to leave it distributed in small pieces between the individual data items. The two approaches may be pictured via the following diagram;



The first of these techniques, clustering the unused space, requires that all data items be stored contiguously at one end of the page so that the unused space may be

collected together at the other end. When one of the data items contracts, or expands, then other data items must also be moved. For example, in the first diagram above, if data item 3 is shortened, data items 1 and 2 must be moved up to compensate; if item 3 is lengthened then items 1 and 2 must be moved down to make room. This moving requires processor time; a simple change of adding one character to the last record may require the moving of several thousand other characters to make room. This technique, however, is also very simple to manage - only one area of free space exists; its location and length may be described by two values.

The second of these techniques, distributing the unused space, has the advantage that many expansions and all contractions require no movement of other data items. However, some expansions may require more space than is immediately adjacent; these expansions require other items to be moved to acquire the unused space adjacent to them. In addition, a more complicated space management scheme is required - there exists a variable number of variable length free areas to manage; keeping track of the total free space is slightly more difficult.

The technique used by NAM is the former. Since the pages

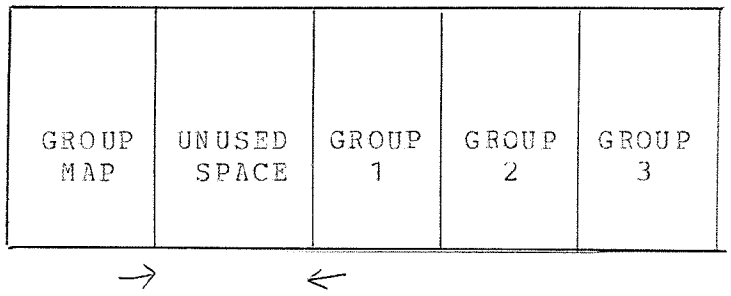
are expected to contain a fairly large number of items (as many as a hundred), it would be expected with the distributed scheme to have a fairly large number of rather small items of unused space. The management of these small items would require a fair amount of space and in addition it is expected that a fairly high proportion of expansions would require more unused space than would be immediately adjacent, particularly in view of the fact that users add data to their aggregates much more often than they remove data. This, in conjunction with the simplification of the algorithms needed to manage the space, and in conjunction with the presence in the hardware of single instructions to move large areas of data, led to the adoption of the clustered scheme. On the machine used (IBM /370 [IBM 6], it is easier to move data to a location whose address is lower; in addition, more expansions of data than contractions are expected; thus the unused space is kept at the lower end of the page, so that expansions involve movements from higher to lower addresses.

For the purposes of allocating data items to pages within the aggregate, it is important to know the amount of unused space existing in each page. This information could be obtained by reading each page in turn, but this is obviously a very costly way to obtain the information. NAM therefore

keeps a map which contains one entry for each page in the aggregate. Each such entry indicates the amount of unused space in the corresponding page. This free space map is kept in the first page of the aggregate, which is always resident in main memory in NAM, when the aggregate is being accessed. Thus NAM is able to locate all areas of unused space within the aggregate immediately without performing any data transfers.

One other facet of space management exists. Each page in the aggregate may contain a variable number of data items, each of variable length. Some means of locating these data items, or groups, as they are called in NAM, within the page is needed. In order to locate the groups each one is identified by a unique number, the group number. This group number is used to index a table stored in the front of the page. Each entry in the table contains the offset within the page and the length of the corresponding group. This group map has a variable length; as groups are added to the page, entries are added to the group map which extends into

the bottom of the unused space, as illustrated here.



The first entry in the group map is reserved to describe the free space within the page.

This, then, constitutes the space management scheme used by NAM. Each page contains a set of groups, a group map, and some unused space. The location of the free space and each group within the page is given by the group map. In addition, NAM retains the amount of free space in each page through the global free space map. This technique was developed independently of, but is quite similar to, that used in the System R relational data base system [Astrahan et al 1976].

The use of a group map also assists in a technique for addressing individual groups. Each group can be located

through two pieces of information; the number of the page containing the group, and the group number of the group within the page. This 2-tuple remains constant regardless of the position of the group within the page; it varies only if the group is moved from one page to another.

One important advantage of the space management and addressing scheme used in NAM should be noted here; the origin of the groups in any one page is unimportant. They may, for example, be constituent parts of unrelated files. This allows for the efficient handling of very small groups of data, such as a file with only a few records. The data portion of such a file may be stored as a single group and consequently a page may contain a number of such files. This is an important consideration in order to avoid degraded performance when small files are used. Many file systems, since they treat each file as an independent entity, have a minimum amount of disk space that can be allocated for each file. For example, in UNIX or RSX, a file occupies a minimum of one sector on disk (512 bytes); in IBM's OS/VS, most files require at least one track of disk space (7000 to 20000 bytes).

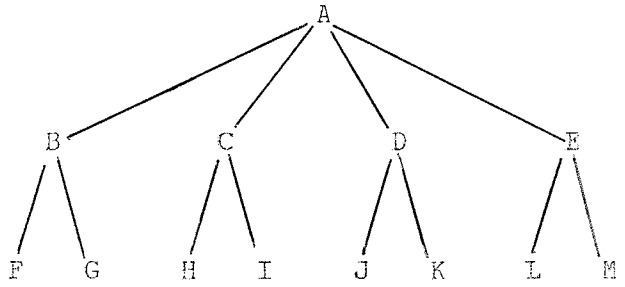
4.3 FILE STORAGE

Within the aggregate, two distinct sets of data must be stored; the directory information representing the hierarchy, and the data records. These two require very different storage techniques and hence are discussed independently. Considering the file hierarchy, it is apparent that each node in the hierarchy will have to be stored, along with information relating the nodes together and relating nodes to their data records. There are thus four basic considerations in storing the nodes; the methods by which a node is related to its data records, the means by which nodes can be located by name, the organization of the nodes within the aggregate, and the methods by which nodes are related to each other.

The means by which a node is related to its data records will be discussed in the next section on record storage. For now it is only important to realize that each node possessing data records will contain an index, which will be stored with it.

4.3.1 LOCATING BY NAME

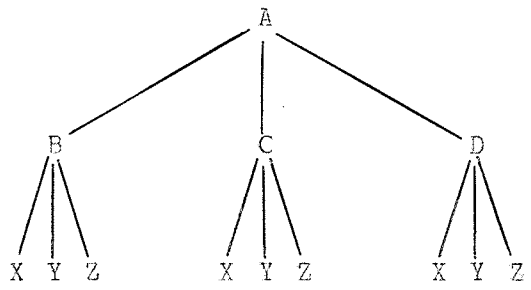
Since the method by which nodes are located by name affects the other two considerations, the organization and relating of nodes, it is discussed first. In most file systems which support a hierarchy, only totally qualified names are permitted. This simplifies the search somewhat since the name indicates at each level in the hierarchy which subnode to search. For example, in



the name A.D.K tells the file system to first select subfile D from A, and then to select subfile K of D. Only three nodes need be examined. However, if the user only specifies the name "K", then searching the hierarchy from the top down requires much more effort. At the first level, the file system would have to examine all subfiles B, C, D, and E, and would then have to examine each of their sons in turn until node K were found. Since duplicate names are allowed

(file L could also be given the name K), the file system could not stop when node K were found, since it would still have to check the rest of the files to see if the supplied name were ambiguous. Since this search could be very time-consuming for large hierarchies, some other mechanism is needed.

A much more efficient technique is to use an index which locates nodes without using the hierarchy itself. For example, if an index were to exist which could indicate the locations of all nodes with a given name, then a direct search to these nodes could be made. When partially or totally qualified names were given, the rightmost name could be used with the index to locate a node with that name, and then the hierarchy could be searched from this node up to check each path against the given qualifiers. For example,



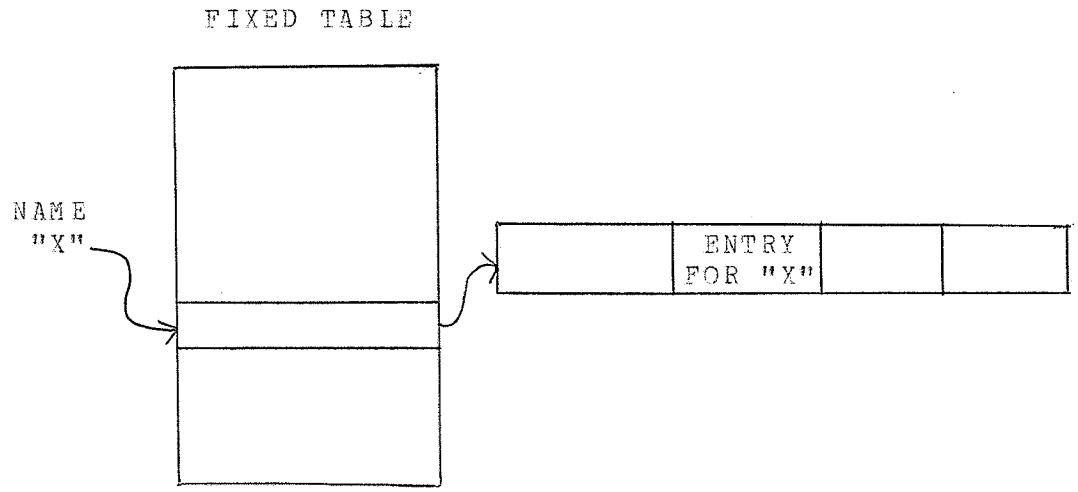
if the name C.Y were given, the index could be used to locate all nodes with name Y. Then the ancestors of each

may be examined to see which path or paths would satisfy the qualified name. It would be expected that most names would refer to nodes whose names were unique, so that only one path would in fact be examined in most cases.

To implement such a scheme, there are two considerations. First of all, each node must contain a pointer to its father. This pointer will be discussed later. Secondly, an index to the nodes must be created. This index should have associated with it a reasonably efficient search algorithm, at least in comparison to the equivalent tree search algorithm; its purpose is to avoid an entire tree search, and thus its purpose would not be met if it were as expensive to search as the original hierarchy. It will, of course, have to contain one entry for every node in the hierarchy; the user may wish to locate any node by name. It should be small enough to fit in main memory so that no data transfers are needed to search it; since this information must be retained in the aggregate, it is desirable to be able to fit it into the first page of the aggregate since this page is kept in main memory to keep the free space map accessible.

Various techniques for providing such an index exist.

called a chained hash scheme. These lists of addresses are built dynamically; each node which is transformed to a given entry in the fixed table by the hashing function will have an entry created in the corresponding list containing its address, as in the following diagram.



Given a hierarchy with, for example, 1000 nodes, the average list length expected would be approximately 10 entries long, since the fixed table has 103 entries in it. This would imply that for each search by name 10 nodes would have to be examined. To reduce this number, each entry in the list includes a second hash value. Thus, two hash values are computed using the name; the first selects an entry in the fixed size table, the second selects an entry from the variable length list. This yields a value of 26,368 unique hash values in NAM, since the second hash

value takes on all values from 0 to 255 (one byte of storage). However, it is still possible that two different names may possess the same two hash values; such "collisions" require NAM to examine an extra node for name searches upon either of the two names which collide. The expected number of such collisions is very small, though, since there are 26,368 distinct possible values; a later chapter will provide statistics on the number of such collisions. Of course, two nodes with the same name will always generate the same hash value. Whenever such a node is searched for, both entries in the hash list will have to be examined.

It should now be clear how a node may be located by name. Two hash functions are computed on the name. The first provides an index to the fixed hash table; the second selects a subset of elements from the corresponding list. For each such element, the corresponding node is read and the name is checked (due to the possibility of hash collisions). For each node with the correct name, the path up through all its ancestors to the root node is searched. There are four possible results of such a search. In the first of these, a path may be found for which the given name is a totally qualified name; each ancestor in the path matches the corresponding qualifier. Such a path is

accepted as the result of the search, because the file with such a name is unique. The second possibility is that a single path is found for which the given name is a partially qualified name. Such a path is also accepted. The third possibility is that several paths may have been found which match the given name. If all of the paths represent different paths to the root for the same network node, then the first such path is accepted. Otherwise, the given name is ambiguous. A fourth possibility exists in that no path matching the given qualifiers exists. Such a request, like the previous ambiguous case, is in error.

One slight modification to the above search strategy exists. It was mentioned earlier that a facility equivalent to the concept of a current directory for delimiting name references to subportions of the hierarchy exists. This facility is implemented in NAM by allowing the user to specify a restricting, or qualifying, file position to delimit searches by name. When such a qualifying file position is given, all name references are taken to refer only to files which are subfiles of the given restricting node. The only modification to the above search strategy required to provide this facility is that instead of a path being followed up to the root node, the search is stopped when the restricting node is reached. All paths which do

not encounter the restricting node are ignored; the aforementioned four possibilities are then applied to the subset of paths which terminate with the restricting node.

4.3.2 ORGANIZATION OF THE NODES

The next basic consideration in the storage structure used for the hierarchy is the organization of the nodes. Several constraints must be realized before a structure may be decided. First of all, each node possesses the following information; it contains a name, some information relating it to other nodes (to be discussed in the next section), some information relating it to its data records (discussed later), and a list of attribute and password fields. All of these pieces of information may easily be collected together into one data structure; no difficulty exists in their storage. A node may simply be viewed as a single entity, which is subdivided into its internal components. Such an entity in NAM is most easily stored as a single group; for the purposes of space management and addressability, the entire node may then be treated as a unit, through the page and group number mechanism already mentioned.

A second constraint, however, affects the placement of

these groups within the aggregate as a whole. It is necessary to understand the frequency of access to the nodes to decide where they should be placed. Most of the time, the user operates with a small portion of the hierarchy. He performs a set of editing commands upon one section of his data and then moves to another section. He does not very frequently jump from one place to another, at least relative to the number of data record operations that are being performed. Thus, many data record operations are generally performed upon each file before the user moves to a new file. For these data record operations, the user will need to access only the one node containing the given records. Fairly often, but not nearly as often as he accesses a set of data records from the file, the user will search for a new file by name, when he switches from editing one file to editing another. Even less often, the user will perform a tree traversal of part of the hierarchy, generally in order to list the names of his files. Thus the storage structure should be organized with the view that the user spends most of his time only requiring access to one node of the hierarchy, and should be somewhat less optimized for searches by name, and less again for tree traversal from file to file.

There are three ways in which the groups containing the

nodes of the hierarchy may be placed within the aggregate. Each such group may be placed randomly wherever there is a page with sufficient room, each node may be stored together with its data records, or the nodes of the hierarchy may be collected and stored together.

For the purposes of the most common access to nodes there is very little difference among the three placements. Placing a node with its constituent data records is slightly more efficient since the node and the data records may be read into main memory with one access. However, very often the data records for a node will occupy several pages of the aggregate; the node in this case may only be placed in with one portion of its data records; the user must access that particular portion of his data to realize any gain over placing the node in a random place within the aggregate. In addition, the node need only be read once and then may be stored in main memory for an entire set of operations upon the file, so that an extra data transfer to read the node is not particularly important.

Turning to the less frequent accesses to nodes, a more substantial difference between the placement possibilities exists. For the purposes of performing lookups by name,

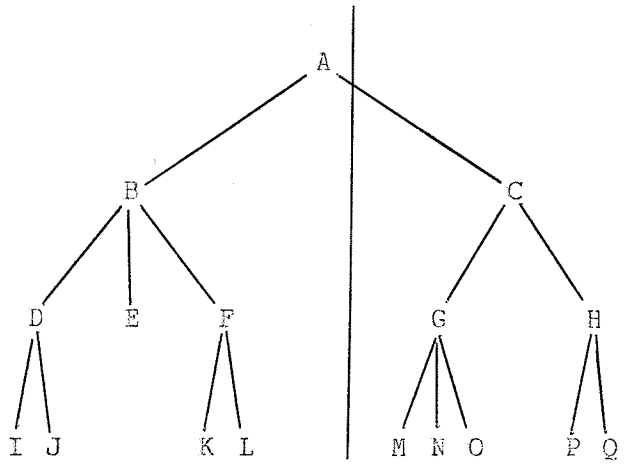
each reference by name requires the examination of each ancestor of the node being located. If the nodes are scattered throughout the aggregate, either randomly or with their data records, many data accesses to different pages will be required to perform a name search. Collecting the nodes together into a small set of pages will be much more efficient for the simple reason that each read from the disk unit will read a set of nodes; the chances of the next node to be examined already having been read are greatly improved.

In a similar fashion, the least frequent access to nodes, to perform tree traversal, is also more efficient if the nodes are collected together into a small set of pages. When performing a tree traversal a set of nodes is examined in succession; this operation obviously requires fewer reads from disk if the nodes are collected together into as small a number of pages as possible.

Thus in NAM the nodes of the hierarchy are collected together into one set of pages. The small losses in efficiency resulting from sometimes having to read two pages in order to have both a node and some of its data records memory resident is more than compensated for by the very

large difference in efficiency for searches by name and tree traversal operations. These pages in which the node groups are stored are called directory pages, and are not allowed to contain any data records; just nodes. The directory pages are allocated as needed from the set of pages belonging to the aggregate and thus the number may increase as needed; there is no fixed number of such pages.

Given these directory pages containing sets of nodes, there is still the consideration of the order of placement of the nodes among the directory pages. The nodes may be placed randomly within the directory pages wherever there is space, or they may be ordered in some way. NAM attempts to order the nodes in such a way that name searches operate more efficiently. When a new node is added to the hierarchy, it is placed along with its brothers in the page containing its father. If this page does not possess sufficient room, then a new directory page is allocated and the contents of the page are split across the two directory pages. This split is performed in such a way that the leftmost half of the portion of the hierarchy stored in the original page is left in that page, while the rightmost half is moved to the new page. For example, if a directory page contained the following portion of a hierarchy,



then the vertical line represents a possible split position; nodes to the right are moved to the new page. In this way, nodes are placed with their fathers as much as possible; in the above example, all of the leaf nodes and their next two levels of ancestors are stored in the same page; in practice many more nodes than shown here will be stored in one directory page.

4.3.3 RELATIONS BETWEEN NODES

One consideration remains regarding the storage of nodes. Some means of indicating the relations between nodes is needed. Some constraints exist to limit the choice of technique; name searches require that pointers from each node to its fathers exist, the ability to handle network files must be present, and the structure must permit tree

traversal operations, particularly because the hierarchy is intended to be used for the purpose of subdividing files into smaller pieces.

Many different techniques have been used to represent tree and network structures [Martin 1975]. However, most of these may be rejected due to limitations on one or more of the above constraints. The basic technique used to represent a tree structure is simple physical contiguity; the nodes are simply placed in left to right order. However this technique obviously is not adequate, both for name searches and for implementing networks of files. Structures based on the use of brother or twin pointers in which a node contains a pointer to one or more of its brothers likewise are not particularly applicable; a network node may have several sets of brothers. Using such a structure would require the use of a variable number of brother pointers to handle network nodes. This would be a very complicated structure to manage.

The approach used in NAM is to include with each node pointers to its fathers and its sons. This combination allows tree traversing both downwards from a node to its sons, and upwards to its fathers. Network nodes are handled

by using multiple father pointers. Movement from a node to its brothers for the purposes of a tree traversal may be performed by examining the son list from the node's father. Since NAM attempts to store nodes and their fathers together in the same page, this usually requires no extra data transfers.

Thus the representation of the hierarchy may be summarized as follows. Each node is represented in the aggregate by means of a group containing the name of the node, pointers to all its sons, pointers to its fathers, its attributes and passwords, and some information (not yet specified) relating the node to its data records. These groups are collected together into separate pages within the hierarchy and are arranged in such a way to keep each node in the same page as its father. To perform searches by name, a hash table is stored in the first page of the aggregate.

4.4 RECORD STORAGE

A major consideration in the storage of data records, particularly those which are most often accessed in a sequential fashion, is the collection of such records into blocks. Consecutive records are grouped together into blocks so that one data transfer may be used to access a large number of records. The larger these blocks, the fewer the number of data transfers, so that it is important in NAM to maintain the largest possible such blocks, given the restriction of the NAM pagesize. To provide blocking of records, consecutive records are stored together in groups; that is, a set of adjacent records from a file are collected together and are stored and managed by NAM as a single group. The records of a file containing source programs frequently begin with a number of blanks, especially if indentation is used to display the program structure. Thus, NAM compresses out leading blanks (and assumes that the editor does not supply any trailing blanks) so that more records may be stored in each group. This provides a simple way of increasing the effective blocking factor.

To build the data groups, NAM attempts to make them as large as possible in the following manner. When the first

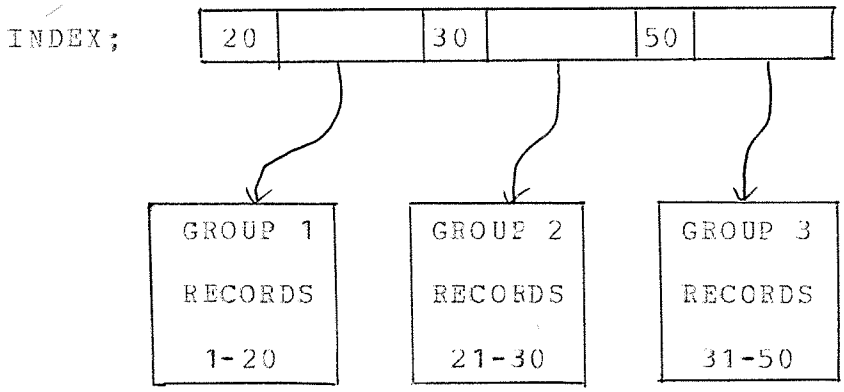
record is inserted into a file, NAM assumes that the new data file will occupy at least half a page, and thus attempts to find a page that is at least 1/2 empty, if possible; or if not, the largest remaining space, providing that it is large enough to contain at least several records. Even if the data file does not occupy as much as half a page the remaining space within the page is still usable for future expansion of the group, or even to store other groups of records. As records are added to the end of the file, they are added to the group which is allowed to expand until the page has less than 1/16 of its space left as unused space. This 1/16 of the space is left for future expansion of data within the group; the actual figure of 1/16 was determined by experiment.

If the user continues to add records to the file once the page has filled, then one of three actions takes place. If the new record is added to the end of the file, a new group is created and the sequence continues. If the new record lies within the file, then the corresponding group must overflow to a new page, in one of two ways. If the group is fairly large, larger than 1/2 a page, it is split into two groups occupying parts of two pages, instead of the previous one page. Each of these two pages will now have some unused space, so that insertion may continue. If, however, the

group is fairly small, i.e. less than 1/2 a page, splitting it into two would result in the file possessing rather small groups, with a corresponding increase in the number of data transfers required to read the records of the file. Thus for such small groups, an attempt is made to find a page within the aggregate which has sufficient free space to contain the entire original group plus some extra space (half as much again) for expansion. If such a page can be found, the entire page may be moved and then may be expanded, thus leaving the largest possible group. Of course, if insufficient space exists to move the entire group, then it must be split into two, in the same way that larger groups are.

In order to keep the groups of data records as large as possible, NAM also has the facility to recombine groups after records are deleted. When a record is deleted from a group, the two groups immediately preceding and following it in the same file are examined. If sufficient space exists in one of the pages containing the original group, or in those containing the previous or following groups, then two groups may be recombined into one larger group. In this way, the data file is not split into smaller groups after a series of insertions, followed by a series of deletions. The data groups are split into smaller groups on insertions,

the set of groups containing its data records. This means must provide both for sequential retrieval of the records, and random access of a particular record by its key. Since each record in a particular file possesses a unique key, this retrieval problem is an example of a standard retrieval problem, known as primary key retrieval, to which much effort has been devoted [Knuth 1973]. Fortunately, source files such as NAM manages, usually do not possess very large numbers of records. While databases tend to contain tens or hundreds of thousands of records, it is very unusual to find a unit of program text containing more than a few thousand records; with the subdivision allowed by the file hierarchy in NAM it would be more usual to find files which consist only of hundreds of records. Thus, multiple level dynamic index techniques, such as AVL trees [Foster 1965], B-trees [Bayer et al 1972], or indexed-sequential techniques, such as IBM's VSAM [IBM 4], are not necessary. A simple, one-level index, containing one entry per group of records in the file, is adequate. Each such index entry contains the record key of the highest record in the associated group, and the address of the group. The collection of these index entries is then stored, in order by record key, in the node for the file, as depicted by the following diagram.



To locate records sequentially, successive index entries are used to obtain the addresses of the data record groups; to locate a record by key, the index is searched to find the first index entry whose record key is higher or equal to the given key; the record, if it exists, will be contained in the corresponding group. Access to a record by key may thus be performed directly, once the node containing the index to the records has been located.

Thus, because of the small size of source data files, record storage is quite simple. Each node possesses a set of index entries, each of which contains the address and highest record key of a corresponding group of data records.

CHAPTER 5 - INTEGRITY

In a file system such as NAM there is a greater chance for a loss of integrity than with other file systems. In the first place, NAM performs direct updating and because of this, many data items have to be linked together, rather than simply being entered in sequential fashion. This means that often more than one change to the aggregate is required for one file system operation. If, for any reason, only one of these changes is performed, the data within the aggregate could be left in an inconsistent state. Regular file systems operate upon only one file at a time so that only that one file could be damaged by a failure. However, NAM operates upon an entire set of files within the aggregate at once. A potential exists for a failure to damage an entire set of files.

The chapter is divided into five sections. First of all the types of failure are discussed. Then, various techniques for handling these failures are presented. The last three sections discuss the techniques used specifically in NAM, divided into sections on aggregate-specific techniques, file-specific techniques, and finally record-specific

techniques.

5.1 CAUSES OF FAILURE

There are two major categories of system failures. The first of these is characterized by incorrect data being written into the file. The second type of failure is one in which the data is correct, but not all of the data is present.

5.1.1 CAUSES OF INCORRECT DATA

The major cause of incorrect data in the user's aggregate is a coding or design error in the file system itself. Such errors are invariably present, at least in its early life, in a file system which is as complex as NAM. Another possible cause of incorrect data is the overwriting of memory areas used by the file system owing to errors elsewhere (in some other program, or in the operating system itself). Normally the operating system protects the memory areas of one user from other users but errors within the operating system itself can damage users' areas of memory. Fortunately, such errors are very rare; in any case, nothing can be done, in the file system, to prevent them. Similar-

ly, errors in the actual hardware can result in incorrect data. Much checking is performed within the hardware to detect such errors. For practical purposes, the possibility of hardware error can be ignored.

5.1.2 CAUSES OF INCOMPLETE DATA

Most of the integrity problems are caused by some failure which causes incomplete data to be written. For example, if two data transfers are required to perform an operation, such as creating a node in the hierarchy, then it is possible for some event to occur between the two transfers so that only the first one is performed. This can cause inconsistent data to be present. For example, while creating a node in the hierarchy, the node itself may be written, but the pointer to it from its father may not be written.

Such incomplete data may result from many causes. A coding or design error in the file system may result in the omission of a required data transfer. Similarly, the operating system or hardware may err and neglect to perform a data transfer it has been requested to perform, although such a cause should be extremely rare. A more common cause of incomplete data is a power failure which causes the

machine to halt between two successive data transfers. Similarly an error somewhere in the operating system may cause the system to halt. A broken connection on the communications line to the user's terminal may cause the system to terminate all activity for the user. Finally, a human error may also cause the system to halt unexpectedly; as, for example, if the operator stops the machine.

5.2 RECOVERY TECHNIQUES

This section discusses recovery techniques in general; the specific application of these techniques in NAM will be discussed in later sections. The techniques discussed are divided into two types; those that are used to prevent problems from occurring, and those to recover from such problems afterwards.

5.2.1 PREVENTION METHODS

It was mentioned that both general categories of failure, causing incorrect and incomplete data, could be caused by incorrect design or coding of the file system itself. While the design and construction of correct software is a topic which is outside the scope of this thesis, some general

comments are applicable. The production of error-free software is greatly facilitated by the division of the software into functional modules. For example, in NAM, only one routine performs the function of adding a piece of data to a group; once this one routine has been debugged, one may add data to groups with reasonable confidence that the function will be performed correctly. In NAM, each such module performs extensive checking on its input parameters, as well as on the actual data within the aggregate. If any of these checks fails, a diagnostic memory dump is taken and a close operation is initiated so that further, possibly incorrect, operations may not be performed until the internal memory areas are reinitialized.

In order to minimize the impact of system failures, one possible technique is the ordering of the transfers to disk that occur. For an example, when a data item is to be removed from one list and is then to be placed on another, if the two operations are not ordered correctly, the transfer placing the data on one list may occur, without the corresponding transfer to remove it from the original list. This may cause inconsistent data, since each list may indicate that the data item belongs to it. If the data transfers were performed in such an order that the item was removed from the first list in the first transfer and then

added to the second list in a second data transfer, then no inconsistency could result from a system failure. At worst, the data item would not exist on any list but it could not exist on both.

As an example of a file system which does not utilize this technique, and which suffers problems as a result, the UNIX operating system uses linked lists to manage the allocation of blocks on disk. Each block may either exist on a list of unused blocks, or it may exist on a list indicating its ownership by a file. After a system failure some blocks are often left on both such lists, since UNIX does not order the data transfers appropriately. Inconsistencies such as these sometimes make it impossible to restart the system; in these cases the system must be recovered from an older backup copy.

Another technique exists for the prevention of integrity problems. Before some data is to be modified, a separate copy of the data may be made, and the changes may be made to this separate copy. After all the changes have been performed to a set of such copied data, any pointers to the old data may be modified to point to the new altered data, and then the old data is deleted. If a system failure

occurs, then the old consistent data is still pointed to. This method was not adopted anywhere in NAM for one simple reason: copying the data is quite expensive; cheaper techniques were found to work.

5.2.2 RECOVERY METHODS

Recovery from problems caused by system failures may be accomplished in the following way. A special routine may be provided which is called, either manually through operator request, or automatically by the system, after an operating system error, or when power is restored after a power failure. Such a routine may finish any incomplete operations which were in progress when the system was stopped, using data still remaining in main memory. However, such a technique requires both the existence of non-volatile memory which keeps its contents during a power failure, and also requires special support by the operating system to call the routine after a failure. Neither of these exists in the particular environment used by NAM, so that this technique is not applicable to NAM. In addition, after an operating system failure, there is some question as to the consistency of any operating system lists or control blocks; such a recovery technique may depend upon the existence of control information which was damaged by the original problem that

caused the system failure.

However, there is a way in which an equivalent method can be applied. Before a file system function requiring many data transfers is executed, it is possible to first perform an extra transfer to write out both an indication of the operation to be performed, and a list of all parameters affecting that operation. The completion of this data transfer is then awaited before the operation is started. If a system failure occurs during the execution of the operation itself, then the indication of the operation in progress can be examined when the system is restarted. The associated parameter list can then be used to restart and complete the operation. When the entire operation has been successfully completed, a final data transfer is performed to erase the indication that the operation is in progress. It must be realized that such a recovery procedure is complicated by the fact that the operation may have been partially completed before the system failed; the recovery routine must be able to cope with any partial completion of the operation and continue from that point.

If redundant information is present and one copy of the information becomes corrupted, the other copy can be used to

recover it. This method is widely used in NAM. For example, the free space map is redundant; each page also contains the amount of free space present in the page. Thus, if the free space map does not get written properly after changes have been made to other pages which alter it, it can be recovered from the data within the pages themselves.

5.3 AGGREGATE INTEGRITY

The specific techniques used to keep the aggregate as a whole safe will now be discussed. Three such global techniques are used; writes to disk are ordered, redundancy is used to recover the free space map, and flags are set to indicate aggregates which are active.

The most important technique for providing integrity in NAM is that of ordering writes to disk. A general buffer management scheme has been implemented which controls the buffers into which pages of an aggregate are read, and from which they are written. This routine also permits a calling routine to request the writing of pages in a certain order. In order to minimize the number of data transfers, NAM allows additional changes to such ordered pages, provided

that such changes are made only to the last of the ordered pages. However, if a second routine asks to change an ordered page which is not the last of a set of ordered pages, or asks that certain pages be written in a different order than requested by an earlier routine, then NAM will write the entire set of pages out first, to guarantee the integrity of the first operation, before allowing the second to continue. In this way, NAM combines page ordering with the minimization of the number of data transfers: even though a set of pages is ordered for writing, other changes may be made to the pages until a conflicting request is issued. The specific places where page writes are ordered are discussed in the next two sections.

The management of space within the aggregate involves one potential integrity problem. There are two places which indicate the amount of free space within a page; the free space map and the page itself. It is possible that these two values could differ because a page was modified and written without the change in the free space map being written. In addition, changes to the amounts of free space in the pages occur very often. It would be very expensive to write out the free space map every time the length of a group in a page was altered. For this reason, changes in the free space map do not cause rewriting of the page

containing it although this page may be rewritten due to other changes, or when the aggregate is closed. Thus, the free space map could easily be incorrect after a system failure. To handle this, NAM uses the free space map only as an approximate value; to find some free space, the free space map is read, but when a page supposedly containing an appropriate amount of unused space is found through a search of the free space map, the page itself is read and the actual unused space within the page is used. If, owing to an incorrect free space map entry, this space is insufficient, the search for space using the free space map is continued. In addition, every time a page is read into main memory, the corresponding free space entry is set to the correct value, so that errors in the free space map are corrected as pages are reread.

In addition, one other aggregate related integrity measure is taken. When an aggregate is opened, an indication that the aggregate is open is written into the first page; this "aggregate in use" indication is not removed until the aggregate is closed in a normal fashion. Thus if a system failure occurs while an aggregate is open, the indication that the aggregate is open will remain on. When the aggregate is later accessed, this indication may be examined and some recovery actions may be performed. Currently, the

only action performed is to send a message to the user that the system failed during his last session; it is planned that eventually a validity check of the aggregate will be performed when the aggregate is opened. This validity check will be able to correct such errors as incorrect free space map entries, or other problems within the aggregate.

5.4 FILE INTEGRITY

Most of the operations which manage the file hierarchy implement some integrity measures. This occurs since the nodes are related to each other via bidirectional pointers. If two nodes are stored in different pages, then to link them together always requires two operations, one for the pointer in each direction. Thus if a system failure occurs while linking two nodes, then only one of these pointers may have been established. Another problem area is the hash table; when a new node is added, or a node is renamed, the name of the node and the hash table may not be consistent with each other if a system failure occurs. Yet a third problem area involves the function of splitting a directory page into two to make space for a new node. In this case, a system failure could leave a large number of invalid pointers, due to the large number of changes to be made to

the aggregate in order to move a set of nodes from one page to another.

Turning to the first of these areas of concern, two techniques are used together to recover links between two nodes. A very important principle used in all file operations is to write no pointers to data until the data referred to has already been created and written out to disk. In conjunction with this, links to existing nodes are always ordered in such a way that the father to son pointer (henceforth called the son pointer) is established before the corresponding son to father pointer (called the father pointer). In this way, tree traversals through the hierarchy can always be performed without encountering invalid pointers; each son pointer will point to a valid node. The existence of pointers in both directions between nodes both causes the integrity problem and provides the solution, since the redunancy provided by the two pointers can be used to recover a missing pointer. Whenever the locate operation positions to a son of a node the existence of the corresponding father pointer is checked; if this pointer is missing it is simply inserted. In this way, a tree traversal through the file hierarchy restores any missing father pointers. The son pointers are all known to be valid because of the ordering of the writes for each operation.

The hash table is kept consistent using similar techniques. The hash table entry for a node is always created only after the node itself has been successfully created and linked to its father. Thus no inconsistent hash entry pointing to a non-existent node may exist. However, if the system were to fail between the creation of the node, and the creation of the corresponding hash entry, it is possible that a node may have no corresponding hash entry. Again, redundant data is used to create the required hash entry; whenever a node is read by the locate operation, its hash entry is checked. Any missing hash entries are then added automatically. Thus a tree traversal through the hierarchy will correct both missing father pointers, and missing hash entries. Such an action is usually the first action a user takes after an unusual event, such as when the system responds that his file is not found (due to the missing hash entry). It is planned that a later version of NAM will automatically insert any missing hash entries and father pointers after a system failure by using the "aggregate in use" indication mentioned in the previous section. Whenever an aggregate is opened, this indication will be checked; if the system failed during the previous use of the aggregate, then a traversal through the hierarchy will be performed automatically to insert any missing information.

The third area of concern, that of performing directory page splitting, is somewhat more complicated. All of the other operations upon the file hierarchy can be handled by carefully ordering all write operations to disk, followed by reinserting any missing items using data redundancy. However, the operation of splitting a directory page involves the moving of a set of nodes to a new page; each such node will change its address in the process. Thus pointers to any of these nodes must be altered. Some of these pointers may be son pointers, others may be father pointers, and still others may be the pointers in the hash table. There is no possible ordering of operations that will always leave the son pointers correct, with only the father or hash table entries invalid, since the movement of any one node may require altering both son pointers from the node's ancestors, and a father pointer for each son. The solution used in this case is that of writing an indication of the operation in progress before starting the operation. Along with this indication, stored in the first page of the aggregate, is stored an identification of the page being split, the split position within the page, and the new page to which some of the nodes are to be moved. This indication is not rewritten until the split is completed; if a system failure terminates the operation, it is automatically

resumed and completed the next time the aggregate is used. In this way, splitting of a directory page is completely safe, at the cost of two extra data transfers to set and reset the operation in progress indication.

Now that the general techniques used by the file operations have been discussed, the specific details for each particular operation may be mentioned. The operations of locating to a node, unlocating from a node, and adding or removing attributes or passwords, have no integrity concerns, since either no data is modified or else only one item, involving only one data transfer, need be changed. The operations which will be discussed, then, are the create, link, delete, rename, and move operations. Creation, linking, renaming, and moving all simply order the data transfers appropriately; deletion uses more complicated methods. Copying of a part of the file structure is not mentioned since it simply uses the create and link operations to build the new part of the hierarchy required.

Considering the create operation, to create a new node, it is apparent that three basic actions must occur. The new node itself must be created, the son pointer must be added to its father, and a hash entry must be created. These

three functions are ordered by first creating the new node, then by adding the son pointer to its father, and then by creating the hash entry. If a system failure occurs between the creation of the node and the creation of the son pointer, then the space occupied by the new node will be lost since no pointers exist to it. This small amount of lost space may be recovered during a later operation of copying the aggregate to a new location, or during a validity check when the aggregate is next accessed. A failure between the last two functions, creating the son pointer, and the hash entry, is recovered in the manner already mentioned. Of course any of the three functions involved may require a directory page split to make room for the addition of new data; this possibility does not alter the ordering of operations used.

Linking a node under a second (or subsequent) father is an even simpler operation. The new son pointer is added, followed by the father pointer. If a system failure occurs between the two operations, then the missing father pointer is automatically inserted when the father and son are next accessed.

Moving a file under a new father is similar to the link

operation. The new son pointer is inserted first, followed by an alteration of the father pointer from the old to the new father, followed by a deletion of the old father to son pointer. If the system fails between any of these writes then there will be two son pointers, but only one father pointer. A subsequent tree traversal will insert the missing father pointer corresponding to the son pointer which has no inverse pointer. Thus a network node will be created having both new and old fathers as its two fathers. The user can then delete the extra set of pointers. At the present time, there is no indication to the user that his file has been altered into a network node. It is planned that a later version of NAM will correct this event.

Renaming a file is performed by first removing the old hash entry, followed by altering the name in the node itself, followed finally by adding the new hash entry. Any system failures will result in a missing hash table entry which will be added by a subsequent tree traversal (or validity check).

The last file operation to discuss is that to delete a node from the hierarchy. There are two cases; a node to be deleted may have several fathers, or it may have only one.

In the first case, the node itself remains; only one pair of pointers is removed. The father pointer is of course deleted first, leaving the son pointer in case of a system failure. The node will thus remain undeleted and the user may redo the delete operation when he discovers that the node still exists.

The other case, the regular node, involves somewhat more work. Such a node may possess a set of groups of data records, pointed to by a set of index entries within the node. Each such group of data records must be removed along with the node itself. However, the data groups cannot simply be removed before the node itself is deleted since a system failure would leave the index entries pointing to non-existent data. Deletion of the index pointers first, followed by deletion of the data record groups could lead to a loss of a large amount of space in the case of a system failure since the data record groups could be left undeleted with no index entries present pointing to them. Thus the deletion routine uses the same technique as the directory page split function. An indication of the node being deleted is first written out; then if a system failure occurs the operation can be completed. The removal of the node then starts with the deletion of the data groups. Once these data record groups are all deleted, the hash entry for

the node is removed, and the node itself is deleted by removing first the father pointer, followed by the son pointer, in the same manner as the deletion of a network node. After all of these operations are complete, the indication that a delete operation was in progress is removed.

5.5 RECORD INTEGRITY

Providing integrity during record operations is somewhat simpler than for file operations. This is because most operations require at most one data transfer; for example, adding a record to a group normally requires only one write of the lengthened group, unless the group must be split to create more room. However, several areas of concern still exist. These include operations which manipulate both a data record group and a corresponding index entry, and the functions of splitting and recombining data record groups.

There are three ways in which both an index entry and a group of records are manipulated together. The first of these occurs when the key of the last record in a group is renumbered, added, or deleted. In any of these cases the key stored in the index entry must be altered since it

always indicates the key of the highest record in the group. If the index and group are ever written in such an order that after a system failure the index contains a lower key than the corresponding group, then those records in the group with higher keys may not be located by key, since the index is used to locate records by key. Thus when a higher key is added to the group, the index is written first; when the key of the highest record is lowered (either by renumbering or deletion), the group is written before the index. In this way, the index will always contain a key that is higher than or equal to the highest record and searches by key will still work. Every time an index entry and the corresponding data group are read, the index is checked against the highest key in the group and is altered automatically if incorrect.

The second time at which the index entry and group of records are manipulated together is that of adding a new data group to a file. In this case, a new index entry must be added, in addition to the new data group. The new data group is written first, since otherwise a system failure could leave the new index entry pointing to a non-existent group. If the system fails after writing the data group and before the index entry is written, then the data group is left with no pointers to it; the space it occupies is lost

until the aggregate is copied elsewhere or a validity check is performed. However, no inconsistencies exist; the aggregate still contains correct data.

The final way in which the index entry and group are manipulated together occurs when a data group is deleted. Like the previous case of adding a data group, the two operations of deleting the index entry and deleting the group must be ordered so that an index entry pointing to a non-existent data group does not occur. Thus the index entry is deleted first. If a system failure occurs the space occupied by the data group is lost, but the aggregate is left in a consistent state.

The second major problem area mentioned above was the splitting of data groups into two. Three data transfer operations must occur to split a data group; some of the records must be copied to a new group in the new page, these records must be deleted from the original group, and a new index entry must be added. There is no possible ordering of these three operations that will leave correct data after every possible system failure; with one ordering, data may be lost after a failure, with another, some of the records may appear twice. Thus the solution used is the same as

that used for directory page splits; an indication that a split is to be performed is written out, along with the address of the group being split, the address of the new group being created, and the address of the node whose index entries are affected. After a system failure this information is used to complete the split operation. When the split has been successfully completed, the indication is removed.

Similarly, recombination of data groups after deletions cannot be ordered in a safe way. Thus the same method of writing an indication, performing the operation, and then clearing the indication, is used. In this way, both splitting and recombination of groups are safe operations.

Most data record operations do not involve any potential integrity violations since most operations either do not alter any data (reading records, for example), or only change one item, the group containing the current record. Unless this record is the last one in the group, or a data split or recombination is needed, no integrity problem can occur since only one data transfer is needed to perform the operation. Thus the individual record operations will not be discussed further; the measures listed above are suffi-

cient to handle all possible cases.

5.6 SUMMARY

To summarize the integrity support provided by NAM, there are four unusual conditions that may exist after a system failure. The free space map may be incorrect, there may be a missing father pointer, there may be a missing hash entry, or some space may be occupied but not accessible. The free space map is treated as being only approximate, and is automatically corrected when pages are read into main memory. Missing father pointers and hash table entries are reinserted when the affected node is located (usually during a tree traversal of the hierarchy). Space occupied by inaccessible data records or nodes is left filled until the aggregate is copied, since it does not affect operations upon the aggregate.

In addition to these four conditions, there are four operations which may have only been partially completed, but which write an "in progress" indication before they are started. These are the operations of splitting a directory page or data group, deleting a node, or recombining data groups. The completion of these operations is performed

automatically without the knowledge of the user and leaves the aggregate with consistent data. The integrity of the aggregate is maintained at very little cost in terms of extra data transfers. Occasionally, the ordering of page writes causes extra transfers in order to maintain ordering across two successive operations. In addition, each of the four operations mentioned that use "in progress" indications, written before the operation, and cleared after the operation, require only two extra transfers to perform this action.

Since an "aggregate in use" indication is provided to indicate uses of the aggregate which were terminated abnormally, a means does exist to correct all of the unusual conditions mentioned above automatically. When an aggregate is opened, this indication can be checked and a special routine may be called if the previous use of the aggregate was not terminated normally. This special routine may perform a validity check of the aggregate, correcting all of the four unusual conditions which may have occurred. In this way, the user will be protected completely against system failures; the last operation being performed may not have completed, but the aggregate will be consistent and no unusual events will occur.

CHAPTER 6 - PERFORMANCE

It was stated when the features of the file system were discussed, that there were two major concerns about the practicality of such a file system. The first of these was a concern about the integrity of the user's data, which has been discussed in chapter 5. The other concern remaining to discuss is that of the performance of the file system. If the file system were too costly to operate, then it would have very limited usefulness, despite the power and flexibility of its features. This chapter, then, discusses the cost of using the NAM file system, as it is used by the MANTES editor.

The cost of providing the direct updating and hierarchical support occurs mainly in two areas. The first of these is the extra space required to store the data. In a regular file system, supporting sequential files, the only space required is that for the actual records. In NAM, due to fragmentation, and to allow for expansion, extra space is used. The second major cost is that of the input/output operations required for direct updating and random access to data. Other costs will be discussed in a separate section.

The main emphasis will be a comparison of the costs incurred by NAM, compared to those incurred by a standard copying editor. Empirically gathered statistics, collected by NAM in a special area of the first page of each aggregate, will be used, rather than attempting to study analytically the file system operations, since most of the costs are very dependent upon the usage of the facilities by the editor. The statistics used were gathered from 30 aggregates which were in use at one time during one typical day. It should be noted that these statistics give the cost of providing those facilities used by the MANTES editor; other users of the file system, if any, might utilize the features of the file system in an entirely different manner.

6.1 SPACE OVERHEAD

The space used by NAM can be divided into four areas. These include the space occupied by fixed tables, such as the free space map, the space occupied by directory nodes, the space required for data records, and unoccupied space used for expansion purposes.

Starting with the space used for fixed tables, part of

the first page of the aggregate is used to contain three fixed groups. These three groups are the free space map, the hash table, and an area used tc contain statistics on the aggregate usage. This statistics group occupies 236 bytes while the free space map has as many bytes as there are pages allocated to the aggregate. The hash table occupies 309 bytes for the fixed table portion, plus three bytes per node in the aggregate for the variable portion Thus the total fixed table size can be given by;

```
236 + 309 + npage + 3*nnodes        bytes

"npage" is the number of pages allocated
"nnodes" is the number of nodes
```

One page is the maximum space that these tables may occupy since NAM does not allow them to overflow to another page.

Considering next the storage of the nodes of the hierarchy, each node consists of a flag byte, a list of sons, a list of fathers, a name, a set of passwords, a set of attributes, and an index to the data records. Each one of these fields consists of a one-byte length plus the field itself, giving a total size for a node of;

```
7 + 2*nsons + 2*nfath + lname + lpass + lattr + index

"nsons" is the number of sons (2 bytes each)
"nfath" is the number of fathers (2 bytes each)
"lname" is the length of the node name
"lpass" is the length of the password information
"lattr" is the length of any attributes
"index" is the index to the records
```

The value of "index" can be computed by;

```
nindex * (3 + lavgi)

"nindex" is the number of data record groups
"lavgi" is the average key length
```

Since it is difficult to estimate the values for fields such as "nsons", "lname", etc, the empirical figures may be examined to see how long the nodes are on average. Looking at TABLE 6.1, the average length of each of the 2,586 nodes is 38.3 bytes. (This length includes the group map entry required in the page to contain the offset and length of the group containing the node).

NUMBER OF NODES	2,586	
SPACE OCCUPIED BY NODES	99,014	BYTES
AVERAGE NODE SIZE	38.3	BYTES
SPACE OCCUPIED BY DIR PAGES	199,334	BYTES
UNUSED SPACE IN DIR PAGES	100,320	BYTES
% OF SPACE USED IN DIR PAGES	49.7	%
NUMBER OF DATA RECORDS	211,812	
SPACE OCCUPIED BY RECORDS	8,963,129	BYTES
AVERAGE RECORD SIZE	42.3	BYTES
SPACE OCCUPIED BY DATA PAGES	10,816,320	BYTES
UNUSED SPACE IN DATA PAGES	1,853,191	BYTES
% OF SPACE USED IN DATA PAGES	82.9	%
SPACE OCCUPIED BY FIXED TABLES	19,546	BYTES
TOTAL SIZE OF AGGREGATES	11,035,200	BYTES
TOTAL UNUSED SPACE	1,953,511	BYTES
% OF SPACE UNUSED	17.7	%
SPACE OCCUPIED BY EMPTY PAGES	4,970,432	BYTES
TOTAL SPACE OCCUPIED	16,005,632	BYTES
% OF SPACE IN EMPTY PAGES	31.1	%

- TABLE 6.1 SPACE USAGE -

However, these nodes are placed into their own directory pages, each with unused fragmentation and expansion space. The row entitled "SPACE OCCUPIED BY DIR PAGES" gives the total number of bytes of usable space in the directory pages. This space includes any space left over from the fixed tables in the first page of the aggregates, since this space is usable for storing nodes of the hierarchy. The

next two rows in the table indicate that only 49.7% of this directory page space is actually used for data; the rest is left available for expansion. This somewhat low utilization is due to two factors. The first is that each directory page can hold approximately 100 nodes; several of the aggregates only possess a small number of nodes but require a full directory page to hold them. In addition, NAM does not recombine directory pages after nodes are deleted; it is possible to delete a set of nodes and leave a directory page with only one node in it. In any case, the important statistic is that the 2,586 nodes occupy 199,334 bytes of space within the aggregates.

The third set of data occupying space within an aggregate is that consisting of the data records themselves. Turning again to TABLE 6.1, it can be seen that there were 211,812 records occupying 8,963,129 bytes of storage (including the record keys and control information), for an average of 42.3 bytes of storage per record. These records were contained in pages totalling 10,816,320 bytes of space, for a utilization of 83%. The remaining 17% of the space contains the unused space left in each page due to fragmentation and for expansion purposes.

Totalling these figures gives an overall figure of 17.7% unused space within the aggregates. In addition, it can be noted that the 199,344 bytes occupied by the nodes of the hierarchy comprise less than 2% of the total space occupied. Summarizing these figures it can be said that the cost in space of supporting the hierarchy is very low; even though the directory pages are less than 50% utilized, the hierarchy occupies less than 2% of the total space used.

The support for direct updating is not quite as cheap, however. A copying editor could presumably store the records in the same 42.3 bytes each (approximately the same amount of control information would be needed), but since a copying editor uses sequential files with no unused space, the 17% of unused space could be eliminated. This 17%, then, is the extra space cost required by the direct updating support provided by the NAM file system.

One problem that NAM shares with other file systems is with the allocation of disk space to aggregates. The particular operating system used, IBM's OS/VS system, requires that a user make an initial guess as to the size of his file. The operating system then attempts to allocate the specified amount, called the primary allocation, of

space contiguously on disk, to minimize the disk seek times when accessing several parts of the file in sequence. In addition, the user must specify a size by which the file will be extended if insufficient space exists in the primary allocation. (However, only a small number of such extensions are allowed). Unfortunately, much space is wasted by overestimation of the initial space requirements. The amount of such space unused but allocated in the sampled aggregates is indicated in the column "SPACE OCCUPIED BY EMPTY PAGES", in TABLE 6.1. Over 31% of the total space occupied by the aggregates was due to this overestimation. In order to alleviate this waste of space, NAM returns an indication at open time to the user if his aggregate has many free pages, more than 25% of the total space in the aggregate, and has been in existence for a reasonable time. (Obviously messages should not be given for new and therefore empty aggregates). The user may then copy the data from the aggregate to a smaller one, or he may use a special command (added since these statistics were gathered), which returns empty pages to the operating system leaving only a specified percentage of free space. In fairness, it should be mentioned that this problem exists for all users of operating system software, and NAM was forced to perform allocation of disk space to aggregates in this manner. Other operating systems, such as UNIX, have the ability to

allocate single blocks to files as they are needed, albeit with longer seek times to access blocks of the files.

6.2 INPUT/OUTPUT COST

The second major cost in a file system such as NAM is the number of data transfers required. As mentioned previously, these accesses are very expensive; they are very slow when compared to processor speed and a limited number of such accesses may be performed per unit of time. For example, an IBM 3350 disk drive takes an average of 30 milliseconds to access a data block. Thus only about 30 disk accesses may be performed per second, on such a device. In addition, there are many other users of the disk units; user programs, other data files, and system virtual memory pages are all stored on disk as well. Thus it is important to minimize the number of data transfers needed, as too large a number of such transfers could make the file system unusable, particularly in a time-sharing environment where fast response to requests is necessary.

Some general comments about the statistics presented are required. TABLE 6.2 presents statistics on the usage of the thirty sampled aggregates. The row "AGGREGATE DAYS" indi-

cates the cumulative life of the aggregates measured in number of days of existence, while the row entitled "NUMBER OF OPEN ACCESSES" indicates the number of times aggregate opens were issued to access the aggregates. The bottom two sections of the table give the number of times each of the file and record operations were issued. Unfortunately, the number of data transfers performed for each specific file system operation cannot be measured; NAM does not write changes to data back to disk until it is required to, either because the main memory is needed for other data, or to order page writes to maintain integrity, or because the user issues a purge request to force all changes to be written out. Thus a given write transfer may include changes from several operations; no relation can be established between a specific write and a specific operation. In addition, it should be mentioned that these statistics include the writes performed owing to the MANIES editor performing purging commands. It issues a purge operation to cause the aggregate to be kept up to date after each editing command, or after every 10 new records added to the aggregate. Thus the given statistics include data transfers issued solely for the purpose of keeping aggregates up to date to protect against loss of data in the event of system failures.

AGGREGATE DAYS	4,174
NUMBER OF OPEN ACCESSES	8,174
NUMBER OF READ TRANSFERS	223,205
NUMBER OF WRITE TRANSFERS	269,814
TOTAL INPUT/OUTPUT TRANSFERS	493,019
NUMBER OF LOCATE OPERATIONS	509,832
CREATE FILE	5,442
LINK FILE	62
DELETE FILE	3,037
RENAME FILE	721
MOVE FILE	177
ADD/DEL ATTRS & PASSWDS	1,500
TOTAL FILE OPERATIONS	520,771
NUMBER DIRECTORY PAGE SPLITS	31
NUMBER OF POINT OPERATIONS	662,895
INSERT RECORD	632,440
DELETE RECORD	403,470
READ RECORD	9,413,047
REWRITE RECORD	135,865
RENUMBER RECORD	1,304,443
TOTAL RECORD OPERATIONS	12,552,160
NUMBER OF DATA GROUP SPLITS	1,016
NUMBER OF GROUP RECOMBINATIONS	815

- TABLE 6.2 NUMBER OF OPERATIONS -

Examining the statistics, it is evident that the file operations comprise only a small proportion of the total, approximately 4%, and of these 98% are locate operations which do not make any changes to the aggregate. Thus the most expensive operations in terms of input/output cost,

those which manipulate the hierarchy, do not comprise a significant proportion of the overall total. Thus it is not important that each such file manipulation operation may take several data transfers to complete.

The total number of operations issued was 13,072,931 using 493,019 data transfers, giving an average of 26.5 operations per data transfer. This fairly high average is due to the large number of times an editing operation reads the entire file, or a large part it. Over 70% of the operations were read record operations. Turning to TABLE 6.3, it can be seen that the records were blocked at an average of 44 records per group. Thus an average of 44 read operations per data transfer would occur during the reading of files.

NUMBER OF RECORDS	211,812
NUMBER OF DATA GROUPS	4,810
AVG NUMBER OF RECORDS/GROUP	44
NUMBER OF DATA PAGES	2,965
AVG NUMBER OF RECORDS/PAGE	71

- TABLE 6.3 BLOCKING FACTORS -

To compare the cost of editing using NAM to that of a copying editor, the following comparison may be made. Using the same average record length, 42.3 bytes, and the same pagesize, 3648 bytes, as NAM, a sequential file as used by a copying editor can fit 86 records in each block. Thus the operations to copy the file to and from the internal work file operate at a speed of 86 such reads or writes per data transfer. Looking back at TABLE 6.1, it can be seen that there were 8,174 aggregate opens. Thus, dividing this into the total number of transfers, 493,019, there were 60 transfers per open aggregate. If each such aggregate open corresponded to an edit of a single file using a copying editor, then, ignoring entirely the cost of handling the internal work file, to achieve the same or better input/output cost, the copying editor would have to use 60 or fewer transfers per file edited. With a blocking factor of 86 records per block, 60 transfers allow the accessing of slightly more than 5,000 records, but since each record has to be both read into the temporary file and then written back, the 60 transfers would only allow for a file of 2,500 records. Thus, while NAM made accesses to 8,174 aggregates, each containing an average of 7,000 records, in only 60 data transfers per access, a copying editor with the same size records and blocks can only just manage to access 2,500

records, without performing any editing at all, and ignoring any transfers necessary to manage the internal work file, in the same 60 transfers.

Thus the input/output cost to use NAM for source file editing can be summarized as follows. On the 30 aggregates sampled, NAM stored an average of 44 records per group, thus giving an average of 44 operations per data transfer, for sequential operations. This is twice as slow as a standard sequential file using the same record length and pagesize. Including all operations, allowance for the direct accessing and updating of records and files reduced the figure to 26.5 operations per data transfer. However, for editing, the NAM system is much cheaper than the use of sequential files with a copying editor, since the number of transfers required to copy the files for a copying editor is greater than the transfers required for NAM to perform the editing, using comparably sized files. In addition, the cost of editing using the NAM file system is not a function of the number of files or records in an aggregate. Editing a file will take approximately the same number of data transfers whether there are a large number of records and files or not, since the time to access a record from a file is approximately constant. Accessing a record requires only a scan of the index entries, followed by a direct read of the group

containing the record. However, in a copying editor, the cost is very dependent upon the file size. If the total number of records in a file is doubled, then the number of transfers to copy the file in to the temporary work file and back is doubled. Thus the relative advantage of NAM increases as the number of records the user wishes to handle increases.

6.3 OTHER COSTS

Another cost, but one of less importance, is the central processor time required to perform operations. Certainly a file system such as NAM would be expected to take more processor time per operation than a sequential file system, due to the overhead of managing the index pointers, the groups, the unused space, the password scheme, and so on. Since the majority of operations requested of NAM are record read operations, these have been optimized in terms of processor time in NAM. Examining TABLE 6.4, it can be seen that 166,848 read operations were performed in 24.2 seconds of processor time, for an average of about 7,000 reads per second , while 84,324 mixed updating operations were executed at an average time of approximately 4,000 operations per second. To give these figures some meaning, the

most basic sequential access method was able to read records at a rate of roughly 60,000 records per processor second. Thus NAM is almost 10 times slower at reading records than a simple sequential file system, and is almost twice as slow again performing updating operations. However, this extra processor time is not a major factor. Looking back at TABLE 6.2, it can be seen that the 13,000,000 file system operations were performed over 4,174 aggregate days and 8,174 accesses. Thus approximately 3,100 operations were issued per aggregate per day, or 1,600 per aggregate access. Thus less than .5 seconds of processor time per day were used performing file system operations for each aggregate. At the present time, approximately 500 MANTES editing sessions are performed per day. If each of these accesses one aggregate at the same rate of 1,600 operations per access, then the total processor time per day would be under two minutes. Even a factor of ten times this rate would still only represent less than 1% of the total processor time. However, there is one occasion where the high processor cost is noticeable. Occasionally, a user wishes to perform a large number of operations at once, for example, to copy an entire aggregate to another place. In this case, a sufficient number of accesses would occur for the processor time to be noticeable. Copying 35,000 records (which is a large aggregate), occupies between 10 and 20

seconds of processor time; the user who wishes to perform such an operation may have to wait for some time before the operation is completed.

NUMBER OF READ OPERATIONS	166,848	
PROCESSOR TIME TAKEN	24.2	SECONDS
AVERAGE RATE	6,894	/SEC
NUMBER OF MIXED OPERATIONS	84,324	
PROCESSOR TIME TAKEN	21.2	SECONDS
AVERAGE RATE	3,506	/SEC

- TABLE 6.4 PROCESSOR TIME -

Another cost associated with the file system is the amount of main memory required to run the file system. NAM itself occupies approximately 110,000 bytes of main memory for the programs themselves, but only approximately 65,000 bytes are used to provide the functions mentioned in this thesis. (NAM also provides management for standard operating system files, and batch job output files). This figure of 65,000 bytes is comparable to the size taken by the manufacturer's file systems; some of them are smaller and some are larger. Considering dynamic memory space required to support each open aggregate, both the manufacturer's

supplied file systems and NAM use approximately five times the pagesize for memory buffers. Thus NAM costs approximately the same amount for main memory occupancy as the standard file systems.

6.4 SUMMARY

The performance of NAM may be summarized in the following manner. NAM uses more secondary storage space to store the user's records than a sequential file could. In the thirty sampled aggregates, approximately 17% extra space was used to store the extra control information and to provide expansion space over the space necessary to simply store the records. As far as input/output cost, for simply processing records sequentially, NAM is approximately twice as expensive as a sequential file system. However, for editing purposes NAM is cheaper to use than sequentially ordered files; the larger the amount of data to be handled, the cheaper NAM is, compared to a copying editor. In processor time, NAM is considerably more expensive than a simple sequential file system, but still overall uses a fairly small amount of processor time. In memory space, NAM uses approximately the same amount of main memory for data as standard file systems.

One other comment may be stated regarding the performance of NAM. The manufacturer's editing system, which the combination of MANTES and NAM replaces, uses uncompressed 80-byte records for source files. Thus NAM, even with its 17% unused space, uses less space than the manufacturer's editing system. (It uses about 63% of the space used by the supplied editor). In addition, this higher record size increases the difference in data transfers encountered between NAM and a copying editor, since the number of read or write operations per data transfer is only approximately 46 as compared with the 86 possible with compressed records, and used for the previous comparisons. Thus NAM is cheaper in all categories except processor time when compared to the supplied editing system it replaces.

CHAPTER 7 - CONCLUSIONS

The original goal of this thesis was to produce a file system which would provide significantly enhanced functions for source file editing systems. A set of such enhancements, in a combination not existing in currently available editing systems, has been described in this thesis. These facilities include direct updating, structuring of files, network structures, versions of files, password protection using the hierarchy, inherited attributes, and the ability to treat entire substructures as a single unit. In addition, file lookup by unqualified name is permitted.

A file system including these features has been constructed and an editor using all of the provided facilities has also been implemented (by other people) to provide enhanced editing functions based upon the file facilities. Both the file system and the editor are in standard use by a user population of approximately 600 people. The comments received from the users have been in favour of the system; so much so, that many people used the system before it was completed, and during a time when program errors sometimes inadvertently destroyed some of the users' data. (It should

be mentioned however, that the editing system that was replaced was not particularly powerful or flexible). From the users' reaction it can safely be said that the facilities provided are of real use to the general user population and improve user efficiency in editing. In this respect, the design has been an unqualified success, although much of the credit for the users' acceptance of the system should go to the editor design and not the file system.

As far as the reliability and integrity of the system are concerned, a high level of reliability has been achieved. NAM has been in regular use for one and a half years and as mentioned, at the time of writing handles data for approximately 600 users. In that time, several dozen aggregates have been damaged by the file system in such a manner that recovery from a backup copy (user files are backed up every night) was necessary. All of these damaged aggregates were the result of coding errors in the actual implementation, usually with respect to recently implemented features; none was the result of incorrect handling of system failures. The features provided to maintain integrity across system failures work well, and there is general unconcern among the user population about the safety of their data. With the previous copying editor, there was much concern over the possibility of system failures; often after such a failure

large amounts of data and large numbers of editing commands had to be reentered. Now the user's only concern is with the actual lost time resulting from the system not being available during the time of a failure. As an example of the reliability of NAM, information about users required for logging into the MANTES editing system is kept in an aggregate. Over the past year, approximately 70 million file system operations have been issued against this aggregate, for an average of approximately 20,000 operations per day. In that time, even with such heavy usage, and with the number of system failures that have occurred, no data within the aggregate has been damaged. In that time, one missing pointer was inserted automatically; this was the only effect of system failures over the one year period.

The cost of the file system has been well within acceptable limits. While the processor time used is higher than for the previous editing system, NAM provides direct updating and the hierarchy support in less space and with fewer input/output transfers than the previous system, although the space required would have been higher had the previous system used compressed records. The extra space required over that needed by a copying editor making the best use of space is still reasonable, especially considering that NAM reduces the fragmentation of disk space caused

by the very large number of small files that exist when an editor supports only a single file at a time. One additional saving not mentioned previously is that owing to openinig fewer files. NAM only requires one open of an operating system file per aggregate; a copying editor must open each individual file as it is edited. Since the system operation of locating, allocating, and opening a file is quite expensive, this gain is fairly significant.

All in all, a greater functional capability has been provided without incurring any high costs. However improvements could be made. The file system, as it stands, does not yet implement quite all the features mentioned in this thesis. In particular, versions of files have not been implemented, although the MANTES editor has supplied the facility by adding the two mentioned version numbers as extra parts of the record key. The ability to have a file possessing both data records and subfiles has not yet been implemented; each node in the hierarchy may either have a set of sons, or it may contain a set of data records, but not both. Support for this would probably require several months of implementation effort. Heavy usage of this facility would cause the data transfer cost to increase by a small amount; there would be more directory operations, and more directory pages to handle. In addition, if the

subfiles of a file were entered in very random order, there would be a small increase in input/output cost due to a reduced blocking factor caused by spreading subfiles out over different pages. At the current time, records are kept together in fairly large groups; subdividing the files would spread the data over a larger number of pages.

Another facility neither discussed nor implemented is the ability to extend the hierarchy in the third dimension. It might be advantageous to associate with a source file, its object file, and perhaps documentation, and so on. Having a file system implement relationships between files in the third dimension could be very useful to such programs as compilers, text formatters, and so on. The implementation of a third dimension is not extremely difficult - another set of pointers is needed to describe the relations in the third dimension. However, this facility was not considered for implementation because other programs, such as compilers, do not exist to use the facility.

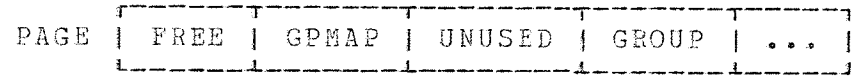
Some restrictions exist in NAM which could limit its use beyond source file editing. Restrictions in the size of the hash table, number of sons of a node, number of records per file, and size of records, as implemented, are not signifi-

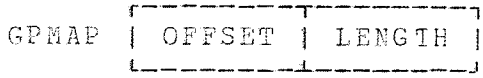
cant for source file editing, but might prevent the usage of NAM in other areas; in particular, it could not become a base file system for data processing applications of any large size (and as mentioned was never intended to). Future efforts are needed to design a common file system which can handle both the editing needs as well as other applications without sacrificing too much efficiency for any particular application.

APPENDIX A - DATA STRUCTURES

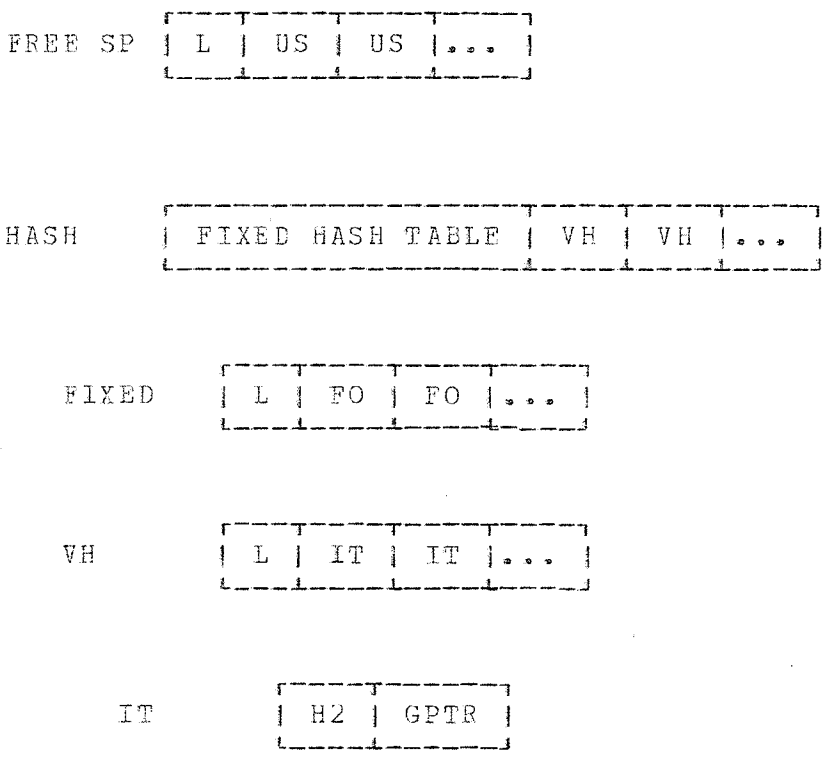
This appendix describes the data structures used in storing an aggregate, using a sequence of successively refining descriptions. The element names used inside the following block diagrams are described in a list following the diagrams.

An aggregate consists of a set of pages, each containing a group map and a set of groups (as well as unused space), as depicted below.

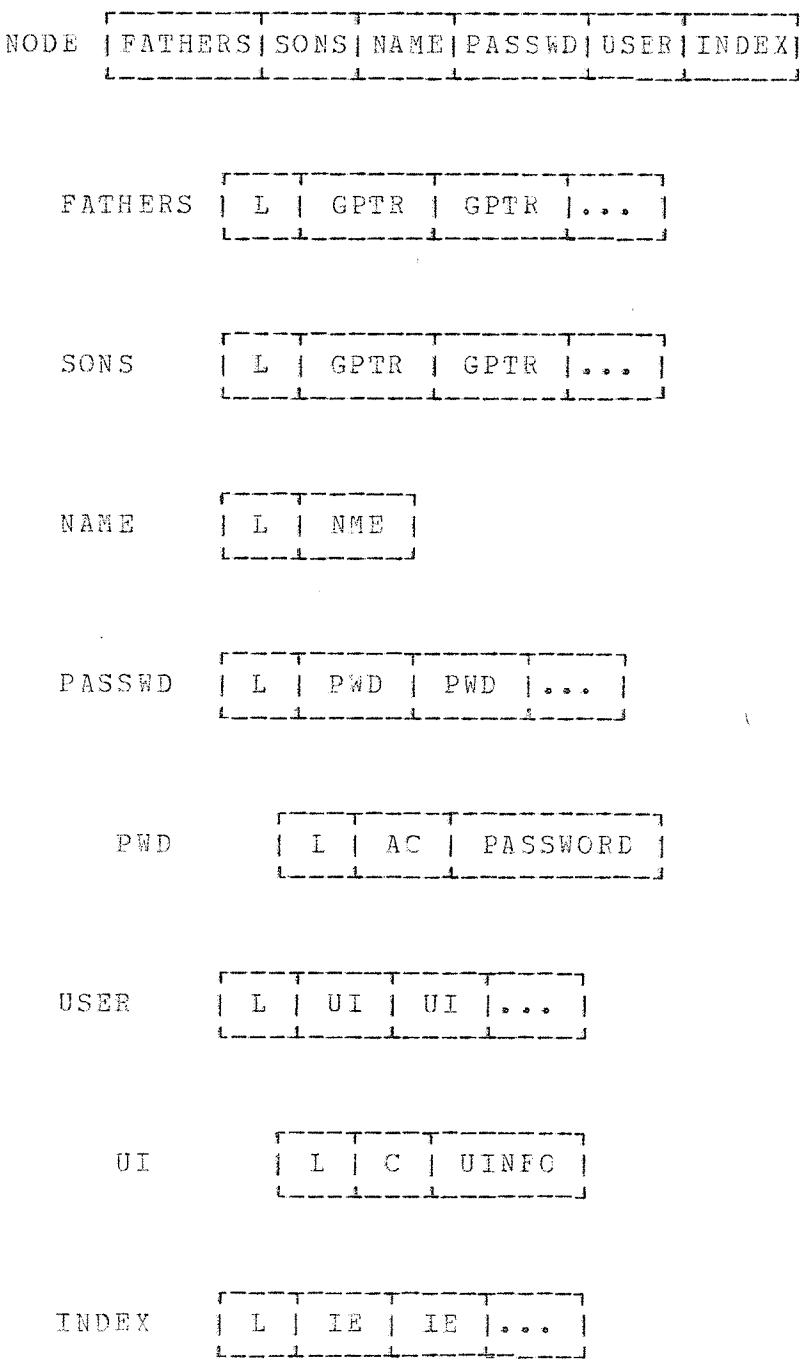




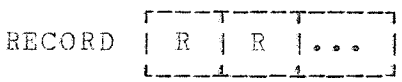
A group may be one of four types, the free space map group (occurs only once in the first page of the aggregate), the hash table (also occurs only once), a directory node group (one for each node in the hierarchy), or a data record group (one for each group of data records). The first two of these, the free space map, and the hash table, are described here.

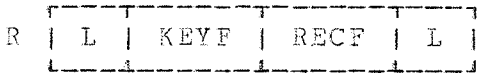


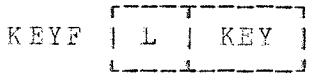
A directory node has the form:

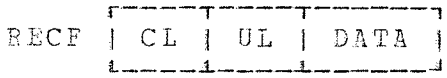


A group containing data records has the format:









The following are the elementary items used in the above diagrams:

AC - password access code
C - user information identification code
CL - compressed length of record
DATA - compressed data record
FO - offset of corresponding VH hash list
GPTR - group address of pointed to group
H2 - second hash value
HIGHKEY - highest key in data group
KEY - key value of record
L - length of entry
LENGTH - length of corresponding group
NME - name of node
OFFSET - offset of corresponding group
PASSWORD - encoded password value
UINFO - user information field
UL - uncompressed length of record
UNUSED - unused space within group

APPENDIX B - SAMPLE ALGORITHMS

This appendix gives some of the more complex algorithms used inside NAM to give the reader a flavour of the type of processing performed. It should be noted that the write command used in these descriptions need not necessarily write a page to disk - it merely marks the page for write back to disk and ensures that when pages are written, they will be written in the order in which the write statements were issued.

In the following algorithms, an indication is written out before the operation is started and is cleared after the operation completes. If the system were to fail during the operation, then when operation is resumed, the indication is tested and control is returned to label "reentry" in each of these routines. All subsequent actions must be able to cope with the possibility that their function was completed normally before the system failure, since no information is known as to how far the operation completed before the system failure.

DELETING A NODE

The delete operation is used to delete a node from the hierarchy. Deletion of either the root node, or a directory node is not allowed. If the node is a network node, (i.e. it possesses more than one father), then the node itself remains; only the link to the given father is removed. For a data node, all of the data groups must be removed. In this case, a delete in progress indication is used, since a system failure could leave much lost space if we were to remove the index, but not complete the removal of the data groups before the system failed.

```
   if node is root or a directory node then
      return error indication;
   fi
   if node is a network node then
      delete son to father pointer;
      write altered node to disk;
      delete father to son pointer from father;
      write father node;
   elif node has no data records then
      delete son to father pointer;
      write altered node;
      delete node itself;
      write page with deleted node;
   else
      write first page with delete indication;
reentry:
      delete index from node;
      for each data group do
         delete data group from page;
         write page containing deleted group;
      end
      write first page with indication removed;
   fi
```

SPLIT_DIRECTORY_PAGE

To split a directory page, the routine is given the number of the page to be split. It divides its contents evenly over the given page, and a totally empty page. Then all nodes which point to the nodes which were moved must have their pointers adjusted to reflect the new addresses of the moved nodes. Since there is no ordering of page writes which would always leave the aggregate in a consistent state (as long as network nodes can exist), a split in progress indication is used to handle the possibility of a system failure during the operation.

```
   origpage := number of page to be split;
   newpage := number of an empty page;
   middlenode := node closest to middle of page;
   for each node "a" higher than middlenode do
      copy "a" to page newpage;
      alter hash table (in-core copy only);
   end
   create group map for copied nodes;
   write page newpage;
   write first page with split indication set;
reentry:
   for each copied node "a" do
      for each father of "a" do
         alter corr. son pointer to new address;
      end
      for each son of "a" do
         alter corr. father pointer to new address;
      end
      adjust in-core tables;
   end
   write all changed pages;
   for each copied node "a" do
      remove "a" from origpage;
      adjust group map in origpage;
   end
   write origpage to disk;
   clear split in progress indication;
   write first page with split indication cleared;
```

SPLIT_DATA_GROUP

Splitting a data group is done in one of two ways, in order to try to maximize the size of the data groups. If the group to be split is fairly small (less than 1/2 the size of a page), then rather than splitting it into smaller pieces, an attempt is made to find a page with at least 1/2 again as much space as the group possesses. If such space exists, then the group is not split, but is merely moved to the new page. Otherwise, the page with the most free space is obtained, and the group is split into two portions. Note that it is possible to order the write operations to maintain integrity; NAM uses a split in progress indication just to prevent any possible loss of space.

```
   if split group < 1/2 pagesize then
      if page with free sp >= 3/2 split group size
         is found then
         copy split group to newpage;
         write newpage to disk;
         alter index to point to new location;
         write altered index;
         remove old copy of group;
         write old page with group deleted;
         return
      fi
   fi
   middlerecord := record closest to middle of page;
   newpage := page with greatest amount of free sp;
   for each record "a" higher than middlerecord do
      copy "a" to newpage;
   end
   write newpage to disk;
   write first page with split in progress indication;
reentry:
   locate index for file;
   add new index entry to index;
   write altered index;
   remove copied records;
   write oldpage;
   write first page with split indication cleared;
```

DATA GROUP RECOMBINATION

The data group recombination routine is entered after a record is deleted. Its function is to try to recombine adjacent (in sequence) data groups which are very small into large groups. The group from which a record has just been deleted is checked with the preceding (in sequence), and following groups to see if there is sufficient space to recombine it with either of these groups. Again, a recombination in progress indication is used, to guarantee integrity across a system failure.

```
   delsize := size of group being recombined;
   precsize := size of preceding group;
   follsize := size of following group;
   if delsize + precsize fits inside precpage then
      combpage := precpage;
   elif delsize + follsize fits inside follpage then
      combpage := follpage;
   fi
   if combpage ¬= 0 then
      write page one with recombination indication;
reentry:
      copy delgroup records into combpage;
      write combpage with new records;
      locate index entry pointing to delgroup;
      delete index entry;
      write index to disk;
      delete delgroup;
      write delpage with delgroup deleted;
      write page one with recombination ind. removed;
   fi
```

BIBLIOGRAPHY

Astrahan M. et al: "System R: Relational Approach to Database Management", ACM Transactions on Database Systems 1,2 (June 1976), 97-137.

Bachman C. "The Evolution of Storage Structures" CACM 15,7 (July 1972), 628-636

Bayer R. & McCreight. "Organization and Maintenance of Large Ordered Indexes", Acta Informatica 1,3 (1972), 173-189

Benjamin A. "An Extensible Editor for a Small Machine with Disk Storage" CACM 15,8 (Aug 1972), 742-747

Bensoussan A., Clinger C., Daley R. "The MULTICS Virtual Memory: Concepts and Design" CACM 15,5 (May 1972), 308-318

Betourne C. et al: "Process Management & Resource Sharing in the Multiaccess System ESOPE" CACM 13,12 (Dec 1970), 727-733

Chapin N. "Common File Organization Techniques Compared" Proceedings AFIPS, FJCC, vol. 35, 1969, 413-422

Considine J and Weis A "Establishment and Maintenance of a Storage Hierarchy for an On-Line Data Base under TSS/360" Proceedings AFIPS, FJCC, vol. 35, 1969,433-440

Corbato F, et al: "An Experimental Time-Sharing System" Proceedings AFIPS, SJCC, vol. 21, 1962, 335-344

Daley R. and Neumann P. "A General-Purpose File System for

Secondary Storage" Proceedings AFIPS, FJCC, vol 27, 1965, 213-229

DEC 1 "Introduction to RSX-11M" Digital Equipment Corporation, order DEC-11-OMIEA-B-D

DEC 2 "PDP11 Peripherals Handbook" Digital Equipment Corporation

DEC 3 "PDP11 Processor Handbook" Digital Equipment Corporation

Deutsch P. and Lampson B. "An Online Editor" CACM 10,12 (Dec 1967) 793-799

Faiman R. and Borgelt J. "WYLBUR: An Interactive Text Editing and Remote Job Entry System" CACM 16,5 (May 1973), 314-322

Feieritag R. and Organick E. "The Multics Input/Output System" Third ACM Symposium on Operating System Principles, Stanford University, Oct 18-20, 1971 17-23

Ferch H. "NAM Calling Sequences" Internal Publication, Dept. of Computer Science, University of Manitoba, 1978

Foster C. "Information Storage and Retrieval using AVL Trees" Proceedings ACM 20th National Conference, 1965, 192-205

Hughes J. "PL/I Programming" John Wiley & Sons, Inc., 1973

IBM 1. "Introduction to OS/VS2 Release 2" IBM Corp., Form number GC28-0661

IBM 2. "OS/VS2 TSO Terminal User's Guide" IBM Corp., Form number GC28-0645

IBM 3. "OS/VS2 JCL" IBM Corp., Form number GC28-0692

IBM 4. "OS/VS VSAM Programmer's Guide" IBM Corp., Form number GC26-3838

IBM 5. "REFERENCE MANUAL FOR IBM 3350 DIRECT ACCESS STORAGE" IBM Corp., Form number GA26-1638

IBM 6. "IBM System/370 Principles of Operation" IBM Corp., Form number GA22-7000

Katzan H. "Storage Hierarchy Systems" Proceedings AFIPS, SJCC, vol 38, 325-336

Knuth D. "The Art of Computer Programming, Vol. 3: Sorting & Searching" Addison-Wesley, Reading, Mass. 1973

Lefkovitz D. "File Structures for Online Systems" Spartan Books, London, 1969

Madnick S., and Alsop J "A Modular Approach to File System Design" Proceedings AFIPS, SJCC, 1969, vol 34, 1-14

Madnick S. and Donovan J. "Operating Systems" McGraw-Hill Inc., 1974

Martin J. "Computer Data-Base Organization" Prentice-Hall, 1975

Maruyama K. and Smith S. "Analysis of Design Alternatives for Virtual Memory Indexes" CACM 20,4 (April 1977), 245-253

Morris R. "Scatter Storage Techniques" CACM 12,1 (May 1968), 38-44

Organick E. "The Multics System: An Examination of its Structure", The M.I.T. Press, Cambridge, Mass. 1972

Powell M. "The DEMOS File System" Proceedings Sixth Symposium on Operating System Principles, Purdue University, Nov 16-18 1977

Ritchie D. and Thompson K. "The UNIX Time-Sharing System" CACM 17,7 (July 1974), 365-375

Saltzer J. "Protection and the Control of Information Sharing in MULTICS" CACM 17,7 (July 1974), 388-402

Schneider B. and Watts R. "SITAR: An Interactive Text Processing System for Small Computers" CACM 20,7, (July 1977), 495-499

Senko M. et al: "Data Structures and Accessing in Data-Base Systems" IBM Systems Journal, vol 1, 1973, 30-93

Stern J. "Backup and Recovery of On-line Information in a Computer Utility" M.S. Thesis, M.I.T. 1974

van Dam A. and Rice D. "On-Line Text Editing: a Survey" Computing Surveys 3,3 (Sept 1971), 93-114

WITS "WITS-II Reference Manual", Computer Centre, University of Waterloo, Dec. 1972

Wulf et al: "HYDRA: The Kernel of a Multiprocessor Operating System" CACM 17,6 (June 1974), 337-344

Yourdon E. "Design of Online Computing Systems" Prentice-Hall, 1972

Zarnke C.R., Perch H., Neufeld G. "MANTES Reference Manual", Internal publication, Dept. of Computer Science, and Computer Centre, University of Manitoba, 1978