AD-774 141

BACKUP AND RECOVERY OF ON-LINE INFORMATION IN A COMPUTER UTILITY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PREPARED FOR
ADVANCED RESEARCH PROJECTS AGENCY
OFFICE OF NAVAL RESEARCH

JANUARY 1974

BIBLIOGRAPHIC DATA SHEET	1. Report No. MAC TR-116	2.	3. Recipient's Accession No.

4. Title and Subtitle

Backup and Recovery of On-Line Information in a Computer Utility

5. Report Date : Issued January 1974

6.

7. Author(s)

Jerry A. Stern

8. Performing Organization Rept. No. MAC-TR-116

9. Performing Organization Name and Address

PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY :
545 Technology Square, Cambridge, Massachusetts 02139

10. Project/Task/Work Unit No.

11. Contract/Grant No.
N00014-70-A-0362-0006

12. Sponsoring Organization Name and Address

Office of Naval Research
Department of the Navy
Information Systems Program
Arlington, Va 22217

13. Type of Report & Period Covered : Interim Scientific Report

14.

15. Supplementary Notes

S.M. + EE Degree, Thesis, M.I.T., Department of Electrical Engineering

16. Abstracts

This thesis describes a design for an automatic backup mechanism to be incorporated in a computer utility for the protection of on-line information against accidental or malicious destruction. This protection is achieved by preserving on magnetic tape recent copies of all items of information known to the on-line file system. In the event of a system failure, file system damage is automatically assessed and missing information is recovered from backup storage. For isolated mishaps, users may directly request the retrieval of selected items of information. The design of the backup mechanism presented in this thesis is based upon an existing backup mechanism contained in the Multics system. As compared to the present Multics backup system, the new design lessens overhead, drastically reduces recovery time from system failures, eliminates the need to interrupt system operation for backup purposes, and scales up significantly better with on-line storage growth.

17. Key Words and Document Analysis. 17a. Descriptors

File Backup
Crash Recovery
On-Line Storage
File System
Computer Utility
Multics
Reliability

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

Unlimited Distribution
Write Project MAC Publications

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page) UNCLASSIFIED

21. No. of Pages ~~124~~ 125

22. Price $4.50

THIS FORM MAY BE REPRODUCED

MAC TR-116

BACKUP AND RECOVERY OF ON-LINE INFORMATION IN A COMPUTER UTILITY

Jerry A. Stern

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095, and was monitored by ONR under Contract No. N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

ACKNOWLEDGMENTS

I wish to thank my thesis supervisor, Professor F. J. Corbató, for his encouragement and guidance in this research, and for his helpful criticisms and suggestions concerning the thesis presentation. Thanks are also due Professors J. H. Saltzer and M. D. Schroeder for their comments and advice. Finally, I would like to thank all of my fellow graduate students in the computer systems research group at Project MAC who offered useful suggestions, as well as interesting distractions, throughout the course of this work.

BACKUP AND RECOVERY OF ON-LINE INFORMATION IN A COMPUTER UTILITY

by

Jerry A. Stern

ABSTRACT

This thesis describes a design for an automatic backup mechanism to be incorporated in a computer utility for the protection of on-line information against accidental or malicious destruction. This protection is achieved by preserving on magnetic tape recent copies of all items of information known to the on-line file system. In the event of a system failure, file system damage is automatically assessed and missing information is recovered from backup storage. For isolated mishaps, users may directly request the retrieval of selected items of information. The design of the backup mechanism presented in this thesis is based upon an existing backup mechanism contained in the Multics system. As compared to the present Multics backup system, the new design lessens overhead, drastically reduces recovery time from system failures, eliminates the need to interrupt system operation for backup purposes, and scales up significantly better with on-line storage growth.

This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, on August 13, 1973 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Among all engineering endeavors, perhaps no concern is more universal, nor more influential, than that of reliability. Certainly, in this respect, computer system design is no exception. When designing computer systems, one must admit from the outset that no hardware device is incapable of failure. Furthermore, programming errors, operational errors, and other human mistakes are inevitable. Thus, computer systems must be prepared to cope with all adverse contingencies ranging from minor recoverable malfunctions to total catastrophes which prevent continued system operation.

The protection of stored information against involuntary destruction is the final backstop of all reliability measures within a computer system. Although mishaps from time to time may cause individual computations to be aborted, or system service to be temporarily discontinued, the cost and inconvenience associated with such disruptions is minimized when all information is preserved. This thesis presents a design for a backup mechanism to be incorporated in a general-purpose time-sharing system for the protection of all on-line information against accidental or malicious destruction.

This protection is provided automatically by the system without requiring any effort on the part of individual users.

Motivation

Due to their simple nature, early computer systems were typically quite resilient. No information stored in memory was essential to continuing operation following a system "crash". Because permanent storage for all user data and programs was maintained outside the system, usually on punched cards or magnetic tapes, the impact of failures was restricted to those jobs being processed at the time of failure. By obeying just one cardinal rule, namely to refrain from destroying input data before successful job completion, full recovery was generally possible. Jobs could be restarted either from scratch or from a planned checkpoint. Of course, the susceptibility of tapes to "clobbering" and the susceptibility of punched cards to crimping and scattering was sometimes sufficient incentive for computer users to keep spare copies of important information.

Owing to numerous innovations in computer system design, modern computer systems dwarf their ancestors in both power and complexity. Ironically, however, these sophisticated systems, particularly time-sharing systems, have grown increasingly sensitive to mishaps. System data necessary for continued operation is commonly maintained both in main memory and secondary on-line storage, and is therefore liable to system failures. User programs and data are no longer stored externally, but instead are entrusted to the care of an internal file system which automatically manages the use of on-line storage devices. Thus, unlike the situation in early computer systems, all user information is constantly threatened by potential system failures. The common practice of updating files in-place may often leave inconsistent those files which were active at the time of a system crash. For this reason, and others, the restarting of interactive processes is, in general, neither feasible nor meaningful. Thus, recovery from failures in modern time-sharing systems usually amounts to salvaging as much on-line information as possible, in some cases restoring backup copies, and making the system available again to users.

Clearly, in order to insure reliability and security, measures must be taken to prevent the involuntary destruction of both user and system information. Typically, a variety of ad hoc methods are employed to protect critical system data. Most time-sharing systems, however, assume little or no obligation to protect user files against accidental harm. Although many of these systems are, indeed, very reliable, none are infallible. Nor are their users infallible. Furthermore, none of these systems are invulnerable to external dangers such as fires, floods, explosions, etc. Consequently, users are forced to provide their own insurance against mishaps. Often the contention is made that a particular system is "reliable enough" for users to do without a system-provided backup mechanism. Unfortunately, this attitude ignores the fact that reliability is seductive. The more reliable a system becomes, the greater the temptation to avoid the cost and nuisance of trying to prepare for potential mishaps. Thus, a user needs the kind of insurance that can only be provided by an automatic backup mechanism.

In principle, at least, it is clear that a file system should be fully responsible for the safekeeping of all information entrusted to it. Some time-sharing systems have, in fact, begun to acknowledge this responsibility. Many systems, for instance, provide a primitive backup mechanism such as the copying of all user files once each night. Mechanisms of this kind, while better than none at all, leave much room for improvement. Most noticeably, few users are willing to accept the loss of a full day's work. No doubt, users of systems offering this level of protection still find it necessary to take precautionary measures of their own. Not only are these primitive backup mechanisms deficient from the user's viewpoint, but also from the perspective of the system maintainer. As storage capacity increases with system growth, so too will the time required to copy the whole of on-line storage. Hence, backup overhead will increase in proportion to storage volume. This implies that with continuing storage expansion, backup overhead will consume an increasingly larger fraction of total system processing time. Clearly, at some point, it simply becomes infeasible to perform daily copying of secondary storage. Decreasing the frequency of backup copying will reduce overhead, but will sacrifice the quality of backup protection.

In this thesis, the need for a system-provided backup mechanism, and accordingly, the design of such a mechanism, are evaluated within the context of a computer utility. Although no existing computer system has fully earned the title of "computer utility", several large-scale general-purpose time-sharing systems (e.g. Multics [1], TSS/360 [2], MTS [3]) have made much progress in this direction. Among these, however, only Multics accepts full responsibility for the safekeeping of all information entrusted to its file system. Many, if not most, users of the computer utility of the future are expected to possess little knowledge of the computer system itself beyond the minimal skills necessary to make use of the various services offered. As in all public utilities, it is essential that the customer not be burdened with technical details. Thus, it is unreasonable to expect a technically unsophisticated user to assume responsibility for insuring against mishaps. A comprehensive system-provided backup mechanism is therefore not merely a convenience, but rather a necessity, in the computer utility of the future.

Objectives

The backup mechanism described in this thesis is intended to augment storage reliability within a computer utility to the point where users may confidently entrust their only copies of information to an internal file system. The underlying assumption in the design of such a mechanism must be that system failures are relatively rare. Hence, the devotion of an immoderate proportion of system resources to backup protection is unwarranted. On the other hand, if users are to feel confident in entrusting information to a computer system, then clearly some limit must be placed on the amount of work which might be lost due to a system failure, user mistake, or other mishap. This time limit defines the "resolution" of the backup mechanism. Another important measure of backup performance is recovery time. Clearly, a computer utility must be capable of quick recovery in order that users may depend upon the availability of its services.

The backup mechanism must offer a reasonable compromise between cost and level of service for various environments. Dissimilar purposes and economic considerations at different sites may dictate different demands for backup protection. Therefore, resolution and recovery time must be adjustable. Even within a single installation, the demand for backup protection may change. For example, in a system undergoing development, one might expect system stability to improve gradually. From time to time, however, major system modifications may cause periods of anticipated risky service during which better than usual resolution and recovery time might be desired. The environment may also change due to system growth. The backup mechanism must, in the face of increasing system size, be able to maintain a constant level of service without consuming an exorbitant share of system resources.

The backup mechanism is geared to the needs and finances of the "ordinary" user. It is assumed that the ordinary user can tolerate small setbacks, perhaps one hour's work lost, provided such setbacks are sufficiently rare; he cannot, however, tolerate exorbitant daily overheads. Although, as stated above, resolution must be adjustable, the range of adjustment is not meant to satisfy the needs of certain applications for which even the smallest losses would be intolerable. Many on-line business applications, e.g. an irline reservation system or a purchase order entry system, fit this category. Typically, such special-purpose systems maintain journal tapes on which are logged all processed transactions and/or all data base updates.[4] Usually, these applications employ a dedicated computer. If, however, such a system were to coexist with ordinary users within a computer utility, it is not expected that the general backup mechanism described here would eliminate the need for the keeping of journal tapes or other highly specialized backup measures. It is expected, however, that the backup mechanism will entirely replace those precautionary measures commonly left to the discretion of individual users such as the storing of spare file copies off-line. Also, it expected that many applications able to tolerate occasional small losses will find the protection offered by the general backup mechanism to be satisfactory.

One objective more of interest to the system designer than the ordinary user is to systematize and automate as much as possible the task of recovery from system failures. Automation is, of course, desirable to speed recovery and to minimize the possibility of operational errors. Also, automated recovery procedures can precisely assess losses, thereby minimizing the recovery effort required and preserving as much information as possible. Although it is clear that recovery procedures must by nature be intimately familiar with specific system characteristics, a high-level organization is presented which is largely system-independent.

The backup mechanism represents a general solution to the problem of protecting stored information against accidental harm. No explicit consideration need be given to techniques for protecting specific types of system data such as accounting records, user identification information, batch job queues, etc. It is assumed that all such information is entrusted to the file system and therefore is uniformly protected by the general backup mechanism.

Background

The backup mechanism described in this thesis is based upon an existing backup mechanism incorporated in the Multics system. Multics, conceived as a prototype computer utility, has included a general backup mechanism since its original design.[5,6] (1) In fact, CTSS [7,8], the predecessor of Multics, also included such a facility. (2)

(1) An early design and implementation plan for the Multics backup system is described by G. F. Clancy in a section of an unpublished Multics design notebook produced at M.I.T. Project MAC (August 18, 1966).

(2) Some aspects of the CTSS backup facility are described by M. J. Bailey and R. C. Daley in an unpublished M.I.T. Computation Center programming staff note entitled "A CTSS Secondary Storage Back-up and File Retrieval Scheme" (September 15, 1964).

As a result, the Multics file system is so reliable that users routinely entrust their only copies of information to it without hesitation. Hence, the common practice of storing spare copies of programs and data on punched cards or magnetic tapes has disappeared from the work habits of Multics users. The burden is instead borne by an automatic backup system.

The approach that has been taken by Multics in providing backup protection is to periodically copy newly created or modified files from on-line storage onto magnetic tape (currently at one hour intervals). The information is thus isolated from the effects of system failures and user mistakes. Certain tapes are stored off premises to guard against local calamities such as fires or explosions in the computer room. The production of backup tapes, known as "dumping", is performed in anticipation of mishaps. When such mishaps do occur, the backup system is then called upon to discover and reclaim any lost information. In the case of a system failure, a process known as "salvaging" is employed to examine the file system directory structure and storage assignment tables for missing or inconsistent data. When the results of salvaging indicate a global catastrophe, i.e the destruction of a major portion of on-line storage, a process known as "reloading" is employed to recover the former contents of on-line storage from backup tapes. For minor mishaps, a process known as "retrieving" is used to recover selected items of information. These are the basic functions of the Multics backup system. It is this fundamental structure upon which the backup system described in this thesis is built.

Although the present Multics backup system must be applauded for having substantially bolstered file system reliability, it has not done so without high costs and unwarranted sacrifices. In an effort to limit work loss to a maximum of one hour, significant overheads have been incurred. In spite of this large resource investment, recovery time from major system failures is currently intolerably slow. Also, the system must be shut down periodically for backup purposes which contradicts a Multics objective of continuous service. Most discouraging of all, however, is the fact that the entire situation steadily worsens with system growth.

It is insightful to briefly review the history of the Multics backup system in order to understand how the present problems developed. Before beginning any implementation, the entire Multics design, including the backup system, was carefully specified in detail. Once Multics implementation began, however, priorities had to be assigned and among the many areas of the system requiring attention, the backup system was not given top priority. Still, some interim backup facility was needed as soon as possible. Hence, a temporary backup system, much less ambitious than the original, was rather hastily designed and implemented. Over a period of time, many changes and improvements were made to this interim facility, also with much hurry, and eventually the original design plans were forgotten.

As one might expect from this rather haphazard history, a well-coordinated design has not evolved. Various components of the backup system were developed independently and consequently their interactions were not carefully planned nor well understood. Thus, the major effort of this thesis will be to develop a top-down view of the backup system and to produce from this perspective a coordinated design. Admittedly, this design will be strongly influenced by the structure of Multics, particularly the Multics file system. Nevertheless, the basic design concepts are thought to be sufficiently general so as to be applicable to many systems of the computer utility category.

Thesis Plan

The thesis presentation is organized as a top-down description of a proposed backup system design for a computer utility. To establish a framework in which to discuss the backup system, chapter 2 first defines certain characteristics of an exemplary file system. The majority of chapter 2 is devoted to a high-level discussion of backup system strategy and organization. The various functions of the backup system, i.e. dumping, salvaging, reloading, and retrieving are described. For the reader interested only in the basic approach and fundamental structure of the backup system, chapter 2 provides a useful overview.

Chapter 3 investigates in further detail each of the backup system program modules introduced in chapter 2. Also, the interface between the file system and the backup system is carefully examined. Together, chapters 2 and 3 represent a complete description of the general backup system design.

A proposed Multics implementation of the backup system is discussed in chapter 4. Basically, certain additions and modifications to the Multics file system are needed to support the backup system components described in chapters 2 and 3. This implementation proposal is intended to demonstrate the suitability of the design for a prototype computer utility such as Multics.

Finally, chapter 5 reviews the results of this thesis and offers a few concluding remarks.

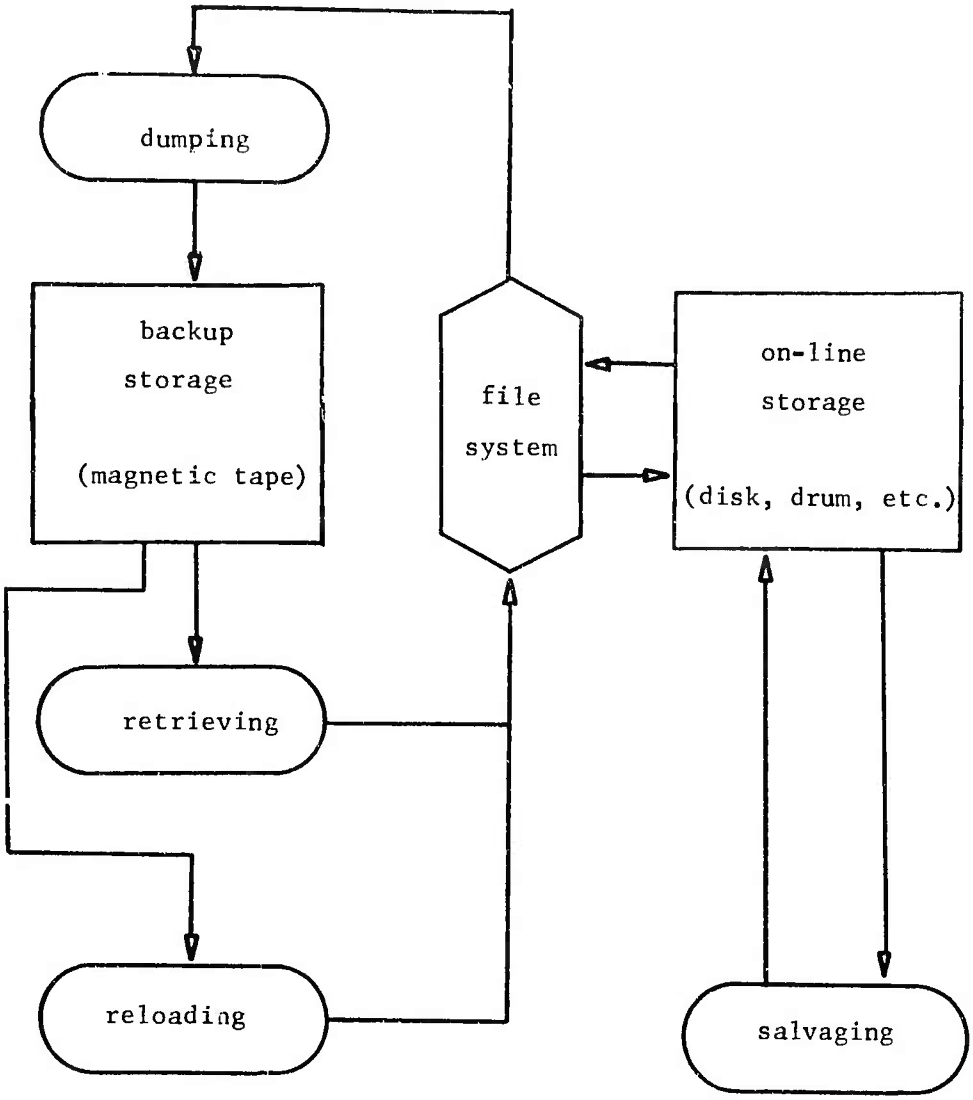
CHAPTER 2

STRATEGY AND ORGANIZATION OF THE BACKUP SYSTEM

The purpose of the backup mechanism proposed in this thesis is to insure the survival of information entrusted to a computer system. Naturally, the specific nature of the backup mechanism must depend to some extent on the specific nature of the computer system which it protects. In particular, the structure of the file system must influence the design of the backup mechanism. A few file system characteristics (borrowed primarily from the Multics file system) are presented at the beginning of this chapter. While by no means a complete description of a file system, these characteristics are sufficient to define an environment in which to present the backup system design.

The bulk of this chapter is devoted to an overview of the backup system strategy and organization. Figure 2.1 depicts the various functions of the backup system and their relationships to the file system. The dumping function is responsible for the preservation of recent copies of all items of information known to the file system. Magnetic tape is chosen as the backup storage medium since it is detachable and thus can be isolated from all perils of the computer system and the computer room. Also, magnetic tape is economical and commonly available in most computer

Figure 2.1: Schematic view of the backup system.



systems. In the event of a system failure, the salvaging function is employed to assess damages to the file system. Clearly, in order to verify file system consistency, the salvaging procedure must embody independent knowledge of the file system structure as represented in memory. If the results of salvaging indicate a loss of information, the reloading function is employed to automatically recover any missing files from backup tapes. For minor mishaps, the retrieving function is used to recover information from backup tapes at the request of users.

The File System

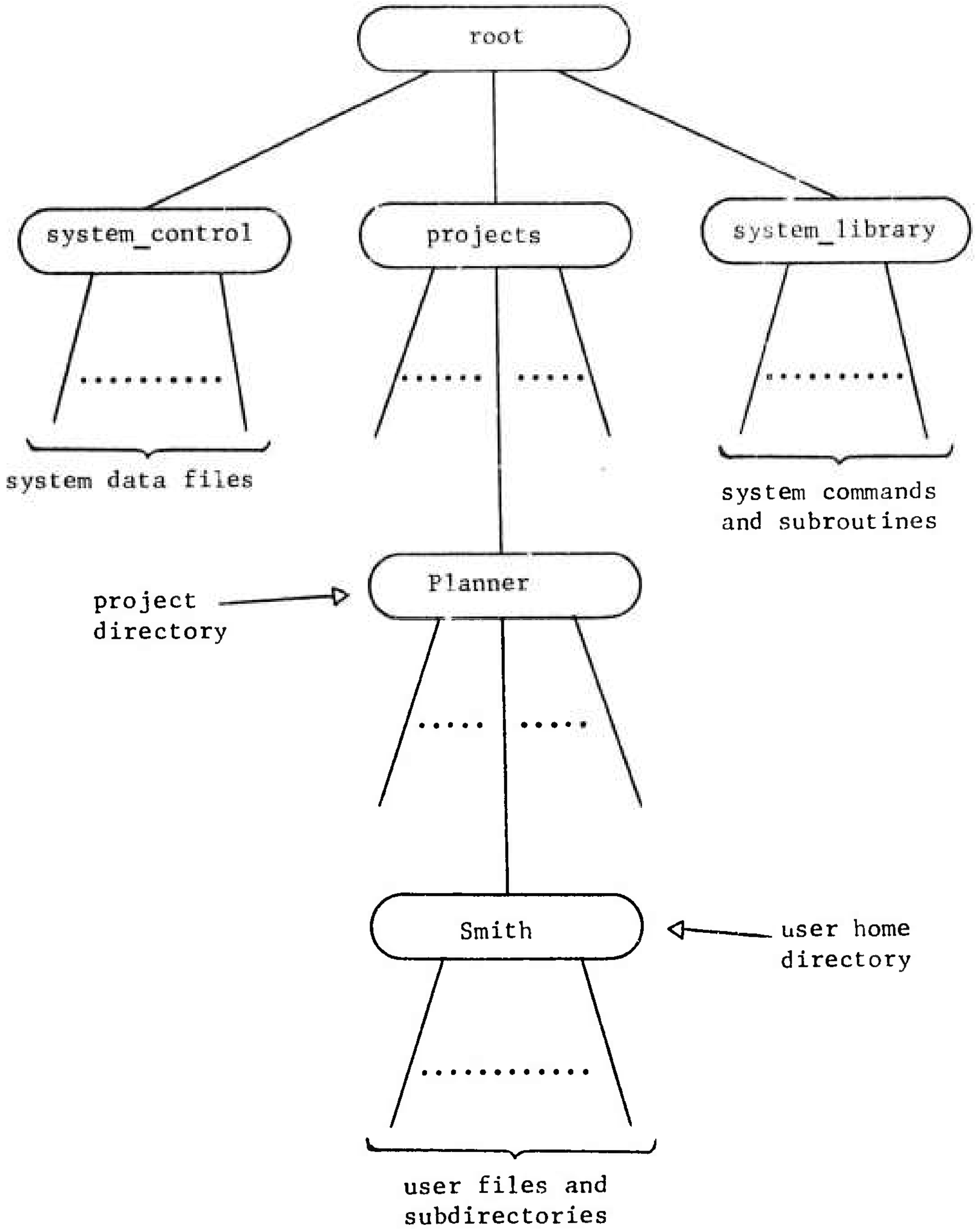
The file system considered in this thesis maintains two classes of objects, namely files and directories. A file is defined to be simply an ordered sequence of bits. As far as the file system is concerned, a file has no other logical structure. Any formats imposed upon a file are known only to higher-level system or user programs. Directories are used to catalogue files. A directory contains a list of entries, analogous to a table of contents. Each entry "points to" a particular file which is said to be contained in the directory in which the entry appears. An entry is essentially a collection of attributes which describe a file. Among these attributes, for example, are the file names, the physical storage location, the time of last reference, the time of last modification, and access control information. A directory also contains certain global attributes which apply to all of its entries. For example, a directory may have a quota which defines the maximum amount of storage space to be used by files contained in the directory.

Entries may describe directories as well as files and therefore one directory may contain another. This organization gives rise to a tree-structured file hierarchy. All files appear as terminal nodes within this tree whereas directories may appear either as terminal or non-terminal nodes. Except for the root directory located at the root of the tree, every file and directory is described by an entry contained in its parent directory. The root directory is defined to have level zero. Entries contained within the root directory describe files and directories at level one. Level one directories in turn contain entries for level two files and directories, and so on for deeper levels. A directory at level $\underline{n}$ is said to be immediately superior to a file or directory at level $\underline{n}+1$, and immediately inferior to a directory at level $\underline{n}-1$. Similarly, a directory a level $\underline{n}$ is said to be superior to a file or directory at level $\underline{n}+\underline{m}$, and inferior to a directory at level $\underline{n}-\underline{m}$ (for $\underline{m}>0$).

In order to uniquely specify a particular entry within the hierarchy, it is not sufficient to give only its entry name. An entry name is unique within its containing directory, but may not be unique with respect to all directories. Therefore a pathname is used to uniquely identify an entry within the hierarchy. A pathname consists of an ordered sequence of entry names which describe a path from the root to the entry of interest. The individual entry names are separated by the symbol ">". Thus, "root>A>B" is the pathname for file B which is immediately inferior to directory A which in turn is immediately inferior to the root directory.

Figure 2.2 shows how the file hierarchy might be used to organize information within a computer utility. This organization has been abstracted from the Multics file hierarchy. All system data files, e.g. accounting records and user identification information, are contained in a "system_control" directory. All system commands and subroutines are contained in a "system_library" directory. For administrative purposes users are grouped into projects, each of which is represented by a project directory. Immediately inferior to a project directory are found the home directories of individual users. A user may create an arbitrary subtree of files and directories below his home directory. The activities of a user, however, are not confined to his home directory alone. He may touch any parts of the hierarchy to which he has been granted access.

Figure 2.2: Example of a file hierarchy.



For example, all users are routinely granted access to use procedures in the system library. A user may grant other users permission to read or modify his personal files.

To this point, we have seen basically an external view of the file system, i.e. the logical structure of the file hierarchy as it appears to the user. This picture adequately defines a setting in which to discuss those backup functions which require only external knowledge of the file system. These are the functions which actually make use the file system, namely dumping, reloading, and retrieving. The salvaging function, however, requires knowledge of internal file system structure in order to verify file system consistency following a system crash. Hence, any discussion of salvaging must necessarily be more file system-specific than the discussions of the other backup functions. For this purpose, some characteristics of internal file system structure are presented below.

The internal file system organization is, of course, dependent upon the nature of the memory system. We will consider a two-level memory configuration composed of main memory and secondary on-line mass storage. Furthermore, we will assume that the transfer of information between main memory and secondary storage is automatically managed by a segmented, virtual memory system.[9] Both files and directories exist as segments in the memory system, i.e.

variable-length collections of bits. Segments are subdivided into fixed-length blocks called pages which are moved in and out of main memory by the virtual memory system.

Before a segment can be referenced, it must first be initiated into the virtual address space of a process. (1) The initiation of a segment is somewhat analogous to the opening of a file in a conventional file system. At the time a file is initiated, a user may request that some or all of his allowed access modes (e.g. read, write, etc.) be enabled. Whenever a file is initiated with write access, a notation is made in the entry of the file. Once a segment has been initiated, it becomes directly addressable and hence can be manipulated in essentially the same fashion as main memory is manipulated in a conventional computer system. Thus, both files and directories are rather dynamic objects in a segmented virtual memory system. This is in contrast to conventional file systems in which files and directories are comparatively static objects. When no further use is to be made of a segment, the segment is terminated from the virtual address space of a process. The termination of a segment is somewhat analogous to the closing of a file in a conventional file system.

(1) We informally define a "process" to be the ongoing computation associated with an individual user.

When a segment is not actively in use, its pages reside in secondary storage. Each entry in the hierarchy contains a page map which indicates the secondary storage addresses of all pages of its associated segment. In order to keep track of secondary storage usage, the file system maintains a storage assignment table for each secondary storage device. Because segments may grow and shrink dynamically, it is necessary that storage assignment tables be kept either partially or fully in main memory to permit fast allocation of storage space. For simplicity, we will assume that storage assignment tables are stored entirely in main memory during normal system operation. Of course, if detachable storage devices are used, then only the storage assignment tables for attached devices need be kept in main memory. Different storage allocation strategies are compared in the discussion of salvaging.

When a segment is actively in use, its pages are transferred in and out of main memory as needed. Before any pages of a segment can be brought into main memory, however, a page table must first be constructed for the segment in main memory. A page table is an array of page table words, each of which holds the address of a page. Intuitively, a page table serves as a working copy of a page map. A segment having a page table in main memory is said to be "active". As a page of an active segment moves between main memory and secondary storage, its corresponding page table word is updated so as always to reflect the current address of the page. When a new page of a segment is created in main memory, its address is recorded in a previously empty page table word. At some later time, when this new page is removed from main memory, a free location in secondary storage is selected from the storage assignment tables. This new secondary storage address is recorded in the page table word for the new page. Eventually, the segment containing the new page will be "deactivated", i.e. its page table will be removed from main memory. At that time, the secondary storage address of the new page is updated into the page map in the entry for the segment.

The brief description given above is intended to characterize how files and directories are accessed and how storage space is allocated. The significance of these two features will be explained in the discussion of salvaging. Other issues of internal file system organization are omitted since they do not significantly influence the design of the backup system.

Incremental (Primary) Dumping

An activity known as incremental dumping is principally responsible for keeping the backup system abreast of modifications to the file hierarchy. The purpose of incremental dumping is to discover newly created or modified files and directories and to copy these files and directories onto backup tapes. Incremental dumping is performed during normal system operation by the incremental dumper, a system process which awakens periodically to scan the hierarchy for modifications. A single such pass over the hierarchy constitutes one incremental dump. The net effect of incremental dumping is to limit the amount of information which can be lost to those modifications which have occurred since the last incremental dump.

In order to restrict the maximum time span during which modifications to the file hierarchy can go unnoticed by the backup system, it follows that the incremental dumper should scan the hierarchy frequently. On the other hand, because the incremental dumper competes with ordinary users for system resources, it becomes economically desirable to lower the incremental dumping rate. Therefore, the time interval between the start of incremental dumps is chosen as a compromise between these two considerations. It is assumed that the incremental dumper enjoys a sufficient scheduling priority to insure completion of a full hierarchy scan within the desired time interval. A typical interval is expected to range from as little as thirty minutes to as much as a few hours depending on the degree of protection desired.

The dumping of an entire file to record modifications encompassing only a small fraction of that file can be a rather expensive and heavy-handed measure. Instead, one might consider dumping only those pages of files which have, indeed, been modified. The dumping of pages rather than whole files will, of course, complicate recovery procedures. Therefore, if the maximum file size is limited and the median file size is small, it is simpler and not much more costly to dump files in full. On the other hand, if the maximum file size is unrestricted or very large, if the median file size comprises many pages, and if file modifications tend to be localized, then dumping pages might be more practical. For simplicity, we will hereafter assume the dumping of whole files. We will return to this issue in the discussion of recovery operations.

The cost of incremental dumping will be proportional to the average number of files and directories modified per incremental dump interval. Intuitively, one would expect this modification rate to be roughly proportional to the processing activity of the system, but to be independent of total secondary storage usage. This assumption is based on the observation that as a typical user accumulates more storage, he still tends to maintain a temporal locality in his file modifications. Hence, it is expected that secondary storage growth will have little influence on incremental dumping costs. Increases in processing activity, on the other hand, should produce a proportional increase in incremental dumping corts. This behavior has, in fact, been observed for the Multics system.

Secondary Dumping

In the event of a mishap, it would be possible to recover the entire file hierarchy by searching all incremental tapes produced since the beginning of system operation. It is necessary that all these tapes be examined since only recently modified files appear on recent tapes whereas files not modified for long periods of time appear only on older tapes. Obviously, as the age of the system increases, this approach becomes less and less practical. Recovery time would soon become intolerable, as would the burden of maintaining an ever expanding archive of backup tapes. Therefore, it is necessary to consolidate the information contained on incremental tapes.

Secondary dumping is the mechanism used to accomplish this consolidation. A secondary dump scans the hierarchy during normal system operation collecting all files and directories which have been incrementally dumped later than some specified time in the past. Since that time, some number of incremental dumps will have occurred. Many evolving versions of a file can appear within those incremental dumps, but only the latest version will appear in the secondary dump. Furthermore, files which appear in an incremental dump and are later deleted will not appear in a subsequent secondary dump. Therefore, a secondary dump can be substantially smaller than the set of incremental dumps which it supersedes. Generally, the larger the set of incremental dumps superseded, the greater the size reduction.

Clearly, in order to limit the number of backup tapes needed to recover the full hierarchy, all incremental dumps performed since the beginning of system operation must be consolidated periodically. This is essentially equivalent to dumping the entire file hierarchy (except for a few files which are so new that they have not yet been incrementally dumped). A secondary dump which supersedes all previous incremental dumps is called a complete secondary dump. A complete secondary dump represents a cutoff beyond which no older backup tapes need be inspected in order to find the latest backup copies of all files and directories which still exist in the file hierarchy.

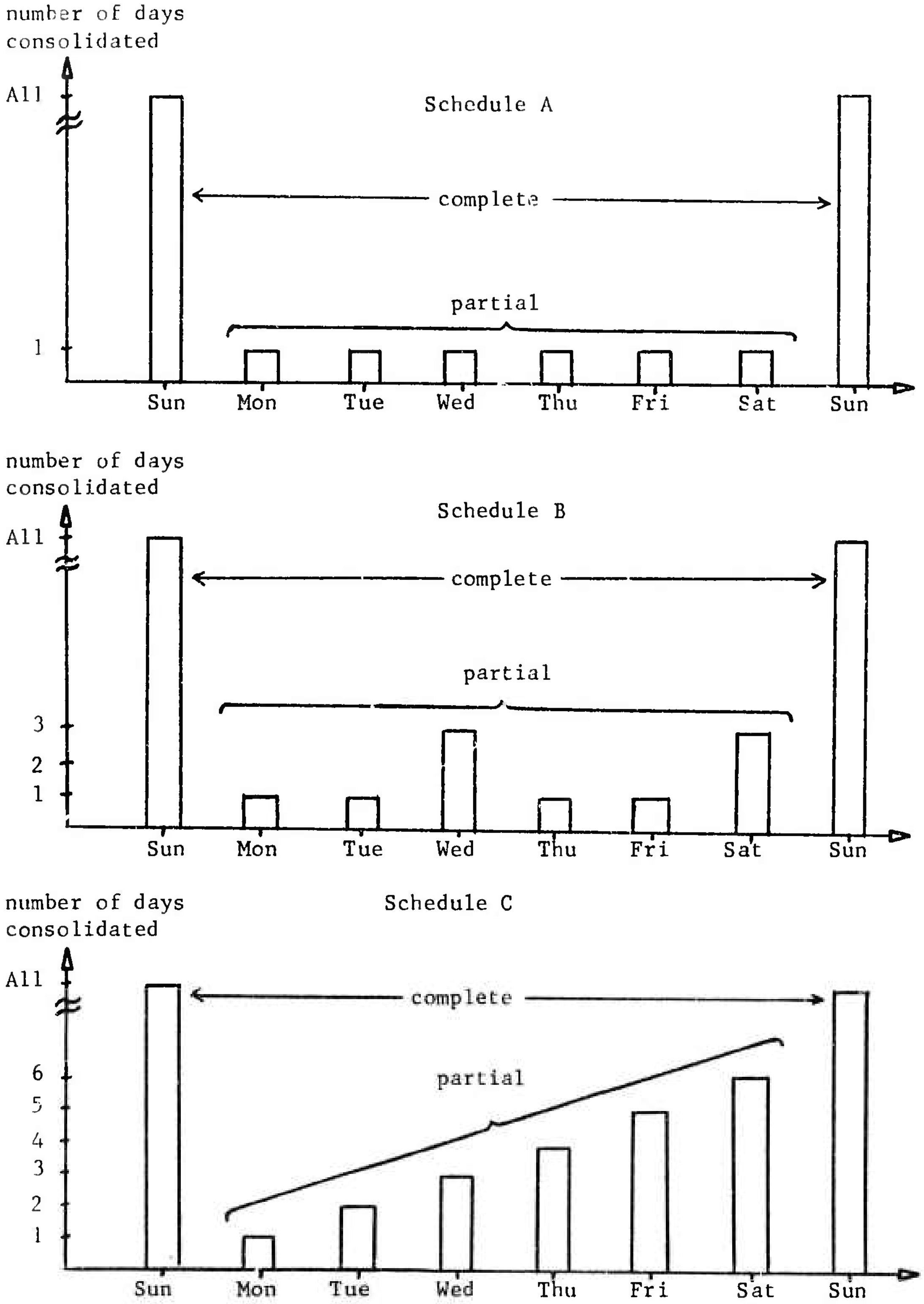
The cost of a complete secondary dump will, of course, be proportional to secondary storage size. For a large file hierarchy, a complete secondary dump will necessarily be very time-consuming and demanding of system resources. Therefore, to avoid excessive dumping overhead, the time interval between complete secondary dumps should be substantially greater than the interval between incremental dumps. On the other hand, the longer the time interval between complete secondary dumps, the larger will be the average and maximum number of incremental dumps needed to recover the file hierarchy in the event of a mishap. Therefore, the frequency of complete secondary dumps is chosen as a compromise between cost and recovery time.

The expense of performing daily complete secondary dumps is likely to be prohibitive in a large computer utility attempting to provide continuous service. Therefore, let us consider a time period of one week or longer between complete secondary dumps. As an intermediate measure for limiting recovery time, secondary dumping can be used to consolidate some or all of those incremental dumps produced since the latest complete secondary dump. A secondary dump used in this fashion is called a partial secondary dump.

Partial secondary dumping is similar in nature to incremental dumping. In effect, a partial secondary dump collects files and directories which have been modified over a period of time encompassing some number of preceding incremental dumps. Let us refer to this period of time as the "consolidation interval" of a partial secondary dump. The cost of a partial secondary dump is then proportional to the number of files and directories modified per consolidation interval. As argued earlier in the case of incremental dumping, this modification rate is roughly proportional to system processing activity, but is negligibly influenced by secondary storage size.

Partial secondary dumping can be employed in a variety of ways. Once again, the principal trade-off is one of cost versus recovery time. Three examples of secondary dumping schedules are illustrated graphically in figure 2.3. The schedules are arranged in order of increasing dumping costs and decreasing recovery time. Complete secondary dumps are performed once per week in each of these schedules whereas partial secondary dumps are used differently among the three. In schedule A, each partial secondary dump consolidates exactly one day's volume of incremental dumps. Hence, the number of partial secondary dump tapes needed for reloading increases linearly through the week, while dumping costs are fairly uniform. In schedule B, the partial secondary dumps performed Wednesday and Saturday consolidate three days' volume of incremental dumps. Hence, a recovery operation performed on Thursday, for example, would not involve partial secondary dumps from Monday and Tuesday since these are superseded by Wednesday's dump. In schedule C, each partial secondary dump supersedes all preceding partial secondary dumps performed the same week. Therefore,

Figure 2.3: Sample secondary dumping schedules.



both dumping costs and recovery time increase gradually through the week.

It should be emphasized that the terms "partial" and "complete" refer to the temporal nature of a secondary dump. A secondary dump is called "partial" not because it dumps a part of the hierarchy, but because it supersedes a part of the incremental dumping history of the system. Similarly, a secondary dump is called "complete" not because it dumps the entire hierarchy, but because it supersedes the complete incremental dumping history of the system. In fact, a complete secondary dump actually does not dump the entire hierarchy. As mentioned above, new files which have not yet been incrementally dumped are ineligible for secondary dumping. Also, for reloading purposes, the system library is ignored by all incremental and secondary dumps. Instead, special dumps of the entire system library are performed either periodically or whenever the number of newly installed programs reaches a certain limit.

As compared to incremental dumps, secondary dumps naturally require a longer time to run. Since the resolution of the backup system is determined by the incremental dumping rate, and because this resolution should be kept constant, incremental dumping cannot be postponed while secondary dumping is in progress. Therefore, incremental and secondary dumps must be able to run concurrently. When they do run concurrently, a considerable load will be placed upon the system. Therefore, it is obviously desirable that secondary dumps be performed at times of light user load, e.g. late night and early morning hours. If a complete secondary dump should require more than a single night to run, it may be advantageous to stagger complete secondary dumping of different sections of the hierarchy over several nights rather than compete with heavier user loads during the daytime.

In addition to reducing recovery time, secondary dumping also reduces the number of backup tapes that must be retained for long periods of time. If one were only interested in recovering the latest copies of files, then as soon as a backup tape was superseded by a later dump, it could be reused. Of course, situations arise in which the latest backup copy of a file is of no use. If, for example, a file is erroneously modified due to a programming bug, then the clobbered file will appear in subsequent incremental and secondary dumps. Hence, the latest backup copies will be of no value, but an older copy can be used to restore the file.

Another reason for not quickly recycling backup tapes is that the added redundancy increases reliability. If for any reason a backup copy of a file cannot be read from a particular backup tape, then at least the next most recent copy can be obtained from an older backup tape. Clearly, the writing of duplicate backup tapes for each dump would improve reliability. But it is important to recognize that tapes written in duplicate are to some extent susceptible to the same errors. Therefore, the temporal separation between different dumps is an important reliability factor which argues for the prolonged retention of backup tapes.

To a limited degree, tape retention time represents a compromise between cost and reliability. However, since the cost of adding more tape reels to a backup archive is basically a small one-time expense, one can usually afford to be generous in this respect. On the other hand, the maintenance of an extremely large library of backup tapes is cumbersome, space-consuming, and prone to operational errors. Therefore, rather than hold all tapes for an equal length of time, it seems more practical to hold a small number of tapes for very long periods of time and the majority of tapes for only a comparatively short time. For example, one might decide to retain incremental dumps for one month, partial secondary dumps for three months, and complete secondary dumps for one year. This strategy recognizes the fact that when a mishap occurs which destroys a file, a backup copy is usually recovered within a short period of time. For the rare case of a user who wishes to recover a file deleted many months ago, the backup system can still offer assistance if the file existed long enough to be recorded on a complete secondary dump.

In order to facilitate recovery operations following a system failure, secondary dumps have been given certain special properties. Secondary dumps always include all directories in the hierarchy as well as all system data files necessary for normal system operation. Complete secondary dumps, of course, already include all files and directories, and hence implicitly satisfy this requirement. Partial secondary dumps will typically include many directories and most system data files anyway, and therefore only a small additional overhead is incurred. The backup system maintains in the entry of each file the tape address, i.e. reel number and record number, of the latest secondary backup copy. The purposes of these special features will be explained in the discussion of reloading.

User-Controlled Backup

As described thus far, the user has no control over the production of backup copies. Whenever an eligible file is encountered by a dumping process, a backup copy of the file is dumped. Hence, a user has no means of controlling, or even determining, precisely when a particular file is dumped relative to any work in progress involving the file. This mode of operation creates two related problems. First, a file may be inconsistent at the time it is encountered by the dumping process. Second, if the creation of backup copies is not synchronized with the user's work, he will find it difficult to discover precisely what modifications were performed before and after a given backup copy was produced.

Since, for any file, the meaning of consistency depends upon its application, the backup system cannot itself determine whether or not a file is inconsistent. The file system does keep track of when a file is potentially inconsistent, i.e. when it is open to modification. Hence, it would be possible for the dumper to simply skip potentially inconsistent files. This approach, however, would provide poor protection for heavily used files which remain open to modification for long periods of time. Even for files which are only briefly subject to modification, unfortunate timing of updates could cause a file to miss several consecutive incremental dumps. Thus, the dumping of a file would become a chance occurrence and, in general, the resolution of the backup system would suffer.

The current Multics backup system entirely ignores the problem of consistency. Files are dumped regardless of whether or not they are open to modification. Surprisingly, however, files are rarely dumped in an inconsistent state. To understand this result, it is necessary to examine the ways in which files are ordinarily modified. Most programs are deliberately cautious about modifying permanent files. For example, a text editor accumulates changes to a text file in a buffer area. The text file is only updated from the buffer at the explicit request of the user. Although the text file remains open to modification during the entire time the editor is in use, modifications are actually brief and infrequent. Similarly, when one compiles a program, the object code is compiled in a temporary area. If the compilation succeeds, this temporary area is then copied into a permanent file in one quick operation. This pattern is typical of many permanent file modifications. In situations of this kind, the probability of dumping a file while in an inconsistent state is small. Even when this does happen, there is still some consolation in the fact that the modifications responsible for the inconsistency will qualify the file for the next incremental dump. Hence, a consistent copy is eventually produced.

In the situations mentioned above, i.e. editing and compiling, it is not essential that a user be able to specify the exact time at which a backup copy is produced. A user will probably be satisfied to know that a backup copy is produced sometime within the next incremental dump interval. The situation is somewhat different, however, if one considers applications involving more sophisticated data management problems. To illustrate, let us imagine some business-like application which processes "transactions", each of which results in an update to a data file. As mentioned in chapter 1, large-scale applications of this type typically maintain journal tapes on which each transaction and/or each data base update is logged. The computer utility, however, may support many small-scale applications of this nature which do not require, nor deserve, specialized backup mechanisms of their own. These small applications may find the level of protection provided by the general backup mechanism to be adequate.

In order for the backup mechanism to be of significant use to a transaction-oriented application, the application program must be given more control over the production of backup copies. In particular, some method for specifying that a particular snapshot of a file be dumped seems necessary. This would not only allow the application program to guarantee the consistency of backup copies, but would also permit the application program to recognize the precise moment at which a backup copy was produced relative to the transactions being processed. For example, suppose that transactions were input to the application program from a terminal. The printed copy of the transactions produced at the terminal could serve as a log. Periodically, the application program would set aside copies of its data base to be dumped. The times at which such backup copies were set aside could also be recorded at the terminal. In the event of a mishap, a particular backup copy could be retrieved and subsequent transactions could be resubmitted.

The important point of the above example is that many applications require some control over the production of backup copies. The application program must be able to specify when backup copies are created. These copies, however, need not be immediately written to tape. Instead, they could be temporarily stored in the file hierarchy until encountered by the incremental dumper. In this way, user processes and the dumper process may continue to operate asynchronously.

In order to provide users with a facility for specifying when backup copies are to be created, each file entry in the hierarchy contains a switch to indicate whether or not user-controlled backup is desired. At the time a file is initiated with write access, a user may optionally request that this switch be turned on. If he does so, a "shadow" copy of the file is created and stored in the same directory as the original file. Thereafter, any attempt to dump the original file will result in the shadow copy being dumped. This permits the user to freely modify the original file without fear of inconsistent backup copies being produced. The user may, at his convenience, request the file system to update the shadow copy from the original file. By doing so, he permits the backup system to dump successively more recent versions of his file. When the user is finished processing the file, he requests the file system to terminate the file. At this point, the shadow copy is discarded (since the original file is now itself consistent) and the switch indicating user-controlled backup is turned off.

In order to distinguish and identify different versions of files, the time of last modification is recorded with every backup copy. Clearly, this modification time can only be changed while a file is initiated with write access. Therefore, at the time a user terminates a file which had been initiated with write access, the file system returns to him the latest modification time of that file. This modification time can be used to identify any subsequently dumped backup copies. However, a file may also be dumped during the time it is initiated with write access. If user-controlled backup is employed, the file system will return to the user the modification time of his file each time he updates a shadow copy. This modification time is stored with the shadow copy and is eventually recorded on tape with a corresponding backup copy. Hence, if user-controlled backup is employed, a user can identify every backup copy of a given file. If user-controlled backup is not employed, the user will not be explicitly aware of the modification times of those backup copies produced while a file is initiated with write access. Note that whenever a file is reloaded or retrieved, its modification time is restored.

One point deserving consideration is the question of how frequently users should update shadow copies. Clearly, in order to take full advantage of the backup system, shadow copies should be updated at least once per incremental dump interval. However, the use of shadow copies still introduces an extra delay which effectively weakens the resolution of the backup system. To see this, let us assume an incremental dump interval of one hour. If user-controlled backup is not employed, a modified file may wait at most one hour before being dumped. Similarly, if user-controlled backup is employed, an updated shadow copy may wait at most one hour before being dumped. However, the file modifications contained in the shadow copy were actually performed prior to the shadow copy update. Therefore, if shadow copy updates are performed only once per hour, then, at worst, a modified file may wait one hour before a shadow copy update is performed and another hour before being dumped. Hence, on the average, backup system resolution is reduced by a factor of two.

In view of this extra delay introduced by user-controlled backup, many users may wish to update shadow copies more frequently than once per incremental dump interval. For example, a user may wish to update a shadow copy every ten minutes, i.e. six times per hourly incremental dump cycle. In this case, a modified file may wait at most 70 minutes before being dumped. Hence, backup system resolution is only slightly reduced. Notice that only one of every six shadow copies will actually be dumped. The others will simply be replaced by more recent shadow copies before being dumped. The purpose of performing six shadow copy updates per hour is not to produce six backup copies per hour, but rather to insure that the one backup copy produced each hour is at most ten minutes old when dumped.

Salvaging

Whenever a system crash occurs, the integrity of the file system must be questioned. In particular, the consistency of directories and of storage assignment tables need be verified. Also, users should be warned of potentially inconsistent files, i.e. those files not properly terminated before the crash occurred. A procedure known as salvaging is employed to detect, report, and correct wherever possible any inconsistencies in the file hierarchy and the storage assignment tables. Those parts of the file hierarchy which cannot be salvaged must be discarded and later recovered from backup tapes.

When main memory loss is not the cause of failure, it may often be possible to recover information from main memory before beginning salvaging. In Multics terminology, this operation is known as an "emergency shutdown". Essentially, an emergency shutdown attempts to revive the system in order to perform a regular shutdown. First, an attempt is made to deactivate all active segments. This involves flushing all pages from main memory to secondary storage and updating the page maps of all active segments whose page tables have been modified. After all segments have been deactivated, the storage assignment tables are copied to secondary storage. The file system is then in a dormant state.

Clearly, the success of the emergency shutdown operation depends to a large extent on the consistency of file system data bases following a system crash. If these data bases have been left in grossly inconsistent states, then the operation is likely to fail. On the other hand, if file system data bases are left consistent, or nearly so, then an emergency shutdown can often succeed. Notice that an emergency shutdown is performed using the standard file system. Hence, if an emergency shutdown runs to completion, one can be fairly confident that no major inconsistencies exist in the file system. There is some small risk, however, that minor inconsistencies may go unnoticed.

The salvaging program, called the salvager, performs three functions: (1) verification of the internal consistency of all directories, (2) reconstruction of the storage assignment tables, and (3) notification to users of potentially inconsistent files. All of these functions are carried out simultaneously during one scan over the directory structure of the file hierarchy. If one wishes to trust the file system following a successful emergency shutdown, a special salvaging mode could be used to perform only the third function.

All directories are thoroughly examined by the salvager to verify internal consistency. Any detected errors are reported and , if possible, corrected. Certain problems can be trivially cured. For example, if an entry was left active, the active indicator is simply reset. The use of redundancy in the directory format can help to correct more serious errors. For example, if entry lists are doubly threaded, a single break in an entry list can be repaired. Still more serious errors may necessitate deletion of entry attributes, single files, or entire directories. In order to facilitate the recovery of files and directories which could not be salvaged, those directories from which entries have been lost are marked by the salvager.

As the salvager reads through the directory structure, it rebuilds the storage assignment tables. This is done by simply noting the secondary storage addresses used by each segment. It may happen, as a result of a crash, that two segments claim the same page in secondary storage. One way to resolve this conflict is to store a segment identifier and page number with every page.

While scanning the directory structure of the hierarchy, the salvaging procedure checks every file entry to see if it was initiated with write access at the time of the crash. If so, a flag is set in the entry which will cause an error code to be returned on future attempts to initiate the file. In this way, users are warned of potentially inconsistent files. If a shadow copy exists for a potentially inconsistent file, both the original file and the shadow copy are retained in the file hierarchy. A user may discover that his original file is hopelessly inconsistent in which case he can replace the original file with the shadow copy. Alternatively, a user may find that his original copy is consistent or can be made consistent in which case the shadow copy can be discarded. Until the user makes this decision, the backup system continues to use the shadow copy for dumping purposes.

Among the most disasterous of system crashes are those caused by the failure of a secondary storage device.

Clearly, the sensitivity of the file system to this kind of loss will depend largely on the storage allocation strategy used. For example, secondary storage might simply be viewed as one large allocation pool in which different pages of the same segment can be arbitrarily assigned to different devices. In this case, the failure of a single device may destroy or damage an undetermined number of files and directories, and is therefore likely to be catastrophic. Merely requiring that all pages of a segment be on the same device adds little improvement since the loss of a directory on one device will effectively destroy inferior files and directories on other devices. Therefore, a more reliable allocation strategy would require that a directory stored on one device should have no inferiors stored on other devices. In other words, each device should hold a subtree, or perhaps several subtrees of the file hierarchy. Using this allocation strategy, the extent of damage due to a device failure is limited to a well-defined portion of the file hierarchy. Therefore, both salvaging and subsequent file and directory reloading is simplified.

Of course, while this allocation strategy is admirable for reliability reasons, it has its disadvantages as far as file and directory growth are concerned. Subtrees confined to a device with no free space available will be unable to expand even though free space may exist on other devices.

To avoid this situation, substantial amounts of free space must be left on each device to accomodate potential expansion which might result in considerable space wastage. For this reason, it may be desirable to assign one very large subtree or several smaller subtrees to a pool of two or more storage devices so as to reduce space breakage. However, if subtrees are permitted to span more than one device, reliability is also reduced. It is convenient to think of subtrees being assigned to "logical devices" where a logical device may be some fraction of a physical device, a whole physical device, or several physical devices. The size of logical devices can then be chosen as a compromise between reliability and space breakage considerations.

The root directory and its closest descendants will form a superstructure for the file hierarchy below which all of the separately assigned subtrees will exist. The loss of this superstructure will, of course, necessitate reloading all of on-line storage. Therefore, to lessen the probability of such disasters, each directory of the superstructure should be duplicated on two independent devices. In this way, no single device failure can cause the loss of the entire contents of secondary storage.

The storage allocation strategy suggested above is desirable not only for reliability reasons, but also because it permits the file system to make use of detachable storage devices. The physical device or devices assigned to a logical device, i.e. a subtree, can be attached and detached as needed. (1) Hence the assignment of subtrees of the file hierarchy to individual storage devices appears to offer both reliability and economy.

When salvaging finishes, the file system is left in a consistent state. At best, the entire file hierarchy is saved and no reloading is necessary. In less fortunate circumstances, certain files and directories must be recovered from backup tapes. To facilitate this operation, those directories from which entries were deleted have been so marked. Also, files which are potentially inconsistent have been flagged.

Reloading

When the results of salvaging indicate a loss of information from the file hierarchy, a procedure known as reloading is employed to recover the most recent backup copies of all missing files and directories. The reloading program, called the reloader, runs under the standard operating system. Therefore, the supervisor, the reloader,

(1) Note that detachable storage devices used in this fashion will hold directory information assumed to be trustworthy by the file system. Therefore, for security reasons, detachable subtrees must be considered system property and must not be permitted to leave the computer installation.

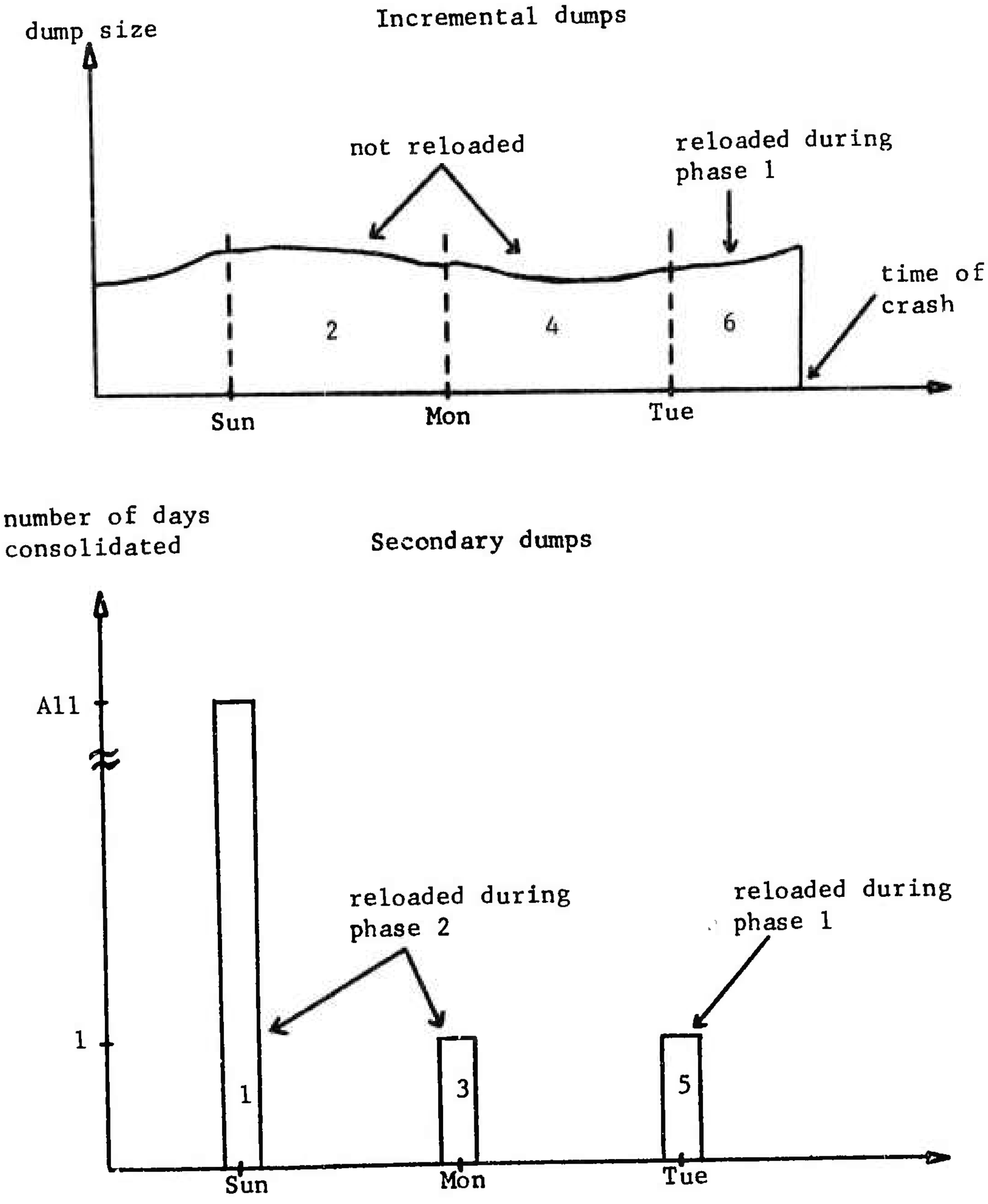
and those commands and subroutines used during reloading must be loaded from a system bootstrap tape before reloading begins. For efficiency reasons, the reloading operation is divided into two phases. During phase 1, all directories and some files are recovered. All remaining files are recovered during phase 2. The system can sometimes be made available to users immediately after salvaging. If not, the system can always be made available by the end of phase 1.

A ledger is maintained by the operators which notes the starting time, completion time, and reel numbers for every incremental and secondary dump. This ledger serves as a reloading guide during phase 1. Beginning with the most recent dump, the dumps are selected in reverse chronological order according to starting time. Each dump is searched from beginning to end for missing files and directories which can be recognized as inferiors of those directories marked by the salvager. Because the dumps are selected in reverse chronological order, the most recent backup copies of files and directories are reloaded first after which older copies can be ignored. A flag is set in the entry of each reloaded file which will cause an error code to be returned on future attempts to initiate a reloaded file. In this way, users are warned of the fact that a backup copy has been reloaded. The identity of a reloaded backup copy can be determined by examining the modification time recorded in the entry.

Phase 1 terminates after the most recent secondary dump has been reloaded. In the example of figure 2.4, the incremental dumps from region 6 plus secondary dump 5 are reloaded during phase 1. Notice that the incremental dumps of regions 2 and 4 are superseded by secondary dumps 3 and 5 and therefore need not be reloaded. In general, at the conclusion of phase 1, no incremental dumps remain to be reloaded. Also, by the end of phase 1, all directories and all system data files have been restored as a consequence of reloading the latest secondary dump. If the latest secondary dump is, in fact, a complete secondary dump, then reloading is finished by the end of phase 1. Otherwise, reloading continues during phase 2.

The purpose of phase 2 is to recover any missing files not yet reloaded. This could be accomplished in the same manner as phase 1, i.e. by searching the remaining secondary dumps in reverse chronological order up to and including the latest complete secondary dump. This searching, however, is time-consuming and inefficient. If, for example, only a small fraction of the hierarchy has been lost, then missing files may be confined to only a few tapes. Notice that since all directories are restored during phase 1, so too are the entries for those files to be recovered during phase 2. Each of these entries contains the tape address of the

Figure 2.4: Example of a reloading situation.



latest secondary backup copy of its associated file. Therefore, the file hierarchy is searched and the tape addresses of all files deserving to be reloaded are extracted. These tape addresses are then sorted into a systematic reloading order and the missing files are located and restored. No time is wasted examining backup copies not eligible for reloading.

By the end of phase 1, all directories and all system data files have been recovered. Therefore, the system can be opened to users at this time. Those users whose files either survived the crash or were reloaded during phase 1 will find the system immediately useful. Other users must await the recovery of additonal files during phase 2. The entries of files to be recovered during phase 2 are specially marked so that premature attempts to initiate these files will generate an appropriate error message. Of course, most users depend upon commands and subroutines in the system library. Therefore, if the system library has been damaged, it is reloaded from the latest special dump of the system library before the system is opened to users.

Often it will happen that salvaging succeeds in saving all but a very few files and minor directories. If all system data bases survive intact, then the system can be made available immediately rather than waiting until the finish of phase 1. The only danger here is that users will be able to modify directories as they are being reloaded. If a user deletes an entry, for example, it may later be restored by the reloader which cannot distinguish between a legitimate deletion made during reloading and a loss caused by a crash. Similarly, if a user inadvertently creates a new entry by the same name as some missing entry, he may prevent the missing entry from being reloaded. These problems are not encountered during phase 2 since the identities of all entries to be reloaded are known by the end of phase 1. Notice, however, that the only sensitive directories are those with missing entries, i.e. those directories marked by the salvager. Hence, one could forbid entry creation, deletion, or renaming in marked directories until they are fully restored. Alternatively, one could simply warn users of the consequences of modifying marked directories. Thus, the system can be made available immediately after salvaging if no system data bases have been destroyed.

The ability to open the system to users during reloading has the obvious advantage of making the system useful again as soon as possible to at least some users. However, these users compete with the reloading operation for system resources causing a reduction in reloading speed and thus further inconveniencing other users awaiting the recovery of files. In the case of a crash which causes only a small amount of damage, it seems reasonable that the system should be made available as soon as possible since only a few users will suffer any delay. However, for more serious crashes, it might be wiser to postpone opening the system to users until a significant amount of reloading has been performed.

To this point we have assumed that only whole files are dumped and reloaded rather than pages of files as postulated earlier. As it turns out, the dumping of pages does not upset the reloading framework already described. If we assume that incremental dumps copy individual pages and secondary dumps copy whole files, no major changes are required. (Directories are always dumped in full.) Phase 1 will simply reconstruct files page-by-page. The tape addresses of all files not recovered by phase 1 will then be determined as before. Phase 2 will finish the job by reloading missing files and missing pages of files which were partially reconstructed during phase 1.

Reloading time will depend upon the size and number of incremental and secondary dumps needed to recover the file hierarchy. The various trade-offs between dumping overhead and recovery time have already been discussed. It is clear that, for the worst of disasters, the work of reloading must increase with the size of secondary storage. However, increases in processing and I/O capacity can be used to counteract the demands of enlarged storage volume. In order to take full advantage of available processing and I/O capacity, parallel processing is employeu both during phase 1 and phase 2 as will be described in chapter 3. Of course, so long as storage capacity grows faster then processing and I/O capacity, reloading time will increase. It should be noted, however, that failures which necessitate reloading the whole of secondary storage are extremely rare. Therefore, much care has been taken to handle the milder, more common failures efficiently. For example, in the case of a single device failure, recovery time will depend on the size of the device, but not on the total size of secondary storage. The ability to open the system to users during reloading can minimize the number of users inconvenienced by a failure.

Retrieving

A procedure known as retrieving is used to recover backup copies of files and directories at the request of users. Retrieving, which is a remedy for isolated system mishaps and for personal user mishaps, is distinguished from reloading, which is a remedy for system-wide failures. The operation of the retrieving program, called the retriever, is quite simple. The retriever is supplied with a pathname and a tape address. The specified tape is mounted and the file or directory named is located and restored to the file hierarchy. The retriever can also be requested to restore an entire subtree. In this case, the pathname of the directory which heads the subtree is specified and all members of the subtree found on the tape are restored to the hierarchy.

In order to request retrievals, a user must have some method for determining when and where backup copies of his files and directories were produced. For this purpose, a dump map is created with every dump. A dump map specifies the name, tape address, modification time, and dumping time of each file and directory included in a particular dump. Of course, it would be extremely tedious to search through a large number of dump maps in order to locate a copy of a given file or directory. Therefore, a user should remember approximately when his file was lost or damaged. Also, if the file was recently modified, the user should know approximately when it was modified. If user-controlled backup was employed, the user should know the exact modification time of the backup copy he seeks. With this information and knowledge of the dumping schedule, a reasonable guess can be made as to which dump contains a suitable copy of the desired file.

It is not intended that users should actually examine printed dump maps directly. This arrangement would be inconvenient and would implicitly prohibit retrievals being requested from remote locations. Also, permitting users to freely browse through dump maps would constitute a security loophole since the names, and hence the existence, of all files and directories would be exposed. Therefore, two other approaches are suggested. The simplest approach would be to give an operator or some other computer center employee responsibility for examining dump maps. Users could then request backup information concerning particular files or directories, or could simply ask for a copy to be retrieved according to modification time or some other criteria. Preferably, however, dump maps could be maintained on-line and users would be permitted to inspect these maps subject to access controls enforced by the system. In this case, all dump map searching could be performed automatically. Of course, this approach involves some overhead in on-line storage costs for the dump maps. In either case, retrieval requests would be submitted via the computer system so that the identity of a user making a request could be verified.

Summary

The backup system presented in this thesis is designed to protect a tree-structured file hierarchy. The files and directories which comprise this file hierarchy are implemented as segments in a segmented, virtual memory. The various activities of the backup system can be divided into two general categories: (1) activities performed in anticipation of mishaps and (2) activities performed in response to mishaps.

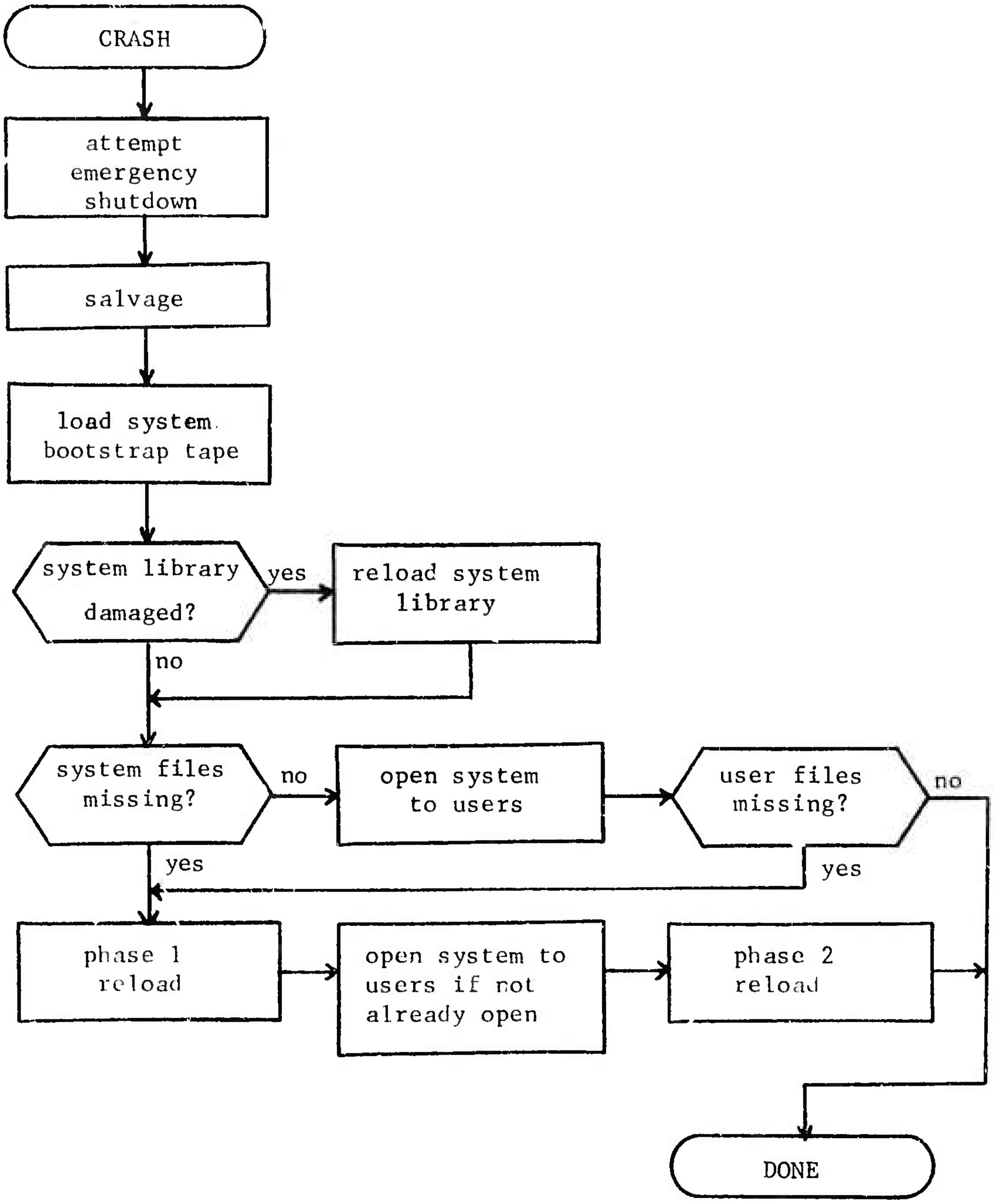
The first category of activities includes both incremental and secondary dumping. Incremental dumping is responsible for keeping the backup system abreast of modifications to the file hierarchy. Periodically, the incremental dumper scans the file hierarchy and copies to tape all newly created or modified files and directories. The interval between the start of incremental dumps defines the resolution of the backup system. In order to limit the time needed to recover the file hierarchy from backup tapes, the information contained on incremental dump tapes must be consolidated periodically. Secondary dumping is the mechanism used to accomplish this consolidation. Two types of secondary dumps are employed by the backup system. A complete secondary dump is used to consolidate all incremental dumps since the beginning of system operation. A partial secondary dump is used to consolidate some or all of those incremental dumps performed since the latest complete secondary dump.

User processes and the dumping processes operate asynchronously. Files eligible for backup are copied at the time they are encountered by the dumper. This mode of operation is satisfactory while files are not subject to modification. Certain applications, however, require control over the production of backup copies while files are subject to modification. This is necessary to guarantee consistency and to record the status of backup copies relative to work in progress. For this purpose, a user-controlled backup mode can be requested for files initiated with write access. Under this scheme, file copies are created at times specified by the user and stored in the file hierarchy where they can be found by the dumper.

The second category of activities performed by the backup system includes salvaging, reloading, and retrieving. Salvaging and reloading are used to recover from system failures whereas retrieving is performed on behalf of users to recover from personal losses. Following a system failure, the integrity of the file system must be questioned. If possible, an emergency shutdown is performed to recover information from main memory. Next, salvaging is performed to verify the internal consistency of directories, reconstruct storage assignment tables, and warn users of potentially inconsistent files. The salvager also marks those directories, if any, from which entries have been lost. Any missing entries are recovered from backup tapes by the reloading procedure. Reloading is divided into two phases. During phase 1, all directories, all system files, and some user files are restored. All remaining files are recovered during phase 2. The system can be opened to users immediately following salvaging if no system files have been lost. Otherwise, the system can be made available by the conclusion of phase 1. The various steps and decisions of the crash recovery procedure are illustrated in the flow chart of figure 2.5. By comparison, the retrieving procedure is fairly simple. A user wishing to retrieve a backup copy of a particular file or directory specifies the pathname and tape address of the backup copy he desires. The appropriate backup tape is mounted and the desired backup copy is extracted and restored to the file hierarchy. The names, tape addresses, modification times, and dumping times of backup copies are listed in dump maps produced with every dump.

A primary objective of the backup system design is to be able to scale up with system growth. Ideally, one might hope that as secondary storage size increases, resolution and recovery time can be held constant while dumping overhead is kept within reasonable limits. The proposed design closely approximates this goal. The cost of incremental dumping is proportional to system processing activity and therefore will remain a constant fraction of system processing time regardless of on-line storage

Figure 2.5: The crash recovery procedurc.



expansion. The same is true of partial secondary dumping. The cost of complete secondary dumping, on the other hand, is proportional to secondary storage size. However, because the resolution of the backup system is upheld by incremental dumping, complete secondary dumps need not be performed frequently. Therefore, total dumping overhead need not rise sharply with secondary storage growth.

Whether or not reloading time will remain constant with on-line storage expansion will depend upon the "balance" of system growth. If processing and I/O capacity increase at the same pace as on-line storage size, reloading time need not increase. However, if secondary storage expansion outstrips available processing capacity or tape I/O capacity, then reloading time will increase. Of course, a system administrator always has the option of adding additional tape dirives and I/O channels to improve reloading speed through parallel processing.

Recovery time should not be measured solely in terms of the time required to completely reload secondary storage. System failures which destroy all of on-line storage are extremely rare. Therefore, much effort has been devoted to recovering from the milder, more common failures quickly and efficiently. Files and directories which can be salvaged following a system failure need not be reloaded. Most system crashes, of course, will require no reloading whatsoever. More serious failures may necessitate the reloading of a single device or a number of scattered files and directories. In most such cases, the majority of users will be able to resume work as soon as salvaging has completed. In this sense, recovery time from all but the most catastrophic of system failures need not increase with on-line storage size.

CHAPTER 3

PROGRAM STRUCTURE OF THE BACKUP SYSTEM

The backup system is composed of four basic program modules: the dumper (which performs both incremental and secondary dumping), the salvager, the reloader, and the retriever. In this chapter, the operations of the dumper, reloader, and retriever are examined in detail. Due to its predominantly ad hoc character and its intimate involvement with peculiarities of file system structure, the operation of the salvager is not further explored in this thesis. The salvager/reloader interface, however, is described in this chapter.

The Dumper

The dumper program is designed to scan portions of the file hierarchy in search of files and directories eligible for dumping. Each pass over the hierarchy is guided by a control file in which are specified the pathnames of directories, i.e. subtrees, to be searched. If, for example, it was desired to divide complete secondary dumping of the hierarchy among four separate dumps, then four different control files could be used to select four different portions of the hierarchy. As mentioned in chapter 2, the system library is ignored by standard incremental and secondary dumps. Hence, the pathname "root>system_library" would not appear in the control files for such dumps. However, when special dumps of the system library are performed, the control file will contain only the pathname "root>system_library".

The eligibility of a file or directory for incremental dumping depends upon its time of last modification. The file system maintains in every entry the date/time modified (dtm) of its associated file or directory. Also, the backup system maintains in every entry a date/time dumped (dtd). Hence, a file or directory is eligible for incremental dumping whenever dtm>dtd, i.e. whenever the file or directory has been modified more recently than it has been dumped. Immediately before a file or directory is dumped, the current time is read from the system clock. Immediately after a file or directory is dumped, its dtd is reset to the prior clock reading. This procedure guarantees that any modifications occurring while a file or directory is being dumped will qualify that file or directory for the next incremental dump.

It would be extremely time-consuming and inefficient if the incremental dumper were required to search the entire hierarchy for modified files and directories on every pass. The hierarchy might contain thousands of directories, whereas only a comparatively small number of directories are likely to be in use during any single incremental dump interval. To avoid this excessive searching, dtm and dtd for a directory are interpreted to refer to the subtree rooted by the directory. In other words, dtm gives the time that a directory or any of its inferiors were last modified. Similarly, dtd gives the time that a directory or any of its inferiors were last incrementally dumped. Therefore, if dtm>dtd for a directory, then neither the directory itself nor any of its inferiors have been modified more recently than they were last dumped. Consequently, if dtm>dtd for a directory, the dumper need not search the subtree below that directory. Because searching time is minimized in this way, the unit cost of incrementally dumping a file will not increase with the size of the hierarchy.

For reloading purposes, it is desirable that a file or directory contained in a backup dump be preceded by all of its superior directories. Since secondary dumps include all directories and since the dumper searches and dumps the hierarchy in top-down order, every file and directory must be preceded by all of its superior directories. This same property has also been incorporated in incremental dumps as a by-product of the special interpretation of dtm and dtd for directories. Whenever a file or directory is modified, all of its superior directories also appear to be modified.

Consequently, any file or directory contained in an incremental dump must be preceded by all of its superior directories. Special care must be taken to dump the superior directories of those subtrees named in the control file. The incremental dumping algorithm is summarized in the flow charts of figures 3.1 and 3.2.

The eligibility of a file for secondary dumping depends entirely on its incremental dumping history. As explained in chapter 2, a secondary dump is used to consolidate some number of preceding incremental dumps. Therefore, for each secondary dump, one must indicate exactly how many preceding incremental dumps are to be consolidated. This is done by specifying a time called "timel". All incremental dumps which began after timel and before the current secondary dump will be consolidated. For complete secondary dumps, timel is set equal to zero, i.e. the beginning of system operation. For partial secondary dumps, timel is set equal to the starting time of some previous secondary dump. For simplicity in secondary dumping and reloading, we require that no incremental dump be in progress at the time a secondary dump is started. Therefore, a secondary dump can be understood to supersede some whole number of immediately preceding incremental and secondary dumps.

The problem of determining whether or not a file is eligible for secondary dumping is somewhat more subtle than

Figure 3.1: The dumper algorithm (part 1).

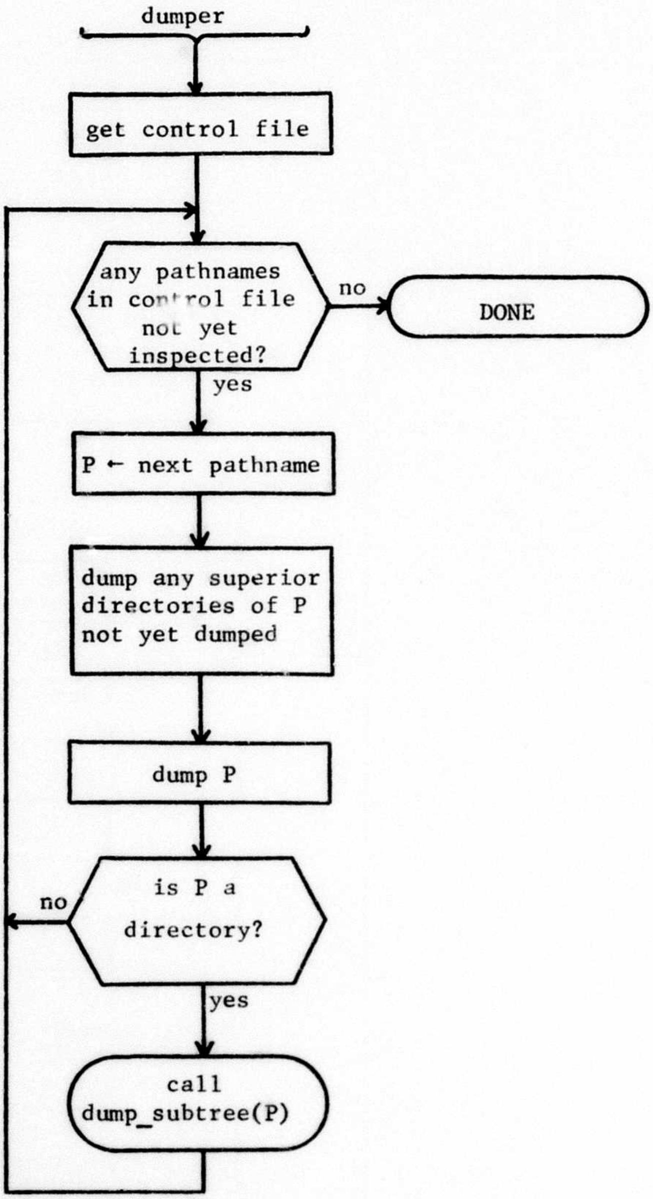
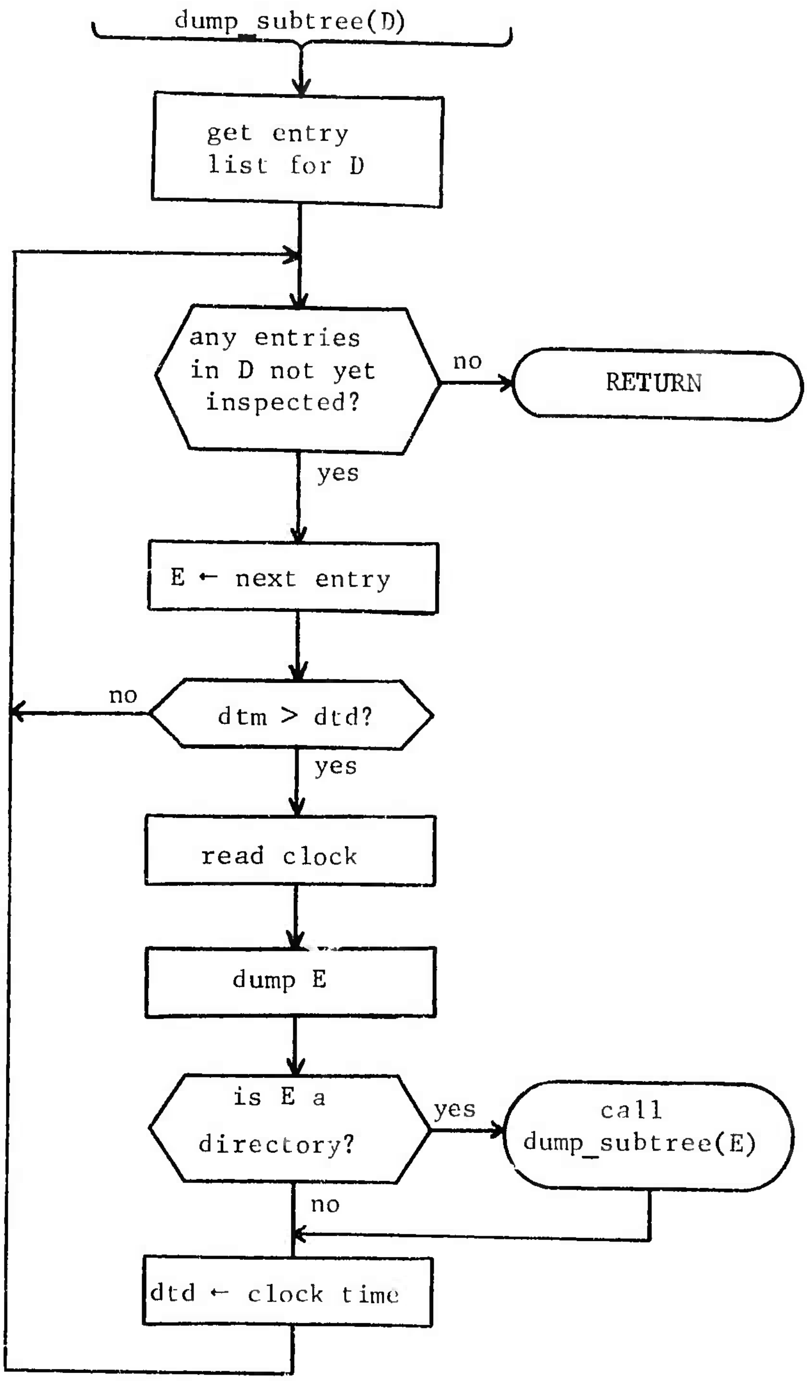

dumper
get control file
any pathnames in control file not yet inspected?
no
DONE
yes
P ← next pathname
dump any superior directories of P not yet dumped
dump P
is P a directory?
no
yes
call dump_subtree(P)

Figure 3.2: The incremental dumper algorithm (part 2).



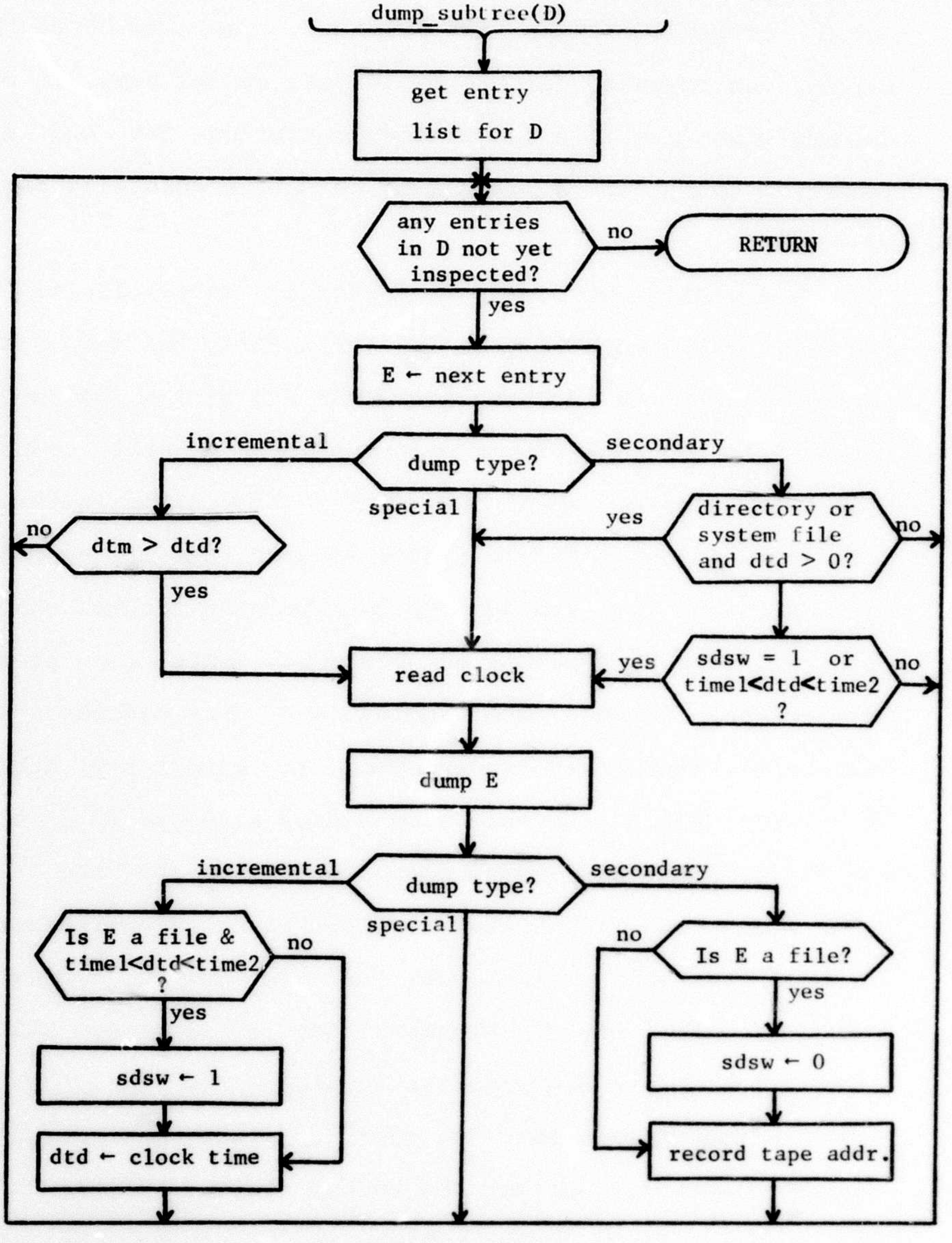
it appears. One might, for example, declare a file eligible for secondary dumping whenever dtd>time1. Unfortunately, this test causes redundant secondary dumping of files. The trouble is that because a secondary dump takes a substantial amount of time to run, it would not only consolidate preceding incremental dumps, but also portions of concurrent incremental dumps. Therefore, the incremental dumps consolidated by consecutive secondary dumps would always overlap to some extent.

In view of this problem, the dumper records its starting time as "time2" at the beginning of each secondary dump. A file can then be declared eligible for secondary dumping whenever time1<dtd<time2. Unfortunately, if a file is incrementally dumped shortly after time2, the secondary dumper cannot determine whether or not it was also dumped between time1 and time2. Of course, in the event of a system failure, a file incrementally dumped after time2 will be reloaded from an incremental tape. (Remember that the latest incremental dumps are reloaded before the latest secondary dump.) Therefore, for reloading purposes alone, the secondary dumper could skip files incrementally dumped later than time2. However, we have assumed that incremental tapes are not retained as long as secondary tapes. Thus, a version of a file omitted from a secondary dump, but included in a concurrent incremental dump, would not survive

in backup storage as long as contemporary versions of files included in the secondary dump. Hence, we must reject this solution.

The difficulty with the above approach is that once a file has been incrementally dumped after time2, the secondary dumper can no longer determine if that file was also dumped between time1 and time2. Therefore, while a secondary dump is in progress, the incremental dumper must not reset dtd for a file without first testing if that file is eligible for secondary dumping. If the file is eligible for the concurrent secondary dump, then the incremental dumper will turn on a secondary dump switch (sdsw) in the entry of the file. Hence, a file is eligible for secondary dumping whenever sdsw=1 or time1<dtd<time2. Remember, too, that all directories and system files which have ever been incrementally dumped are also eligible for secondary dumping. For simplicity, we assume that system files can be identified by a special indicator kept in each entry. The combined incremental and secondary dumping algorithm, part 2, is shown in figure 3.3. Also shown is the special dumping algorithm used for the system library which simply dumps entire subtrees. (Part 1 of the dumping algorithm, shown in figure 3.1, is identical for all types of dumps.) Note that the incremental dumper may, at times, turn on the sdsw for a file after that file has been encountered by a

Figure 3.3: The incremental and secondary dumper algorithm (part 2).



concurrent secondary dump. This means that the file will remain eligible for the next secondary dump. Although this was not the intended purpose of turning on the sdsw, it does no harm since the file has already qualified for the next secondary dump as a result of being incrementally dumped after time2.

User-controlled backup is basically invisible to the dumper. If a shadow copy exists for a file, the file system insures that the dumper sees the dtm of the shadow copy rather than the possibly more current dtm of the original file. Similarly, when the dumper attempts to initiate a file for dumping, the file system insures that if a shadow copy exists, the shadow copy is initiated rather than the original copy. Precautions must be taken to prevent interference between the dumper and user processes when user-controlled backup is employed. The dumper may attempt to dump a file at the same time a user attempts to initiate the file, update its shadow copy, or terminate the file. Once again, the file system coordinates these operations in a manner unseen by the user or the dumper. Specific solutions to these problems for a proposed Multics implementation are discussed in chapter 4.

The use of pathnames to identify file and directory copies causes problems for the backup system. Pathnames are composed of entry names which are, of course, subject to change. If a pathname is recorded on tape with a backup copy, this pathname may no longer be valid at some later time when an attempt is made to reload the backup copy. Hence, the reloader may mistakenly decide that a file has been deleted and should not be reloaded when, in fact, all that has really happenned is that the file has been renamed. To a lesser degree, the use of pathnames also causes similar trouble for the dumper. Between the time that the dumper obtains the entry list for a directory and the time it finishes dumping all eligible entries in that directory, a user may rename one or more of those entries. It will appear to the dumper that these renamed entries have been deleted.

Clearly, what is needed to resolve this naming problem is some means of identification not subject to change. For this purpose, a unique identifier (uid) is assigned to each entry at its time of creation. The file system guarantees that this uid will never again be assigned to another entry in the same directory even after the original entry has been deleted. In the same way that pathnames are constructed from entry names, "pathuids" are constructed from entry uids. A pathuid is guaranteed to uniquely identify a particular entry for the lifetime of the system.

Pathuids can be viewed simply as numbers. Let us assume that each uid is represented by a single machine word. Furthermore, assume that the maximum depth of the hierarchy is restricted to sixteen levels. Then, a pathuid can be represented by sixteen machine words which, together, can be treated as a single binary number. The uid of the root directory is always the high-order word of a pathuid. The uid of some immediate inferior of the root directory is always the second word of a pathuid, and so on for succeeding words. If the entry described by a pathuid is less than sixteen levels deep in the hierarchy, the low-order words of its pathuid are padded with zeroes.

The idea of treating pathuids as numbers is employed during phase 1 reloading to synchronize parallel reloading processes as will be explained later. For this purpose, it is necessary that all files and directories appear in numerical pathuid order within a backup dump. This implies that the treewalk performed by the dumper should be organized so as to encounter files and directories in pathuid order. Therefore, for each directory searched, the entry list must first be sorted into uid order. This procedure guarantees that within a directory, all entries dumped will appear in uid order on the backup tapes. Furthermore, since this procedure is applied recursively to successively inferior directories, it follows that within a subtree, all entries dumped will appear in pathuid order. To insure that different subtrees appear in pathuid order, the pathnames listed in the control file are converted to pathuids and sorted into numerical order at the beginning of each dump. Hence, all entries in the entire dump must appear on the dump tapes in pathuid order as desired.

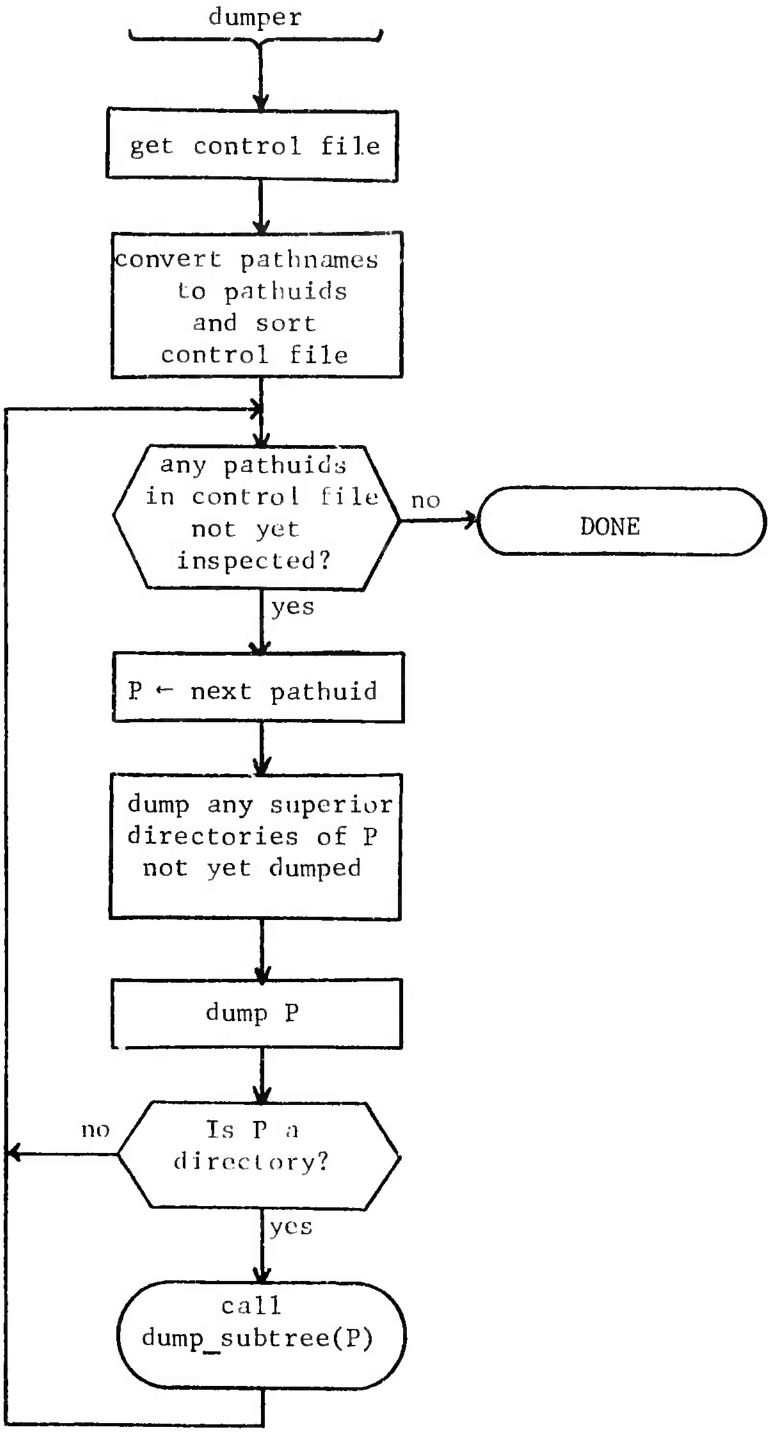
The sorting of entry lists, if performed for a large number of directories on every dump, can be expensive. Therefore, one might specifically design the file system to thread entry lists in uid order so that no sorting will be necessary. We will hereafter assume that entry lists are threaded in uid order. Hence, the only change to the dumper algorithm needed to dump files and directories in pathuid order is the sorting of the control file. This change to part 1 of the dumper algorithm is shown in figure 3.4.

The dump record written on a backup tape for a file or directory includes a preamble followed by the contents of the file or directory. The preamble section includes a pathuid, pathname, time dumped, and the necessary information to reconstruct the entry of the file or directory plus all superior entries. The reasons for including all superior entries in each preamble will be explained in the ensuing discussions of reloading and retrieving.

The Salvager

Unlike the other backup system components, the salvager

Figure 3.4: The dumper algorithm (part 1, revised)


dumper
get control file
convert pathnames to pathuids and sort control file
any pathuids in control file not yet inspected?
no
DONE
yes
P ← next pathuid
dump any superior directories of P not yet dumped
dump P
Is P a directory?
no
yes
call dump_subtree(P)

requires knowledge of internal file system structure. In fact, the specific nature of the salvaging operation is determined by the peculiarities of this internal structure. The task of attempting to generalize or, at least, to systematize the salvager design is not undertaken in this thesis. Therefore, we will not be concerned so much with the method of salvaging, but rather with its results. The overview of salvaging contained in chapter 2 basically describes the current Multics salvager with a few added enhancements. In chapter 4, minor modifications to the Multics salvager are described which would make it compatible with the proposed backup system.

When salvaging concludes, the directory structure of the file hierarchy and the storage assignment tables have been put in a consistent state. Those files which were not properly terminated at the time of the crash have been flagged as potentially inconsistent. Those directories in the hierarchy from which entries were lost have been appropriately marked. Each entry which describes a directory contains a missing entries switch (mesw) for this purpose. Also, to improve reloading efficiency, the salvager labels the superior entries of those entries in which it has turned on the mesw. Each entry contains an inferior reload switch (irsw) for this purpose. When turned on by the salvager, the irsw indicates the existence of an inferior entry having its mesw turned on. However, the irsw is also used for other purposes by the reloader as descri ›ed in the next section.

The Reloader

The phase 1 reloader is designed to read through backup tapes in search of files and directories eligible for reloading. As described in chapter 2, backup dumps are selected for reloading in reverse chronological order sc that the most recent copies of files and directories will be encountered first after which older copies can be ignored. Each dump is scanned from beginning to end so as to retrace the treewalk originally performed by the dumper. This procedure insures that the reloader will never encounter a file or directory before first encountering all of its superior directories (except in the unusual case of a tape reading error described later).

The only files and directories eligible for reloading are missing inferiors of those directories for which the salvager has turned on the mesw. If the reloader finds on tape a file which already exists in the hierarchy, then either that file was successfully salvaged or else the most recent backup copy has already been reloaded. The same is true of directories except when the mesw is enabled. In this case, missing entries should be restored to the directory. Whenever a directory does not deserve to be reloaded, the reloader knows immediately that none of its inferiors deserve to be reloaded unless the irsw for that directory is turned on.

The first step in the hierarchy restoration process occurs when the reloader finds on tape a copy of a directory for which the mesw has been turned on. At that time, all entries in the backup copy of the directory not already in the hierarchy are restored. In each such entry, a reload pending switch (rpsw) is turned on which prevents the entry from being initiated by a user. The mesw can now be turned off since all entries have been restored. However, the irsw is turned on to indicate that inferior files and directories must be reloaded.

After missing entries have been restored to a directory, the reloader proceeds to recover the files and directories belonging to those entries. When the reloader finds on tape a file for which the rpsw has been turned on, the file is reloaded and its rpsw is turned off. Also, a reload flag is set in the entry of the file to warn users that a backup copy of the file has been recovered. Similarly, when the reloader finds on tape a directory for which the rpsw has been turned on, the directory is reloaded and its rpsw is turned off. To insure that the inferiors of this directory are recovered, the rpsw is turned on in each entry contained in the directory and the irsw for the directory itself is turned on. In this way, all missing files and directories encountered during phase 1 are restored to the hierarchy.

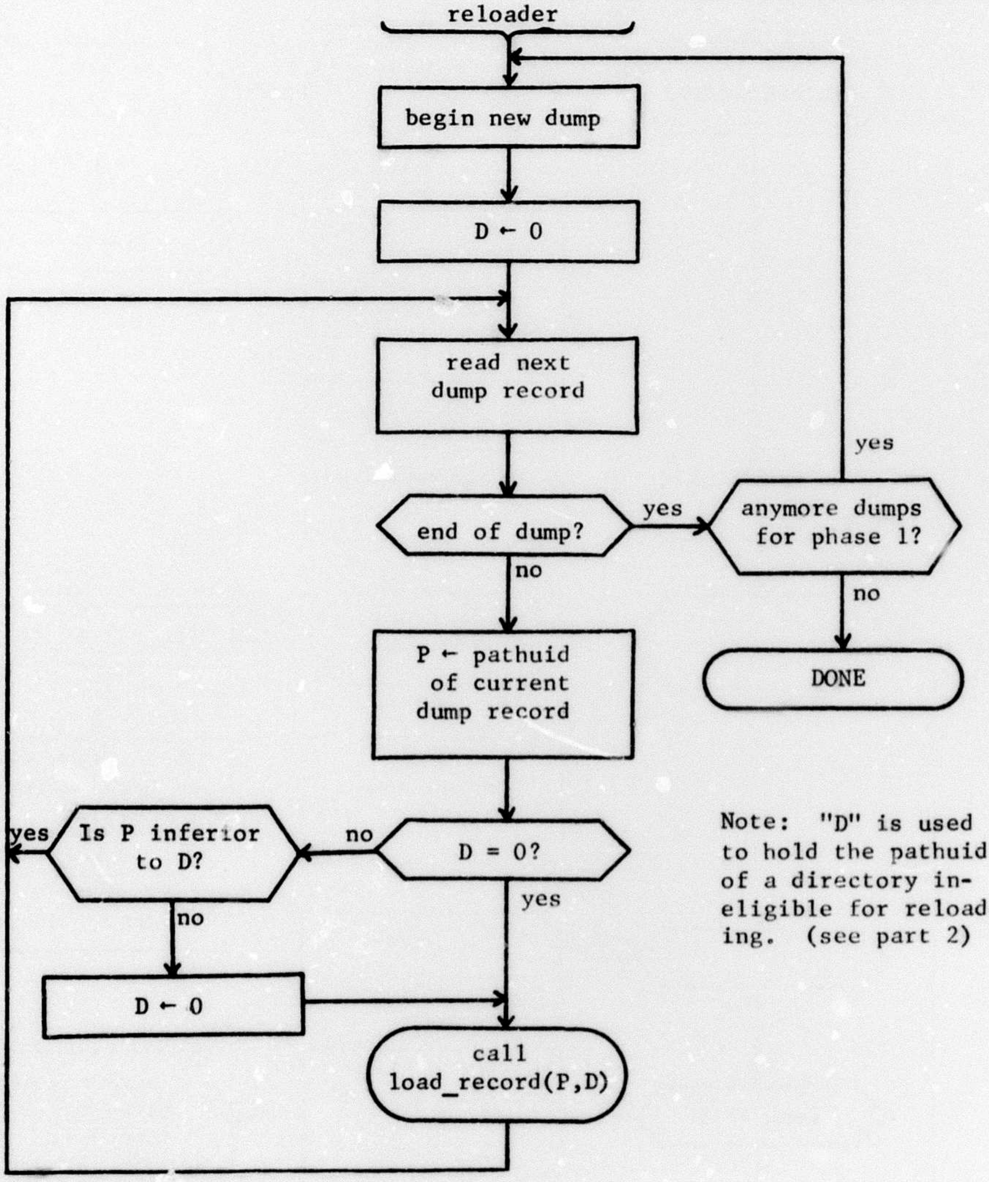
Notice that the entry for a file or directory is restored at the time its parent directory is reloaded. Therefore, if the entry for a file or directory does not exist in the hierarchy at the time the reloader encounters a backup copy of that file or directory on tape, this indicates that the entry has been legitimately deleted and should not be reloaded. Of course, when the reloader determines that a directory has been deleted, it knows that all inferiors of that directory have also been deleted and should not be reloaded.

There is one situation which contradicts the above assumption regarding deletions. When a tape reading error occurs causing backup records to be skipped, the reloader may encounter a file or directory before some of its parent directories have been restored. The reloader should not bypass readable files and directories simply because their superiors could not be read. This would only increase the severity of the error. Hence, the reloader must be able to distinguish between legitimately deleted entries and entries not restored due to tape errors. Fortunately, this distinction can be made quite easily. In the absence of tape errors, the superiors of a file or directory will always precede it in a dump. Therefore, at the time a file or directory is read from tape, it should be true that none of it superior directories will have the mesw or rpsw turned on. If such a superior directory does exist, then a tape error (or perhaps some other error) must have prevented intervening directories from being reloaded. Therefore, the reloader must fabricate these intervening directories in order to reload the file or directory in hand. The entry for this file or directory plus the entries of all superior directories are contained in the preamble section of the current dump record. Hence, these entries can be restored immediately. Other entries, however, cannot be restored to the fabricated directories. Therefore, in the entry for each fabricated directory the reloader turns on a fabrication switch (fabsw) to indicate that these directories have not been completely reloaded. If a copy of a fabricated directory is later encountered on an older dump during phase 1, missing entries and non-entry items will be restored and the fabsw will be turned off.

One last concern of the phase 1 reloader is to make certain that for each reloaded file and directory, the dtd, sdsw, and secondary dump tape address are properly reset. Except in the case of tape errors, the entry for a file or directory is restored at the time its parent directory is reloaded. At the time a directory is dumped, however, its inferiors have yet to be dumped. Hence, the dtd, sdsw, and tape address for an entry may change after its containing directory has been dumped if the entry itself is dumped. Therefore, if a file or directory is reloaded from the same dump as its parent directory, the latest backup information must be restored when the file or directory itself is encountered. When reloading an incremental dump record, the dtd and sdsw must be updated. When reloading a secondary dump record, the sdsw and tape address must be updated. The phase 1 reloading algorithm is summarized in the flow charts of figures 3.5 and 3.6.

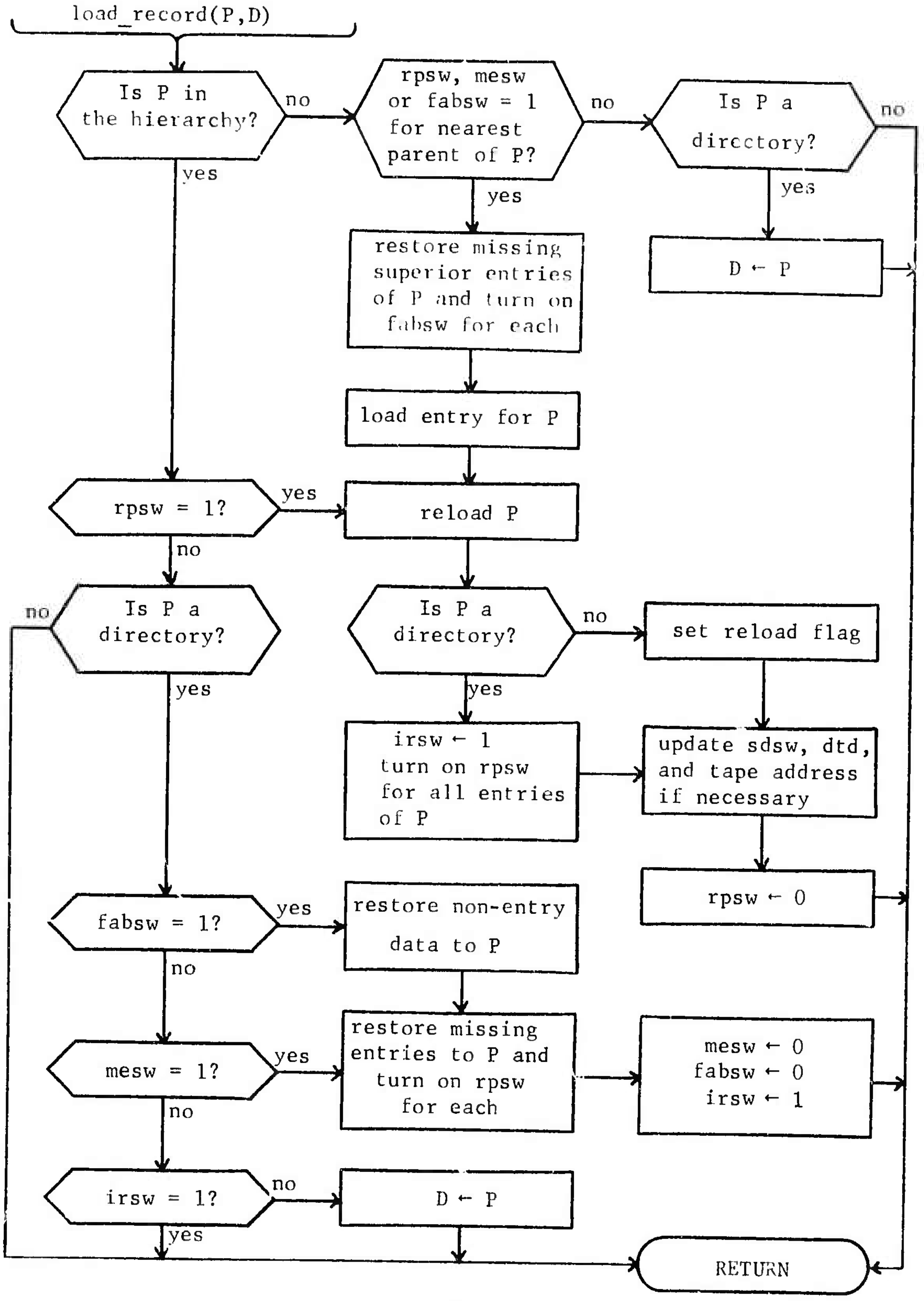
As mentioned earlier, multiple reloading processes can be employed during phase 1 to reload several dumps concurrently. One rule of the phase 1 reloading algorithm is that the latest backup copies of files and directories must be encountered first. Hence, if several dumps are to be reloaded at once, some mechanism must be employed to insure that this rule is obeyed. As explained earlier, files and directories always appear in pathuid order within a dump. A pathuid uniquely and permanently identifies the tree position of a file or directory within the hierarchy relative to all other files and directories. As the reloader scans a dump from beginning to end, it essentially retraces the treewalk originally performed by the dumper.

Figure 3.5: The phase 1 reloader algorithm (part 1).



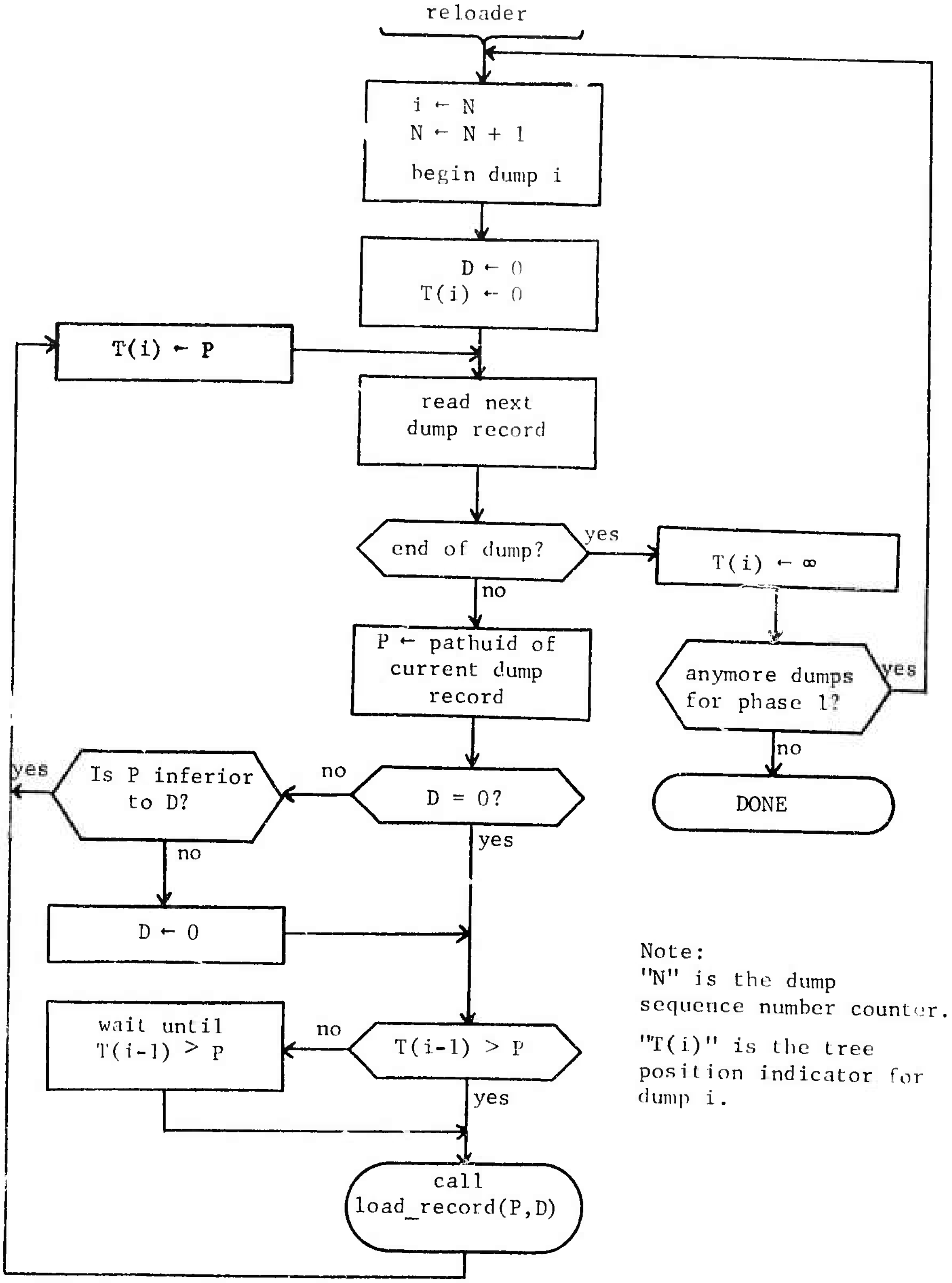
Note: "D" is used to hold the pathuid of a directory ineligible for reloading. (see part 2)

Figure 3.6: The phase 1 reloader algorithm (part 2).


load_record(P,D)
Is P in the hierarchy?
no
yes
rpsw, mesw or fabsw = 1 for nearest parent of P?
no
yes
Is P a directory?
no
yes
D ← P
restore missing superior entries of P and turn on fabsw for each
load entry for P
rpsw = 1?
yes
no
reload P
Is P a directory?
no
yes
Is P a directory?
no
yes
set reload flag
irsw ← 1 turn on rpsw for all entries of P
update sdsw, dtd, and tape address if necessary
rpsw ← 0
fabsw = 1?
yes
no
restore non-entry data to P
mesw = 1?
yes
no
restore missing entries to P and turn on rpsw for each
mesw ← 0
fabsw ← 0
irsw ← 1
irsw = 1?
no
yes
D ← P
RETURN

Therefore, parallel reloading processes can be synchronized according to tree position. A reloader scanning a particular dump must stay behind (in terms of tree position) all reloaders scanning more recent dumps. In order to distinguish the relative ages of dumps, each dump reloaded during phase 1 is assigned a sequence number. The most recent dump is assigned sequence number one. The second most recent dump is assigned sequence number two, and so on for older dumps. A counter accessible to all reloading processes gives the sequence number of the next dump to be reloaded. Initially, of course, this counter is set to one. When a reloading process begins a new dump, it takes the current sequence number from the counter and then increments the counter by one. For each dump to be reloaded during phase 1 there exists a tree position indicator accessible to all reloading processes. The contents of a tree position indicator is a pathuid. As a reloading process scans through dump *i*, it repeatedly updates the tree position indicator for dump *i* with the pathuid of the file or directory it has last scanned. A reloading process working on dump *i* is forbidden to pass the tree position of the reloading process working on dump *i*-1. The synchronization discipline of the phase 1 reloading algorithm, part 1, is summarized in the flow chart of figure 3.7. Part 2 of the phase 1 reloading algorithm is unchanged.

Figure 3.7: The phase 1 reloader algorithm (part 1, revised).



Note:
"N" is the dump sequence number counter.

"T(i)" is the tree position indicator for dump i.

Phase 1 normally ends after the latest secondary dump has been reloaded. At this point, all directories should be restored. If, however, any portion of the latest secondary dump is unreadable, certain directories may not be restored. If a large number of directories could not be recovered, phase 1 can be extended to include older secondary dumps until all directories have been reloaded. If only a small number of directories were missed due to tape errors, these directories can be separately retrieved during or after phase 2 reloading.

At the conclusion of phase 1, the entries of all files to be reloaded during phase 2 can be found in the hierarchy. In each of these entries, the rpsw is turned on. Remember that every entry in the hierarchy contains the tape address of the latest secondary backup copy of its associated file or directory. Hence, the hierarchy is searched and the tape addresses of all files deserving to be reloaded during phase 2 are extracted into a list. Note that it is not necessary to search all directories in the hierarchy for entries having the rpsw turned on. For any such entry, the irsw must be turned on in all superior entries. Hence, entries having the rpsw turned on can be located with a minimum of searching. At the same time it may also be useful, for diagnostic purposes, to produce a reload map, i.e. a listing of all files and directories reloaded during phase 1 or eligible for reloading during phase 2. As described previously, the rpsw, mesw, and fabsw are turned off for those files and directories reloaded during phase 1. Instead of turning these switches off, the phase 1 reloader will set them to a "standby" position so that files and directories reloaded during phase 1 can be recognized and hence a reload map can later be generated.

Now begins phase 2 which is entirely automatic. The list of tape addresses compiled during the hierarchy search is sorted by tape reel number and record number. This sorting groups together tape addresses having identical reel numbers, and within each such group, the record numbers appear in numerical order. Each group of tape addresses having a common reel number can be interpreted as a reloading guide which identifies the locations of files to be reloaded on a given reel. As many reloading processes as possible (or as is optimal) are created, each being assigned a tape drive. Each reloader selects a reloading guide for a particular reel and requests that the corresponding reel be mounted. It then skips through the tape, stopping at the record numbers found in the reloading guide, and reloading the files found at each stop. Notice that no interprocess synchronization is required during phase 2. Reloading ends when all reels containing missing files have been processed.

The Retriever

The retriever program is designed to restore specified files and directories to the hierarchy from backup tapes. The pathnames of files and directories to be retrieved are listed in a control file accompanied by the tape addresses of the desired backup copies. Hence, in order to retrieve a particular backup copy of a file or directory, the retriever merely mounts the specified tape reel, skips to the specified location, and reads the backup record found at that location. The pathnames and tape addresses included in a retrieval control file are determined by examining dump maps as described in chapter 2.

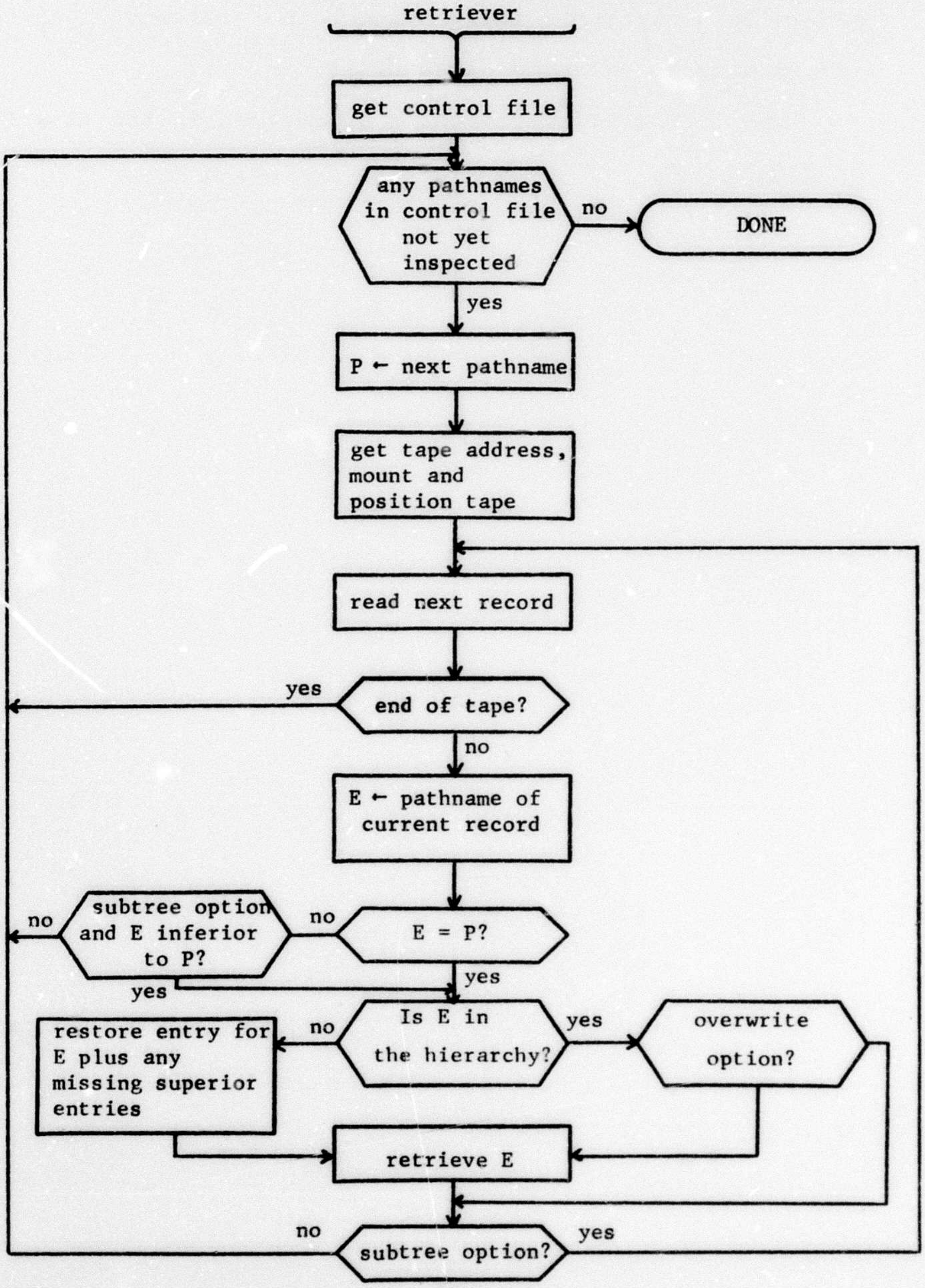
For each directory pathname included in a retrieval control file, a "subtree" option may specified. The subtree option indicates that not only should the directory itself be retrieved, but also all inferiors of that directory on the same tape. The dumping algorithm insures that all members of a subtree must appear contiguously within a dump. Hence, beginning with the head directory of the subtree, the retriever simply reads each backup record in sequence until reaching the end of the subtree or the end of the tape.

The retrieving of a subtree is handled quite differently from the reloading of a subtree. At the time a directory is retrieved, only the non-entry information is restored. The entries contained in the directory are not restored at this time since not all of the inferiors of the directory will necessarily be retrieved. For example, if a subtree is retrieved from an incremental tape, many inferior files and directories of the subtree may not be retrieved because they were not included in the incremental dump. Therefore, the entry for a file or directory is not restored at the time its parent directory is retrieved, but rather at the time the file or directory itself is retrieved. A user may request that a file, directory, or subtree be retrieved for which some superior directories no longer exist. In this case, the superior directories are simply created with default attributes. The entries for these superior directories can be restored from the preamble portion of the backup record being retrieved.

Often a user may wish to retrieve a subtree, some part of which still exists in the hierarchy. In this situation, the user normally will not want existing files and directories to be overwritten by backup copies. He will want only those files and directories which do not currently exist in the hierarchy to be retrieved. Hence, if a file or directory encountered on a backup tape still exists in the hierarchy, it is not retrieved as a matter of standard procedure. In certain circumstances, however, a user may wish to overwrite an existing file or to restore non-entry information to an existing directory. Therefore, for each pathname in the retrieval control file, an "overwrite" option may be specified which will cause the retriever to overwrite existing files and to restore non-entry information to existing directories.

The retriever algorithm is summarized in the flow chart of figure 3.8.

Figure 3.8: The retriever algorithm.


retriever
get control file
any pathnames in control file not yet inspected
no
DONE
yes
P ← next pathname
get tape address, mount and position tape
read next record
yes
end of tape?
no
E ← pathname of current record
no
subtree option and E inferior to P?
no
E = P?
yes
yes
no
Is E in the hierarchy?
yes
overwrite option?
restore entry for E plus any missing superior entries
retrieve E
no
subtree option?
yes

CHAPTER 4

A MULTICS IMPLEMENTATION PLAN

As described in chapter 1, the backup system design presented in this thesis represents a continuing evolution of earlier file system backup facilities implemented on CTSS and Multics. The success of these earlier mechanisms in augmenting file system reliability has already demonstrated the usefulness of the general design approach. The inability of the current Multics backup system to scale up with system growth, however, has been the principal motivation for seeking improvements. The recent movement of Multics to a new hardware base has substantially increased system work capacity. As a result, much concern has developed regarding file system reliability in general, and backup system overhead and performance in particular.

One objective of the work of this thesis has been to produce a practical design which could be implemented on the Multics system and perhaps, with appropriate translation, on other large-scale time-sharing systems as well. In this chapter the job of implementing the backup system defined in chapters 2 and 3 within Multics is examined. This chapter is intended primarily for readers familiar with the Multics system.

The present Multics backup system contains dumper, reloader, retriever, and salvager modules. The Multics salvager will require only minor modifications for use with the new backup system. The present dumper, reloader, and retriever differ substantially, however, from their proposed counterparts and hence must be discarded. New dumper, reloader, and retriever programs must be implemented according to the specifications of chapter 3.

A number of changes and additions to the Multics file system will be required to support the new backup system. The file system is not currently equipped to deal with pathuids, but this capability can be added in a straightforward manner. Several new backup-related items of information must be added to each entry in the Multics file hierarchy, some of which require special interpretation by the file system during reloading. Also, a new file system interface is needed for the dumper and reloader. A number of file system enhancements will be required to provide the user-controlled backup option described in chapter 2.

In summary, the implementation tasks for the new backup system include:

1. new dumper, reloader, and retriever programs
2. handling pathuids within the file system
3. new directory items and related changes
4. new backup system interface to the file system
5. salvager modifications

6. user-controlled backup

The dumper, reloader, and retriever programs have already been carefully described in chapter 3 and hence will not be reviewed in this chapter. Each of the other tasks is separately considered here. At the conclusion of the chapter, an integrated plan is suggested for the implementation of the backup system on Multics.

Handling Pathuids Within the File System

As discussed in chapter 3, pathuids serve two purposes. First, they permit the dumper and the reloader to be unaffected by entry name changes. Second, they permit the synchronization of parallel reloading processes during phase 1 reloading. Each entry in the Multics file hierarchy is assigned a uid. However, these uids cannot be used to name entries in any way understood by the file system.

The ability to accept pathuids in place of pathnames can be added to the Multics file system in a straightforward manner. A file system module known as "find_" is responsible for searching the hierarchy for entries specified by pathname. A new entry point could easily be added to this module to handle pathuids. The basic searching logic of the find_ program would be the same for pathuids as for pathnames. However, the pathname lookup procedure makes use of entry name hash tables kept in each directory. Therefore, in order to achieve equal efficiency in the use of pathuids, uid hash tables must be added to each directory.

The addition of uid hash tables to directories will require a non-trivial amount of work and, once implemented, will slightly increase file system overhead. Therefore, one must question the necessity of such a change. The present Multics dumper and reloader operate without the benefit of pathuids. Unfortunately, however, they do not operate correctly. Both the dumper and the reloader are susceptible to confusion caused by entry name changes. For example, if the name of an entry being examined by the dumper is changed, or if the name of any superior entry is changed, it will appear to the dumper that the renamed entry has been deleted. This implies that the renamed entry and its inferiors, if any, will be skipped by the dumper. A more serious error can occur if a name is moved from one entry to another entry in the same directory. In this case, the dumper may actually dump the wrong file or directory due to the name switch.

Although the present dumper is oblivious to its own problems, it is concerned about the problems of reloading. In an effort to insulate the reloader from the effects of entry name changes, the dumper incrementally dumps every file and directory having a modified entry. The reasoning behind this strategy is that a modified entry might imply a modified entry name. Therefore, anytime any entry attribute is modified, the associated file or directory is dumped. In addition to being grossly inefficient, this strategy also fails to make reloading work properly. Whenever a directory name is changed, the pathnames of all inferiors are also changed. Hence, the present reloader will occasionally fail to reload a directory or, worse yet, will combine two different directories. To rectify this situation, it would be necessary to dump the entire subtree of a directory each time the directory was renamed. (And even this heavy-handed measure would not succeed if a crash occurred during a dump.)

All of the above problems are eliminated by the use of pathuids. Also, user logins during reloading are facilitated by the use of pathuids. Hence, the implementation of pathuids within the Multics file system seems warranted.

New Directory Items and Related Changes

Several new items of backup information must be added to each entry in the Multics file hierarchy. Included in this group are the secondary dump switch (sdsw), the secondary dump tape address, the missing entries switch (mesw), the inferior reload switch (irsw), the fabrication switch (fabsw), and the reload pending switch (rpsw). In order to permit user logins during reloading, the file system must enforce certain restrictions associated with the reload-related switches. First, the file system must be modified to forbid the initiation of files and directories for which the rpsw is enabled. An appropriate error message must be returned to the user. Second, whenever a request is made during phase 1 reloading to add an entry name to a directory or to delete an entry from a directory for which the mesw or the fabsw is enabled, the user must be warned that his request may affect the reloading of the directory as explained in chapter 2.

A New Backup System Interface to the File System

A number of new file system entry points must be provided exclusively for the use of the backup system. These new file system "primitives" are described below. Note that all of these primitives use pathuids to communicate with the file system.

Four file system primitives are needed by the dumper:

1. get_directory

   This procedure is called by the dumper to obtain a copy of a specified directory. Naturally, a directory copy is produced under the protection of the directory lock in order to insure consistency.

2. uid_initiate

   This procedure is called by the dumper to initiate a file specified by pathuid.

3. set_dtd

   This procedure is called by the dumper when performing an incremental dump to reset the dtd for a specified entry. Also, the sdsw for the entry may optionally be turned on.

4. set_tape_addr

   This procedure is called by the dumper when performing a secondary dump to reset the secondary dump tape address for a specified entry. Also, the sdsw for the entry is turned off.

Nine file system primitives are needed by the reloader:

1. reload_status

   This procedure is called by the reloader to obtain the rpsw, irsw, mesw, and fabsw settings for a specified entry. If the specified entry does not exist, the same information is returned for the closest superior entry which does exist.

2. load_directory

This procedure is called by the reloader to restore an entire directory. The rpsw is turned on in each entry contained in the directory. The rpsw for the directory itself is turned off and the irsw is turned on.

3. reload_initiate

This procedure is called by the reloader to initiate a file to be reloaded. It differs from the "uid_initiate" call only in its ability to initiate a file for which the rpsw is turned on.

4. reload_terminate

This procedure is called by the reloader to terminate a specified file which has been reloaded. The rpsw is turned off for the file.

5. get_uid_list

This procedure is called by the reloader to obtain a list of uids for all entries of a specified directory for which either the mesw or the fabsw is enabled. By comparing this list to a list of uids obtained from a backup copy of the directory, the reloader can determine the identities of missing entries deserving to be reloaded.

6. load_entry

This procedure is called by the reloader to restore a missing entry to a specified directory for which the mesw or the fabsw is enabled. Either the rpsw or the fabsw may optionally be turned on in the restored entry. Note that the entry must be inserted in uid order within the entry list.

7. reset_dir_switches

This procedure is called by the reloader to turn off both the mesw and the fabsw for a specified directory and to turn on the irsw after all missing entries have been restored.

8. set_dtd

(same as above)

9. set_tape_addr

(same as above)

Salvager Modifications

The current Multics salvager can be employed by the new backup system with relatively few changes. The only significant modification to salvager operation is the communication between the salvager and the reloader. The salvager will indicate to the reloader those directories from which entries are lost. This is accomplished through the use of the irsw and the mesw as described in chapter 3.

The cooperation of the salvager and reloader in assessing file system damage and reloading only missing files and directories represents a considerable improvement over the present Multics recovery scheme. Currently, whenever the salvager fails to recover all but a few files or minor directories, a complete reload of secondary storage must be performed. Even though the salvager may succeed in saving a large fraction of the hierarchy, these files and directories are simply abandoned due to the nature of the current reloading scheme. Because a storage device failure always implies a complete reload of on-line storage, the Multics salvager has not been designed to run after a device failure.

Under the new recovery scheme, a device failure will not necessarily imply a complete reload of on-line storage. As discussed in chapter 2, the level of catastrophe will depend upon the storage allocation strategy used. Currently, the Multics file system views secondary storage essentially as one large allocation pool. Pages of files and directories are arbitrarily assigned to different storage devices. Hence, the failure of any single device is likely to destroy some number of directories and thereby effectively destroy inferior files and directories on other devices. For this reason, even under the new recovery scheme, a device failure is likely to necessitate a major recovery effort.

The proposed salvaging/reloading method will benefit substantially from the adoption of a new storage allocation strategy. The idea of assigning subtrees of the file hierarchy to a portion of a storage device, a whole storage device, or a group of storage devices was described in chapter 2. Under this scheme, the failure of a single device will necessitate reloading only a limited, well-defined section of the file hierarchy. If such an allocation strategy is adopted, it will then become worthwhile to modify the current Multics salvager to operate following a system crash caused by a device failure. This basically involves the additon of a mechanism to inform the salvager that a particular device is "out-of-service" and hence any pages located on the device must be considered destroyed.

One other file system change which could improve salvaging success is the storing of a uid and page number with every page in secondary storage. Following a system crash, the salvager may detect a reused address, i.e. a page in secondary storage claimed by more than one segment. Currently, the salvager has no means for determining the true identity of such a page and hence, for security reasons, the page cannot be awarded to any segment. This problem can be eliminated by storing a uid and page number with every page.

User-Controlled Backup

A user-controlled backup option was described in chapter 2 which effectively gives the user the ability to specify the times at which backup copies are produced. With this degree of control, an application program can guarantee the consistency of backup copies and can record the status of backup copies relative to file updates in progress. Currently, no such facility is available in the Multics file system and little demand for such an option has developed. However, as the Multics system ventures into the commercial world, the need for a user-controlled backup facility may evolve.

Enhancements to the Multics file system will, of course, be required to support the user-controlled backup facility. The file system must be adapted to create and maintain shadow copies. Users must be provided with some means for initiating, updating, and terminating files in user-controlled backup mode. Similarly, the dumper must be provided with a means for initiating and terminating shadow copies.

The file system must manage the creation, updating, and deletion of shadow copies in such a fashion that user processes and the dumper may continue to operate asynchronously without unwarranted delays or other interference. A problem arises, however, if a user attempts to update or terminate a file while its shadow copy is being dumped. Clearly, the shadow copy cannot be discarded at that time. Therefore, the dumper will turn on a dump switch in the entry of a shadow copy before dumping it. If a user attempts to update a shadow copy for which the dump switch is on, a second shadow copy will be created. If a user attempts to terminate a file with a shadow copy for which the dump switch is on, the shadow copy will not be deleted immediately. When the dumper finishes dumping a shadow copy, it will turn off its dump switch. If a more recent shadow copy has been created, or if the file is no longer initiated in user-controlled backup mode, the shadow copy will be deleted.

The frequent updating of shadow copies can be prohibitively expensive, especially when large files must be copied. Therefore, it may be desirable to simply copy the page map for a file rather than copy the file itself. A special bit must be turned on in each page map word of the file to indicate that if the associated page is ever modified, it must be written out to a new secondary storage address. Hence, the file and its shadow copy will initially share all pages of the file in secondary storage. As the file is used, however, modified pages will be written to new secondary storage locations. Note that a file and its shadow copy will not share any pages in main memory.

Three new file system primitives are required for ordinary users:

1. ucb_initiate

   This procedure is called by a user program to initiate a specified file in user-controlled backup mode. If a shadow copy does not already exist, one is created.

2. ucb_update

   This procedure is called by a user program to update the shadow copy for a specified file. The former shadow copy is discarded unless its dump switch is turned on in which case a second shadow copy is created. The dtm of the new shadow copy is returned to the user.

3. ucb_terminate

   This procedure is called by a user program to terminate a specified file. The current shadow copy, if any, is discarded unless its dump switch is turned on or unless the file is initiated in user-controlled backup mode by other users. The dtm of the original file is returned to the user.

Two new file system primitives are needed by the dumper:

1. dumper_initiate

   This procedure is called by the dumper to initiate a specified file for dumping. If a shadow copy exists, the shadow copy is initiated instead. The dump switch is turned on in the entry of the file or its shadow copy, whichever is initiated.

2. dumper_terminate

   This procedure is called by the dumper to terminate a specified file or its shadow copy. If a shadow copy is being terminated, but the file is no longer in user-controlled backup mode, or a more recent shadow copy has been created, then the shadow copy is deleted. Otherwise, the shadow copy remains and its dump switch is turned off.

Suggested Implementation Plan

As can be seen from the preceding discussion, the implementation of the proposed backup system involves a substantial amount of work. Rather than attempting to implement all of the new backup system features at once, the work can be divided into several stages. The first stage of implementation will include only those tasks absolutely necessary to produce a preliminary, operative version of the new backup system. Succeeding stages can then add those features omitted from the preliminary implementation. A plan of this type is suggested below.

Stage 1: A preliminary backup system

1. Modify the file system to accept pathuids.
2. Add new entry items to the directory format.
3. Implement the new backup system interface to the file system.
4. Implement the dumper, reloader, and retriever.
5. Modify the salvager to set the mesw and the irsw.

At this point the new backup system can begin operation. The easiest way to change over to the new backup system is simply to shut down the system and perform a complete dump of the entire hierarchy with the new dumper. Normal Multics operation can then be resumed using the new backup system.

Stage 2: User logins during reloading

1. Modify the file system to properly interpret the rpsw, mesw, and fabsw.

Stage 3: File system reliability improvement

1. Modify the file system storage allocation strategy to assign subtrees to "logical devices" and to duplicate the superstructure of the file hierarchy.

2. Modify the salvager to be able to run after a device failure.

Stage 4: User-controlled backup

1. Modify the file system to create and maintain shadow copies.

2. Implement the user and dumper interface to the user-controlled backup facility.

3. Modify the dumper to use the new interface.

CHAPTER 5

CONCLUSIONS

Summary of Results

This thesis has described a design for an automatic backup system to be incorporated in a computer utility for the protection of on-line information against accidental or malicious destruction. As discussed in chapter 1, the present Multics backup system has served as a starting point in the development of this design. Therefore, it now seems appropriate to review the problems of the Multics backup system which this thesis has attempted to solve, and to identify improvements and innovations in the new design.

The present Multics backup system has succeeded in augmenting storage reliability to the point where users routinely and confidently entrust their only copies of information to an internal file system. This confidence stems from the ability of the Multics backup system to enforce a maximum work loss limit of one or two hours. Unfortunately, however, the Multics backup system has not achieved a reasonable compromise between cost and recovery time. In spite of significant dumping overhead, recovery time from all but the mildest of failures is intolerably slow. Also, Multics operation must be interrupted periodically for backup purposes which contradicts a computer utility objective of continuous service. The most distressing shortcoming of the Multics backup system, however, is its inability to scale up with system growth. The problems of high cost, slow recovery, and interruptions of service all worsen steadily with on-line storage expansion.

The need to periodically shut down the Multics system for backup purposes is due to a special type of dump known as a "save". A save is essentially a page-by-page copy of secondary storage which ignores the logical structure of the file hierarchy. The virtue of a save is that it can be reloaded more quickly than a standard backup dump. Unfortunately, however, a save cannot be performed during normal system operation. Therefore, to limit recovery time, Multics is shut down for a few hours every other day to perform a save. Currently, these interruptions of service are tolerable. However, with continuing on-line storage growth, these interruptions would soon become intolerable. Therefore, saves have been eliminated from the new design.

The high cost of dumping in the current Multics backup system is due at least in part to the inefficiency of the dumper program. Hence, in the new design, the dumper program has been streamlined substantially. The new file system interface for the dumper suggested in chapter 4 should yield a significant performance improvement. Of course, the elimination of saves represents a major cost reduction in the form of increased system availability.

The foremost innovation of this thesis has been the design of an entirely new crash recovery procedure which offers dramatic improvements over the current Multics recovery scheme. The cooperation of the salvager and the reloader in assessing file system damage and reloading only missing files and directories is one such improvement. In the current Multics backup system, no communication link exists between the salvager and the reloader. Hence, whenever the salvager fails to recover all but a few files or minor directories, a complete reload of on-line storage must be performed. Thus, moderate failures are no less disasterous than total failures under the present Multics recovery scheme.

A second major improvement to the crash recovery procedure is a new tape searching strategy employed by the reloader. The current Multics reloader searches every backup tape which might possibly contain files or directories eligible for reloading. In the new design, the reloading operation has been divided into two phases. During phase 1, the latest incremental dumps and the most recent secondary dump are searched for missing files and directories. The bulk of reloading, however, is generally performed during phase 2 which requires no tape searching. The tape addresses of all files to be recovered during phase 2 are known by the end of phase 1. Hence, only those tapes containing missing files need be examined during phase 2. If not for phase 2, it would be necessary to examine all secondary dump tapes produced since the latest complete dump no matter how small or how large the extent of damage. Hence, it is the efficiency of phase 2 reloading which makes the salvager/reloader cooperation valuable.

A third major improvement to the crash recovery procedure is the ability to open the system to users during reloading. This is especially attractive in view of the fact that salvaged portions of the hierarchy will be immediately available. If no system files require reloading, the system can be opened to users as soon as salvaging has completed. Otherwise, the system can be made available to users after phase 1. Hence, only those users whose files are actually destroyed by a system failure need suffer any considerable delay before resuming work.

The fourth and last major improvement to the crash recovery procedure is the parallel processing capability of the reloading operation. Multiple dump tapes can be simultaneously processed both during phase 1 and phase 2 reloading. Therefore, maximum advantage can be taken of available processing and I/O capacity.

All of the above improvements significantly enhance the ability of the backup system to scale up with system growth. Having discarded saves, the problem of system shutdown time increasing with on-line storage size is entirely eliminated. Also, in the absence of saves, dumping costs need not rise sharply with secondary storage growth. The cost of incremental dumping is proportional to system processing activity and therefore will remain a constant fraction of system processing time regardless of on-line storage expansion. The same is true of partial secondary dumping. Although the cost of complete secondary dumping will increase in proportion to secondary storage usage, complete secondary dumping need not be performed frequently. This is feasible because the resolution of the backup system is upheld by incremental dumping and because recovery speed has been substantially increased under the new design.

An important consequence of the parallel processing capability of the reloading operation is that reloading time can be held constant with system growth so long as on-line storage expansion is balanced by corresponding increases in processing and I/O capacity. Of course, in many computer systems, on-line storage growth typically surpasses increases in processing and I/O capacity. Therefore, the time required to fully reload on-line storage will increase. However, if this should become a serious problem, additonal tape drives and I/O channels can usually be added to compensate for storage grcwth.

System failures which necessitate reloading all of on-line storage are extremely rare. Hence, much care has been taken to optimize recovery from the milder, more common failures. Due to the new salvager/reloader cooperation and the new two-phase reloading strategy, the work of recovery has been made commensurate with the extent of damage. Only missing files and directories need be reloaded and, during phase 2, only tapes containing missing files need be examined. Also, the ability to open the system to users during reloading will, in most cases, permit the majority of users to resume work quickly following a system crash. Thus, for all but the very worst and most rare of system failures, recovery time need not increase with on-line storage size.

In addition to reduced overhead and faster recovery, a new functional capability has been added to the backup system design. Currently, no convenient means exists for Multics users to exercise any control over the production of backup copies. The user-controlled backup facility corrects this deficiency by permitting application programs to specify the times at which files subject to modification are copied. With this degree of control, an application program can guarantee the consistency of backup copies and can record the status of backup copies relative to file updates in progress.

Remarks

The subject of on-line storage backup and recovery in general-purpose computer sytems has rarely received adequate attention from system designers. In the past, the idea of a system-provided backup mechanism was often wrongly dismissed as unnecessary or impractical. Today there is little argument about the need for on-line storage backup, but few serious design efforts have been undertaken. Too often system designers have underestimated or failed to recognize the difficult problems and important objectives of a backup system. Consequently, many mechanisms have been implemented which do not offer satisfactory protection and which cannot meet the demands of system growth. This thesis has attempted to expose and to deal with a number of issues inherent in on-line storage backup and recovery. It is hoped that the work described here will be of interest to designers of future computer systems.

BIBLIOGRAPHY

[1] Corbató, F. J., Clingen, C. T., and Saltzer, J. H., "Multics: The First Seven Years", *AFIPS Conf. Proc. 40* (1972 SJCC), AFIPS Press, pp. 571-583.

[2] Schwemm, R. E., "Experience Gained in the Development and Use of TSS/360", *AFIPS Conf. Proc. 40* (1972 SJCC), AFIPS Press, pp. 559-569.

[3] Alexander, M. T., "Organization and Features of the Michigan Terminal System", *AFIPS Conf. Proc. 40* (1972 SJCC), AFIPS Press, pp. 585-591.

[4] Tonik, A. B. (chairman), "Recovery of On-Line Data Bases (Panel)", Proceedings of the 1971 Annual Conference, ACM, August 3-5, 1971, Chicago.

[5] Corbató, F. J. and Vyssotsky, V. A., "Introduction and Overview of the Multics System", *AFIPS Conf. Proc. 27* (1965 FJCC), Spartan Books, Washington, *D.C., 1965, pp. 185-196*.

[6] Daley, R. C. and Neumann, P. G., "A General-Purpose File System for Secondary Storage", *AFIPS Conf. Proc. 27* (1965 FJCC), Spartan Books, Washington, *D.C., 1965, pp.* 213-229.

[7] Corbató F. J., Daggett, M. M., and Daley R. C., "An Experimental Time-Sharing System", *AFIPS Conf. Proc. 21*, Spartan Books, 1962, pp. 335-344.

[8] Crisman, P. A. (editor), "The Compatible Time-Sharing System: A Programmer's Guide", second edition, M.I.T. Press, Cambridge, Mass., 1965.

[9] Bensoussan, A., Clingen, C. T., and Daley, R. C., "The Multics Virtual Memory", *ACM Second Symposium on Operating System Principles*, October *20-22, 1969*, Princeton University, *pp. 30-42*.