Technical Report

UCAM-CL-TR-16
ISSN 1476-2986

Number 16

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Reliable storage in a local network

Jeremy Dion

February 1981

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 1981 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

A recent development in computer science has been the advent of local computer networks, collections of autonomous computers in a small geographical area connected by a high-speed communications medium. In such a situation it is natural to specialise some of the computers to provide useful services to others in the network. These server machines can be economically advantageous if they provide shared access to expensive mechanical devices such as discs.

This thesis discusses the problems involved in designing a file server to provide a storage service in a local network. It is based on experience gained from the design and implementation of a file server for the Cambridge ring.

An important aspect of the design of a file server is the choice of the service which is provided to client machines. The spectrum of choice ranges from providing a simple remote disc with operations such as read and write block, to a remote filing system with directories and textual names. The interface chosen for the Cambridge file server is "universal" in that the services it provides are intended to allow easy implementation of both virtual memory systems and filing systems.

The second major aspect of file server design concerns reliability. If the server is to store important information for clients, then it is essential that it be resistant to transient errors such as communications or power failures. The general problems of reliability and crash resistance are discussed in terms of a model developed for this purpose. Different reliability strategies used in current data base and filing systems are related to the model, and a mechanism for providing atomic transactions in the Cambridge file server is described in detail. An improved mechanism which allows atomic transactions on multiple files is also described and contrasted with the first version. The revised design allows several file servers in a local network to cooperate in atomic updates to arbitrary collections of files.

Preface

I wish to thank my supervisor, Professor R.M. Needham, for his advice, encouragement, and practical assistance during the course of this research. The work described in this thesis has benefitted from his criticisms, and from invaluable and numerous discussions with C.N.R. Dellar. I would also like to thank J.G. Mitchell for his detailed criticism of an early draft of this thesis.

This work was supported by the Natural Sciences and Engineering Research Council of Canada. I would like to thank the Council for their financial support, and for allowing me to take up my scholarship in the United Kingdom.

I hereby declare that this dissertation is the result of my own work and except where explicitly stated includes nothing which is the outcome of work done in collaboration. I also declare that this dissertation is not substantially the same as any that I have submitted for a degree or other qualification at any other university. I further state that no part of my dissertation has already been or is being concurrently submitted for any other degree, diploma or other qualification.

Contents

Chapter 1

Introduction

Recent years have seen the development of computer local area networks [Metcalfe75, Wilkes79a, Hopper78]. These are collections of autonomous computers linked by a common communications medium which enables them to exchange information. Local area networks represent an intermediate point in the scale of computer interconnection techniques. At one extreme are multiprocessor crossbar switches which serve to connect the processors and memory of a single computer, and operate at speeds greater than 10 Mbits/second over distances of a few meters. At the other extreme are international networks which span thousands of kilometers and have transfer rates of 50 Kbit/second or less. Local networks occupy a middle range with internode distances of up to a few kilometers and bandwidths of about 1-10 Mbits/second, though these limits may be extended in the near future.

The use of a local network gives rise to a certain number of alternatives not usually available to the software designer. Consider a program controlling a data structure which provides some generally useful function. We shall use the terms client and server to distinguish the requester and provider of a service. The programs which are the clients of a server are to be distinguished from human users on whose behalf they act. These terms are of course relative, since a server for one client may itself be a client of some other server.

Given a local network, an obvious possibility is the distribution of the server's functions over a number of cooperating machines. By arranging for a number of machines each to execute a part of the server program, it may be possible to perform the service more quickly, or alternatively to replace a single machine by a cheaper collection of simpler ones. In practice, this advantage is not easy to achieve due to the difficulty of decomposing many problems into subproblems which can effectively be solved in parallel. These issues are discussed further in Stroustrup79 and Dellar80.

A second possibility is the replication of a service's program and data by arranging for several machines to provide identical functions to a collection of clients. Replication can increase availability, since a client will have a better chance of contacting a working server, and also reliability if the servers compare their results to detect errors. This idea is well known to hardware designers as modular redundancy [Randell78a, Randell78b]. In contrast with distribution, replication makes a given service more expensive to

perform, and brings no increase in performance. A difficult problem with replication is maintaining the various copies of the service in an identical state. This problem has been extensively studied in the context of data bases [Thomas79, Gifford79, Menasce80], where conflicting updates may arrive simultaneously at different copies of the data base.

Centralisation is the converse of distribution, and is motivated by a situation in which each client has access to a private instance of a service. By replacing the multiple instances by a single shared server, it may be possible to perform the equivalent function with less hardware. The advantages of centralisation are mainly economic, and conflict with the reliability and processing power increases gained through replication and distribution.

To illustrate these three ideas, consider a local network in which each computer has a directly attached disc. The disc and its controlling software can be considered as providing a file service, with the remainder of each machine's code is its sole client.

In this situation, the centralisation of the file service in a single machine with attached discs would probably represent an economic gain. The private disc units could be replaced by a much smaller number of large discs attached to the server machine. If these were also faster than the private discs, the server might be able to share their use effectively among its clients.

If the centralised file server were unable to cope with the volume of requests from its clients, its functions could be distributed among a number of machines. There would be a variety of ways to do this, but all would have the aim of increasing effective parallelism. Loosely coupled partitions in which every client request invokes only some of the distributed functions seem advantageous, since a failure might not halt service completely. One way to do this is by partitioning the files among a number of machines each running a complete copy of the server program. A failure would only prevent access to files on broken servers, since the distributed parts would be relatively independent. Reed [Reed79] and Sturgis et al. [Sturgis80] discuss distributed file systems of this variety. Distribution with tighter coupling is the aim of the designers of CM* [Ousterhout80].

The application of centralisation and distribution to the original situation will have decreased reliability. Initially, the failure of a single instance of the server (the local disc or its controlling software) would have affected only one client. After centralisation, however, a failure of the server would affect all clients. Distribution may also decrease reliability by requiring more machines to be operating correctly for the service to be performed.

Reliability can be increased by replication. By adding enough identical instances of the file service - each possibly distributed - it would be possible to arrange that the reliability of the file service approached that of its only centralised component, the local network. Given replication, the failure of a single instance of the file service would affect no client.

This thesis describes the first step in the above example, the centralisation of management of backing store in a file server for the Cambridge ring. The principal issue was not whether a file server could be built, since the filing system of most operating systems could fairly easily be moved to a separate machine, but rather what design choices would lead to an interface which was useful in a wide range of applications, and an implementation which would share its storage and bandwidth efficiently. Clearly, a centralised file server has the potential of becoming the major bottleneck in a distributed system. Replication or distribution of the file service was not a central issue, since this would have offset some of the economic advantages. However, a method of allowing a number of cooperating file servers on a network is discussed in chapter six as an extension of the basic design.

Distribution of a computation, such as between a client operating system and a file server, brings not only advantages, but also complications in the way failures can occur. In a single machine with cooperation processes, inter-process messages do not get lost, and individual processes do not fail at arbitrary times. In the distributed case, both of these events can occur due to communications errors and machine crashes. These errors will tend to lead to a loss of synchronisation in which the participants in the computation do not agree on how far it has progressed. In a distributed data base, for instance, the crash of one of the participating machines can cause it to lose all knowledge of the transaction in progress. In a file transfer from a client to a file server, the loss of a message in the network can leave a file in a partially updated state.

The problem of maintaining consistency between a file server and its clients is examined in chapters two and three. Chapter two presents the file server interface and describes how communications errors are handled. The file server interface is carefully designed so that all operations are repeatable without side effects; a client receiving no reply will simply retransmit a request without concerning itself whether it was the request or the reply which was lost in the network.

Chapter three discusses the problem of maintaining consistency in the presence of machine failures and concurrency. This problem is not confined to local networks, and has been recognised in data bases for some time [Gray78]. The solution used in data bases has also been

adopted in the Cambridge file server; to allow complex updates to be made safely, an atomic transaction mechanism is used. An atomic transaction has two important properties; it regulates concurrency by appearing to occur either completely before or completely after other conflicting transactions, and it is indivisible over machine failures of a particular kind called interruptions. An interruption occurring a transaction leaves either the initial state of the transaction or the final state, but never exposes an intermediate inconsistent state. The atomic transaction abstraction can be used by clients of the file server to make apparently point-like indivisible changes to files, and thus provides a means for controlling the unpredictable effects of concurrency and interruptions.

There are two main areas of research on which this work is based. One is the relatively new area of local networks [Metcalfe75, Hopper78, Wilkes79a] and of file servers for them. The initial suggestion for this work was a proposal for a "universal" file server by Birrell and de [Birrell80]. Descriptions of two working file servers have appeared in the literature. Swinehart et al. describe the simple WFS file server with page-at-a-time access to files. Sturgis et al. describe the Juniper file server which is perhaps most directly comparable to the Cambridge file server [Sturgis80]. An important feature of their work is that there may be several servers in the network, each controlling a different set of files, which can cooperate to perform atomic transactions on a set of files.

The other main areas of research which have influenced this work are those of concurrency control and error recovery. Eswaran et al. [Eswaran76] give an excellent description of locking as a method of preventing interference between concurrent transactions. Concurrency control has also been studied where identical copies of data must be kept in step [Thomas79, Gifford79, Menasce80], and where the data are distributed [Bernstein80].

Error recovery as a general problem has been defined by Randell [Randell78a, Randell78b, Randell79], and this work is used in chapter three to define a class of error which can be recovered from automatically by the file server. Pragmatic approaches to error recovery in large systems has been extensively studied in operating systems [Fraser69, Stern74] and data bases [Rappaport75, Giordano76, Lorie77]. An excellent case study is presented by Gray [Gray78] and a survey of techniques is to be found in Verhofstad78. Most of the practical mechanisms appear to be variations of the two-phase commit protocol [Gray78] which itself seems to have been derived from the intentions lists of CAL-TSS [Sturgis74]. This subject is pursued further in Chapter three.

Conventionally, concurrency control and error recovery have been considered different problems to be approached with different techniques. Both are required in a working system, however, to prevent transient and unpredictable events from destroying the consistency of a data structure. In a novel approach, Reed shows that by explicitly maintaining the history of changes to data, a more unified approach can be taken [Reed78, Reed79]. A particularly important aspect of Reed's design is that transactions can be combined in a higher-level transaction in a completely general way. This is not true for other reported mechanisms, in which no method for construction of modular transactions is provided.

The work described here is essentially a pragmatic approach to a problem in operating system design. The main requirement was to build a file server which would allow a large number of computers to share a relatively expensive bulk storage resource. The attributes required of it were generality to allow diverse uses, access control to prevent interference between clients, and reliability in order that the consistency of important data could be maintained. Most of all, these requirements had to be met in an implementation which combined effective sharing of the server's storage and processing power with a transfer bandwidth as close as possible to the maximum permitted by the network. Efficiency has thus been an overriding consideration.

Chapter 2

Design of a File Server Interface

The design of a file server interface involves many implicit decisions about its intended use. These lead to choices which are hopefully advantageous in the particular environment, and flexible enough for other unforeseen uses to be accommodated. There is thus no "best" interface for a file server, because the compromise between simplicity and generality will lead to different solutions in different circumstances. In this chapter, the factors leading to the choice of interface for the Cambridge file server will be discussed.

The environment in which the Cambridge file server operates has had a substantial influence on its design. This environment consists of a number of autonomous computers attached to the Cambridge ring [Wilkes79a, Hopper78]. Some of the processors run directly under the control of users, and others provide shared services for general use [Wilkes80]. The ring is extremely reliable and allows point-to-point communication at about one megabit per second for several simultaneous conversations. Although this transfer rate is lower than that of fast disc units by an order of magnitude, it is sufficient to make high speed bulk transfers between hosts on the ring a feasible proposition.

This organisation imposes a number of constraints on the design of a file server. Firstly, a server which is used for all secondary store accesses from several different machines must be capable of high transfer rates to each individual machine, and of sharing its resources among competing clients at a fine grain. If attention were not paid to providing an efficient design, the file server would become an obvious performance bottleneck in the system. As in many other problems of operating system design, a number of alternative solutions to a given problem can be made to work, but the difficulty lies in choosing one which will give acceptable performance.

Secondly, the file server will have no control over the programs which run in the machines which access it. As opposed to designs for distributed file systems such as Reed's [Reed78], no assumptions can be made about the trustworthiness of the programs running in the file server's client machines. Access control barriers are thus clearly needed between the clients and the server, and no reliance can be put on the correctness of any client requests.

2.1 Local Network Communications

To identify the effects of separating a file server and its clients by a local network, we can take as a basis for comparison the corresponding parts of a single-machine operating system. These would consist of a "file server" process through which all secondary store accesses would be funnelled, and a number of user process "clients". These processes would communicate using the inter-process communication facilities provided by the operating system. We shall see that the essential differences between this centralised model, and the distributed one are in the areas of reliability, of communication bandwidth, and of the possibility of loss of synchronisation between client and server.

To the general advantage are the obvious benefits of distributing a computation. There is at least the potential of more work being done in the same time simply due to the increase of available processor cycles. The net reliability of the system may also increase, since the local network provides an excellent barrier to the propagation of errors. In the uniprocessor, a particularly serious error by a user process may bring the entire collection of processes to a halt, and exhausting some implicitly shared resource can cause side effects which propagate to other processes. In the distributed case, however, unless the offending client program floods the local network, it can only affect the operation of its own processor. The network thus provides a convenient firewall against software errors. Hardware reliability may decrease, however, since the minimal amount of hardware which needs to be functional may be greater. Not only must the processor running the file server be in running order, as in the centralised case, but the local network and at least one client processor must also be working correctly. This effect may be reduced by replicating the file server so as to have a number of identical copies.

The other major effects of distribution are due to the characteristics of local network communication. Firstly, by not allowing server and client to share the same memory, access to secondary storage will require store-to-store transfers over the network. A transfer will thus be lengthened by at least the network transmission time required for the data, and correspondingly more if any but the simplest protocol is used to control the transfer. Particular care in the choice of the file server interface is thus required if the potential network point-to-point transfer rate is to be attained or even approached.

As well as being a narrower channel of communication between client and server, a local network has the more serious disadvantage of unreliability. The Cambridge ring is in fact extremely reliable, but of course this does not mean that it can be considered error-free in the design of a file server interface. In a uniprocessor, the inter-process communication system will deliver the request from client to server, and the reply from server to client with near certainty. This is not true in a local network, where either the request or the reply may be lost or garbled in a number of ways*.

Communication errors may cause a client and server to become unsynchronised because the client cannot always know whether a request has been performed or not. In the absence of a reply from the server it is unclear whether the initial request was lost and no action was taken by the server, or whether the reply was lost and the request was in fact successfully performed. The client is thus in potential difficulty, especially if the request was of the nature "withdraw 50 pounds from account 3751"; has the money been withdrawn or not?

To deal with this inherent uncertainty, the client may be able to discover the server's state by sending it further enquiry requests until a reply is obtained. By examining enough information, it may be possible to discover whether or not the request was performed. Unfortunately, this strategy does not generalise, since the enquiries depend on the precise nature of the original update. This sort of error recovery can not be automated without severely restricting the types of requests which can be made.

An alternative is to require the server to detect duplicate requests. To do this, the client must agree to include a sequence number in each request. On receiving no reply from the server, the client would simply retransmit the request without changing its sequence number. The server would need to remember the sequence number of the most recent request, and the reply which was sent to it. On receiving a request with the same sequence number, the server would simply retransmit the reply without performing the operation again.

In an actual implementation, this method becomes rather less attractive. The reply sent to every client request must be retained until the next request is received, and so in effect, the file server must maintain a "virtual circuit" describing the state of its communications with each client. If clients open and close virtual

*In larger networks, there is also the potential for messages arriving out of order. In order for this to happen, there must be two routes from source to destination with different delays, and the transmitter must begin sending of the second message while reception of the first has not yet begun. Neither of these conditions is possible in the Cambridge ring.

circuits to the file server on each request, then the file server need only maintain a small number of virtual circuits at any one time, but performance will be affected by the protocol overheads of managing them. If, on the other hand, each client opens a virtual circuit once at the start of day and uses it for all interactions, then the number of these structures in the file server could be a potential embarassment.

These difficulties arise because in general performing a remote request several times will produce a different state from performing it just once. This is a semantic property of the request, but a particularly useful subclass of requests is repeatable. A repeatable request is one which has the same effect when executed any number of times; it can thus cause a state change the first time it is performed, but on repeated executions has no observable effect.

Clearly, many of the functions which one would wish a file server to perform are inherently repeatable. All enquiry requests, such as reading a file, are repeatable. With a bit of care, writing to a file can also be made repeatable. This requires that the write request specify the locations to be written absolutely, perhaps by an offset within the file and a length, rather than relative to some current position maintained by the server. In the latter case, repeated attempts to write could cause the same material to be appended several times to a file. If absolute coordinates are given, however, repetitions merely cause the same part of the file to be overwritten with the same data.

If all operations in a file server interface are repeatable, then a substantially simplified approach can be taken to network communications. Since by assumption there is no visible difference between succeeding in performing a request the first or the fifth time, retries can be treated exactly as initial attempts. No sequence numbers are needed, and no virtual circuits need be maintained.

For these reasons the interface for the Cambridge file server is carefully defined so that each operation is repeatable. In fact, it is only necessary that repetitions cause equivalent states to be seen by clients; the same sets of future requests must be valid in each case, but repetitions need not produce identical states. For example, repeatedly requesting the creation of a file can result in a new file each time (and the destruction of its predecessor), but at the end of the sequence there must still be exactly one file of the required characteristics. It is not necessary, however, that the file created on the first attempt be the one returned on all future repetitions.

2.2 The File Abstraction

A file server must provide access to some storage abstraction which can be read and written freely. The primary job of the file server is to create and delete such files and to accept requests to read and write them.

The appropriate interface to such a file abstraction is determined by its expected pattern of use. For the Cambridge file server, a proportion of the accesses were expected to be from the CAP virtual memory system [Dellar80a, Dellar80b] as segments or parts of segments were swapped in and out of main store. Here in particular, it was important to achieve the maximum possible transfer rates. This militated against a view of files as a sequence of fixed-length blocks which could be accessed one at a time, as done in the WFS file server, for instance [Swinehart79]. In this method, the transfer of a large file would require many separate accesses to its constituent blocks. It would incur the network delays of transmitting each request, and the much more significant software delays at each end to start, sequence, and stop each block transfer. Swapping might therefore become an intolerable overhead, especially if data returned by the file server could not be written immediately into the reserved segment, but had to have protocol information removed first.

To achieve higher transfer rates, a more convenient file abstraction was defined. A file is a sequence of 16-bit words. Read and write operations define a transfer by a file identifier, a starting word offset, and a transfer length.

A file has two attributes, its size and an uninitialised data value. It is useful if the size is virtual, and defines the maximum permitted address rather than the actual or allocated size. This simplifies many transactions such as compilation where the required size of a file is unknown. In this case, a compiler can initially create an enormous file on the understanding that those parts of a file which are never written will not occupy storage, and it can shrink the file to its required size when this is known. The second file attribute, the uninitialised data value, is a conventional word value which is returned when a client reads words of a file which have never been written.

As well as being a convenient equivalent of the segment abstraction in virtual memory machines such as the CAP, the file abstraction defined by these read and write requests lends itself to high transfer rates. By arranging that the control information flows in a separate logical stream from the data, file contents received from the server can be stored immediately where they are needed, and data received

from the client can be prepared for immediate transfer to secondary storage. The details of this mechanism will be presented in section 5.10, but for the present, we may define the file operations by the following procedures:

read (ID file, INT offset, INT length) DATA

write (ID file, INT offset, INT length, DATA to write)

read file size (ID file) INT

change file size (ID file, INT new size)

where DATA is understood to represent "length" words of arbitrary information and INT is a 32-bit integer. Note that repeatability requires that a changed file size be specified absolutely, rather than relative to the current size.

2.3 Access Control

If a file server is to maintain files for a number of untrustworthy clients, then some means must be devised for preventing undesirable interference between them. One client should not be able to read or modify files owned by a second client without consent. In common with the design of virtual memory systems, however, enforcing complete separation between clients is relatively easy, while providing for controlled sharing is quite difficult.

There are two main choices for building a protection system [Lampson69, Wilkes75]. One is based on the identity of the client; the server allows access to a file only for a particular set of clients, and remembers for each file a list of client names and their privileges. The other mechanism is based on the ability of a client to present a valid capability, or ticket, for the object [Dennis66]. Access to a file under these two schemes may be compared to access of people to a building. Having a security guard check each person against a list of those allowed to enter the building is analogous to an identity-based mechanism. Giving each authorised person a key to the building is analogous to the capability approach.

A characteristic of the identity-based mechanism is that the server must know the name of each client making a request. In the centralised case, this is relatively easy. Presumably, the client process is running on behalf of some user who has identified himself to the operating system by quoting a password. The information which

allows the server to connect requests with authenticated names is thus easily available. This may not be true in the distributed case, because of the autonomy of the individual nodes. One of the attractions of a local network is that the processors which run client programs can be relatively independent; users may be able to obtain machines and load arbitrary programs into them without prior consultation. In the absence of a central authority which notes the arrival and departure of clients, identity-based access control becomes difficult.

One approach would be to oblige client machines to "log in" to the file server for the duration of a session, and identify themselves to it by the initial presentation of a user's password. This approach is feasible, but it seems to involve more mechanism than might be wished for the purpose at hand. The file server would have to provide functions for creating and changing passwords, and would need to note the arrival and departure of clients, timing out their sessions after some suitable idle time. At this point, the capability approach seems more attractive, because it does not depend on any validation of identities.

A capability in a distributed system can be represented by a number chosen from a very large space. By including enough random information in the number, it is possible to arrange that the set of capabilities for existing files is scattered thinly over all possible bit patterns and that capabilities for deleted files are unlikely to be reused. If this is done, it will be difficult for a client to manufacture a capability, or to predict the next in a sequence of recently generated capabilities. In a centralised system, restricting the direct manipulation of capabilities to trusted parts of the operating system provides a guarantee against forgery; only keys from a certified locksmith can gain entry to the building. In the distributed case, since it cannot be assumed that any of the client's code is trustworthy, the guarantee must be replaced by a probability based on the difficulty of forgery.

This approach is taken in the file server interface. All files are identified by unique identifiers (UIDs), which are large integers and behave like capabilities. On creation of a file, the file server creates a UID for it, and records the binding between the UID and the file for the file's lifetime. By arranging that each request must contain a UID for the file to be read or written, the file server is able to validate each request in a simple way; it need only check that the UID presented is for an existing file. An additional advantage to this approach is that capabilities can be freely exchanged and copied by clients, much as the owner of a house might give keys to good friends. In an identity-based system, the creator of a file would be obliged to inform the file server each time he

wished to grant access to a file to another client.

A drawback of the capability approach is the difficulty of associating access rights with capabilities. In centralised capability machines [England72, Wilkes79b, Wulf74], this is done trivially by reserving a few bits in the capability to encode the access rights which it allows on the object. Where capabilities are subject to arbitrary alteration, however, this is not sufficient; it must not be possible for the possessor of a capability to amplify his rights by modifying it. It is thus necessary to generate a completely new capability which bears no perceptible relationship to the original, and for the file server to retain bindings for different capabilities and access rights to the same file. This is not a fundamental objection to this type of capability, but an inconvenience. In the actual implementation, no provision is made for capabilities with different access rights.

2.4 Control of Storage

Janson [Janson76] views the principal function of a storage system as the mapping of a potentially infinite set of objects which are created and destroyed onto a finite set of storage containers which are allocated and freed. At any time, the storage system must maintain the bindings between the set of objects currently in existence and the storage containers in which they reside. Since the supply of these storage containers is limited, one of the most important functions of the storage system is deleting the bindings for unwanted objects so that the storage containers which hold them can be freed for reuse. This is in itself simple, but deciding when an object is unwanted is difficult.

There are two widely used methods for controlling object deletion. In implicit deletion, storage is reclaimed when it is discovered that no program will ever refer to an object. Since the future behaviour of programs cannot be predicted, the more conservative approach is taken of reclaiming storage only when no program can possibly obtain a reference for the object. It must be noted that this method can be used only in capability schemes; if a program can fabricate references at any time, then it has the potential to refer to every object. The normal mechanisms used in implicit deletion are reference counts, garbage collection [Deutsch76, Dijkstra78], or a combination of the two [Birrell78, Garnett80].

Explicit deletion is used in non-capability schemes, where there is no basis on which implicit deletion can be performed. This simply follows from the observation that since the storage system cannot

decide whether a program will ever access an object, this information must be explicitly supplied by the program using a delete operation.

The distinction between these two methods of reclamation is well illustrated in programming languages which provide permanent "heap" storage as well as local storage on the procedure invocation stack. In Algol68 [Lindsay77], where a reference is a protected type in the language, enumeration of all references accessible to a program is possible, and implicit deletion can thus be performed by a garbage collector. In BCPL [Richards69], which does not distinguish between references and other kinds of variables, an explicit delete operation is used.

In the context of a local network with autonomous nodes, explicit deletion would seem to be the obvious choice for storage control in a file server. Since the file server allows the names of files to escape into highly unreliable environments where they may be copied, lost or passed to any number of other clients, there is clearly no way in which the file server or any other program can discover which UIDs are known to at least one client. Implicit deletion is thus impossible.

Explicit deletion, however, suffers two serious defects. In any but the simplest cases, it may be as unclear to the client as to the server when a file can safely be deleted. If its UID has been passed to another client, then each must be prepared to deal with the premature deletion of the file due to the action of the other. Even within the set of files owned by a single client, this decision may be difficult. In the CAP filing system, for instance [Needham77, Dellar80a], a file may not be deleted while any directory contains an entry for it; explicit deletion could thus only be performed by the CAP directory management program as a result of a search of all those file server files in which its directories were stored.

A more serious disadvantage than premature deletion, which can after all be solved by sufficient cooperation between possessors of a UID, is the inevitable creation of lost objects. Creation must obviously be done by an explicit request from client to server which returns the UID of a newly created file. Nothing, however, prevents any of:

1) the file server crashing after creating the object but before sending the reply,
2) the reply which holds the only copy of the UID being lost in the network,
3) the client crashing before being able to record the UID in non-volatile storage.

All three eventualities can lead to the only copy of a UID being lost. The file server will of course have recorded the binding between the UID and its storage container, but no client will ever be able to

request its deletion.

Two solutions to this problem are reported in the literature, both of which are unsatisfactory. In the WFS file server [Swinehart79], the interface allows the enumeration of all files currently known to the file server so that their UIDs may be discovered. Presumably, those not matching in a list of "known" files are explicitly deleted by a program which periodically performs this enumeration. This program, however, will require the cooperation of all clients in compiling the list of known files. Alternatively, requiring each client to be able to enumerate only those files which it created raises again the question of identifying clients to the server in a satisfactory way, both at the time of each creation and during enumeration.

A second solution is proposed in the Pilot operating system [Redell79], in which an interface to a storage system roughly similar to that proposed thus far in this chapter is defined, though within a single machine operating system. On first creation, files are marked as temporary. After recording the file identifier in some safe place, the client calls the file manager to mark the file as permanent. The lost files arising from crashes in the circumstances above are thus precisely the temporary files. During restart, therefore, the storage system enumerates all files, and destroys those marked as temporary. Obviously, it is assumed that clients are sufficiently well behaved to mark files as permanent only when a reference has been safely recorded; marking a file permanent is essentially an undertaking by the client not to lose all references to it. Where the protection barrier between a file server and its clients can be moved further away from the file server into the interface to a directory manager subsystem, this approach is acceptable. It is not if no such trusted subsystem can be assumed to be running in each client machine.

In the context of the Cambridge ring, explicit deletion would seem to pose unacceptable problems. The initial proposal for a "universal" file server [Birrell80], however, provided for implicit deletion in a particularly elegant way using an index abstraction.

The index is the second abstraction provided at the file server interface, and serves both as a simple structuring method for defining relationships among an arbitrary collection of files, and as a means of controlling storage allocation. An index consists of a list of UIDs of files and indices which are held at numbered offsets, and it has a single attribute, its size. There are five index operations:

retrieve (UID index, INT offset) UID
   returns the UID stored at a particular offset in an index.

delete (UID index, INT offset)

overwrites an index entry with zeroes.

retain (UID index, INT offset, UID object)
if 'object' is a valid UID for an existing file or index, it is written into the selected index entry.

read index size (UID index)
returns the maximum number of UIDs which can be stored in the index.

change index size (UID index, INT new size)
adjusts the size of an index. Decreasing the size causes deletion of any UIDs in entries beyond the new size.

Using these operation, a client possessing the UID of an index can discover the UIDs of the objects retained in it, and can repeat the process on any indices found to discover all object reachable from that index. It can also alter this structure in any way it chooses using delete and retain operations, because as for files, read and modify accesses are not distinguished. Any UID can be retained in an index, including that of the index itself, so that completely general graph structures can be created in which cycles may occur.

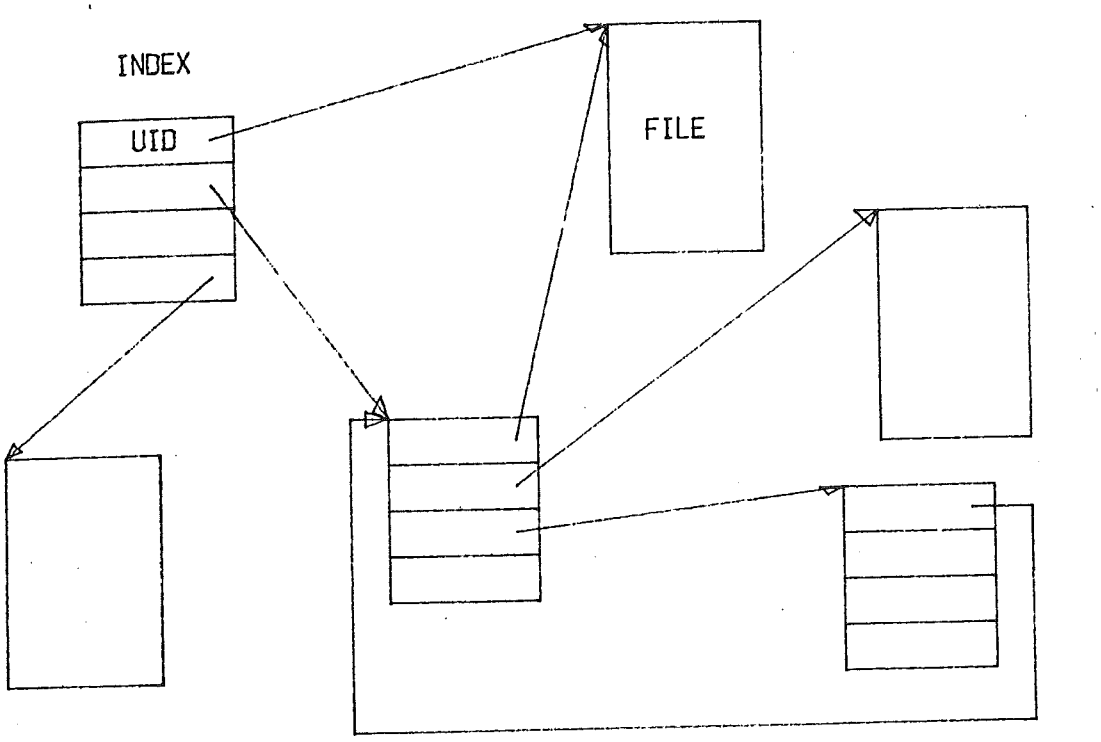

Fig 2.1 File Server Graph Structure

The use of indices for storage control is due to side effects of the retain and delete operations. Among all indices, one is distinguished as the root index of all the objects controlled by the file server. Storage control is performed by defining that the file server will keep in existence only those files and indices which are reachable from the root index by a series of retrieve operations. An object is deleted as a side effect of deleting the last index entry which connected it to the root index. A single delete operation can thus potentially cause the destruction of a large number of objects. Because cyclic structures can be created, it is necessary to use garbage collection periodically to detect those objects which have become detached from the root index. The file server uses an asynchronous garbage collector running in a separate machine in the network. The details of this mechanism are described in Section 5.20.

In practice, it is quite simple for clients to direct the file server's storage reclamation. When a new client wishes to use the file server, an index is created and retained at a free entry in the root index. The unique identifier of this index is then embedded in the client programs. Neither it nor any other client can delete this entry in the root index, since the UID of the root is never publicly distributed. The client can thus create files and indices and retain their UIDs in this index, or in some index reachable from it, with the assurance that they will not be deleted. More generally, the index given to the client when it first gains access to the file server can be considered as the root index of its subgraph of the file server storage.

When a client receives a unique UID from some source, its first action should be to retain it in some index which is known to be reachable from its own root index. If this retention succeeds, the UID is known to be valid, and will not be invalidated until the new index entry is destroyed. When the client has finished using the object, it need only delete the index entry for it. If this makes the object unreachable from the file server's root index, the object will be destroyed by the file server. If, however, some other client still wishes to use it, a copy of its unique UID will still be retained somewhere. Thus, as opposed to a method relying on an explicit delete operation, no interaction is needed to delete a file shared by several clients.

The restriction that all objects to be kept must be reachable requires special attention when objects are created. To avoid an object being retained in no index for a brief time between its creation and the first request to retain it, the create file and create index operations require an index and an offset to be specified. After the object is created, but before its UID is returned to the client, the UID is retained at the given offset in the

index. Only when it is known to be retained successfully is the new UID returned to the caller. Note that if the reply never reaches the client, and the request is repeated, the UID of the second file will overwrite that of the first file in the index entry, causing deletion of the first file.

Thus the requests for creating indices and files are as follows:

create file (UID index, INT offset, INT size, INT uninit) UID
returns a UID for a file of the requested maximum size and with uninitialised store value 'uninit'. Before the UID is returned, it is retained in the supplied index entry.

create index (UID index, INT offset, INT size) UID
returns the UID of a new index which is first retained in the specified index entry.

As a matter of implementation, it is not necessary that the creation of an object and the retention of its UID in an index be indivisible. It is only necessary that each of these two operations be indivisible, so that in the event of a crash, half-created files or half-written indices are never exposed. The actual file server implementation can crash between these two operations, however, leaving a new object retained in no index. This is perfectly acceptable, since the object will be seen to be unreachable at the next garbage collection.

In summary, the Cambridge file server performs storage control by defining two classes of unique identifiers. A copy of a UID possessed directly by a client allows operations to be performed on an object, but cannot prevent the object from disappearing. A copy of the UID stored in an index reachable from the root cannot be used to specify operations directly, but serves to guarantee the object's continued existence. The retrieve and retain operations are the means for converting between these two types of UID.

2.5 Accounting

The storage controlled by the file server represents a shared resource for its clients, and so provides the opportunity for accidental or malicious interference. The interface presented in this chapter contains no means of controlling the amount of storage owned by a client. As with access to individual files and indices, access to the shared storage is on an all-or-nothing basis. Any client possessing the UID of an index can create files without restriction,

ultimately at the expense of all other clients.

To partition a shared resource equitably, accounting of usage is needed. Once limits have been set by an administrator, the accounting mechanism must monitor the demands made by all clients to ensure that their limits are not exceeded. This problem is common in general-purpose operating systems, and the approach used in the Multics filing system [Janson76, Organick72] may be taken as an example. Each file directory in the tree-structured hierarchy can have an assigned quota of disc blocks. This quota represents the total amount of space which can be used in descendents of that directory. A directory with a quota can assign part of its quota to a subdirectory, thus guaranteeing it the use of a certain amount of space. Accounting is done at file and directory creation time; the directory hierarchy is searched upwards until a directory with a quota is met, and the creation is allowed only if it does not cause the current running total to exceed the quota.

This method of accounting encounters problems if a general directed graph is considered rather than a tree. In the file server, for instance, there may be several paths leading from the root index to a particular index or file, and it would be at best arbitrary to assign the cost of the object to a single client if several had references for it. In the absence of cycles in the graph, a fair alternative would be to divide the cost of an object by its index reference count, and to assign one part of this cost to each index holding a reference to it. In this way, the cost of storage reachable from an index could be computed, and by proceeding backwards towards the root index, the cost to be assigned to each client could be calculated. This information would be time-consuming to update incrementally as files were created and deleted, and the method breaks down in the presence of cycles in the graph.

A workable, but less fair solution would be for each index to carry the identity of the client which created it. This can be done without raising the problem of identification of clients at the time of a request, by including in each index the UID of a special accounting file. When a client first gains access to the file server, its root index would have the UID of a new accounting file stored in it at a hidden offset. Ordinary indices would be given the accounting file of the parent index specified at the time of creation. This would give a

systematic, if not completely fair, method for accounting. The cost of a file could, as before, be divided among those indices referring to it; the cost of an index would include the cost of its directly contained files, and would be assigned directly to the accounting file*.

Though workable, this accounting scheme may not be equitable. Once an index is created, all new storage attached to it will be charged to the creator; if the creator passes the index UID to another client and deletes all its index entries for it, he will still pay all storage costs. Thus, in practice, clients would probably never exchange index UIDs.

Ideally, an accounting scheme should compute the set of objects reachable by every client, and then assign costs based on the number of sets in which each object was contained. At a gross level, this information could be quite easily computed during the graph enumeration necessary for garbage collection, but it is hard to see how the information could be kept up to date incrementally as the graph changes.

The actual file server implementation contains none of this machinery, and space accounting is done in a rather crude way. In a research environment of reasonable size, social pressure can be as effective as an accounting mechanism in limiting the demands on a shared resource, and does not require administration. To make garbage collection possible with removable disc packs, each disc pack is in fact a self-contained graph with its own root index, and no inter-pack references are allowed in indices. A new client given an index on some disc pack is therefore automatically constrained to create all new objects on that pack, since at the time of creation each object must be preserved in some index. Thus, the only accounting control possible is exercised in allocating a client an index on a particular disc pack. Thereafter, cooperation is required between clients if a disc pack becomes full.

* If the client were not to be charged for parts of a file which were never written, files would also have to hold an accounting file for use during file writes, and the partitioning of file cost could not be made between different indices.

Chapter 3

Maintaining Consistency

A file server used for all secondary storage by its client computers will probably be called upon to store important data, where importance may informally be taken to be a measure of the disruption which would follow destruction of the data. It is thus worth asking what facilities provided by a file server would help or hinder the task of keeping such structures in a correct state.

There are limitations on what can be done, however. A file is a particularly unconstrained object type; a client may change the contents of a file arbitrarily, whether or not this change corresponds to a correct new state. It is thus impossible for a file server to discover that a client is attempting to make a syntactically correct, but semantically nonsensical change; interpretation of file contents is explicitly not part of the file server's function. We must therefore adopt the more conservative position of assuming that any change to a stored data structure requested by a client is correct. What remains is the isolation of such changes from events beyond the client's control.

To motivate the definitions to follow, consider a data structure consisting of an index and a file which is retained in entry zero of the index. We might wish to write a procedure which will retain a given UID in the first free entry of the index, and increment a count of uses of the entry in the associated file. Fig 3.1 shows a BCPL procedure to do this.

```
LET store (uid) BE
$( LET file = retrieve (index, 0)
   FOR entry = 1 TO readindexsize (index)
   DO IF retrieve (index, entry) = 0
      THEN $( LET count = readfile (file, entry, 1)
              writefile (file, entry, 1, count+1)
              retain (index, entry, uid)
              RETURN
           $)
$)
```

Fig 3.1 A Transaction

When executed, this procedure performs a retrieve operation, a read index size operation, more retrieves, a file read and write, and a retain operation.

This simple example illustrates a number of general points. The operations requested of a file server by a client are grouped into transactions [Gray78], each equivalent to the execution of some procedure like 'store'. A transaction makes a change to a number of objects, its write set, based on the read set of objects examined, and some external information. In the above example, the write set and read set are both {index, file} and the external information is the UID to be stored.

When no transactions are in progress the data are by assumption consistent in that they constitute a valid representation of the client's abstraction. In our example, in a consistent state each entry of the file will contain the exact number of times the corresponding index entry has been used. A transaction, therefore, serves to change one consistent state into another. It does so by performing a series of writes which change the members of the write set incrementally, but in the process it produces temporary inconsistent states. In the store procedure, the counter in the file is updated before the index is modified, so that for a brief period, the file does not accurately represent the index entry usage.

Thus, the notion of a transaction defines that of consistency; the data are consistent while no transaction is in progress, and inconsistent while a transaction has partially, but not completely modified its write set [Gray78].

Transactions are not always as predictable as this example. As Reed points out [Reed78], highly interactive transactions may demand external information on the basis of values found in the read set, and may choose to read and write other objects on the basis of the new values supplied. It is thus not possible in general to predict the members of the read and write sets when a transaction starts.

A simple model captures the idea of a transaction:

Definition: Let $O_1, \ldots, O_n$ be a set of objects. A transaction T with external data e is a sequence of operations (A_i, O_i, V_i), where $A_i \in \{read, write\}$, O_i is an object, and V_i is the value of the object which is read or written:

$$T(e) = ((A_1, O_1, V_1), \ldots, (A_n, O_n, V_n))$$

At the i^{th} step of the transaction, the read set $R_i(T)$ is

$$R_i(T) = \{O_j \mid A_j = read \text{ and } j <= i\}$$

and the write set $W_i(T)$ is

$$W_i(T) = \{O_j \mid A_j = write \text{ and } j <= i\}$$

For each write operation $(write, O_i, V_i)$, the value written V_i is a function of the external data, and of the members of the current read set:

$$V_i = f_i(e, R_i(T)).$$

3.1 Effects of Concurrency

Concurrency in performing transactions is important in a centralised service which might otherwise become a performance bottleneck. In fact, in the interface presented in chapter two, concurrency of transactions is inevitable since no mechanism has been provided to prevent it by synchronising the requests of different clients.

By interleaving operations from several transactions, it may be possible to increase overall speed, but consistency may also be violated. If the read set of a transaction intersects the write set of another, for instance, the first transaction might not observe a consistent state and could propagate the effects of this to its own write set.

Eswaran et al. [Eswaran76], give the minimum constraints needed to preserve consistency when concurrency is allowed. A schedule in their terminology is a permutation of operations from m different transactions

$$S = ((t_1, A_1, O_1, V_1), \ldots, (t_n, A_n, O_n, V_n))$$

where the A_i, O_i and V_i are as before, and t_i is a transaction identifier for transaction T_i. Thus

$$T_i = ((t_j, A_j, O_j, V_j) \mid t_j = t_i)$$

which is ordered as a sequence using the order of occurrence in S. The dependency relation < is defined between transactions as follows. Given a schedule S, $T_a < T_b$ if there are two operations in S, (t_i, A_i, O_i, V_i) and (t_j, A_j, O_j, V_j) such that $i<j$, $t_i = t_a$, $t_j = t_b$,

$O_i = O_j$, A_i = write and A_j = read. In other words, $T_a < T_b$ in S if T_a writes a variable which T_b reads subsequently. Eswaran et al. show that a schedule S will result in a consistent state if and only if S gives rise to the same dependencies as some serial schedule S' in which the operations of each transaction are performed consecutively.

Intuitively, the relation < defines the order in which the transactions make changes to the shared objects, and may be read "is earlier than". An examination of the types of interference possible under concurrency may clarify this result.

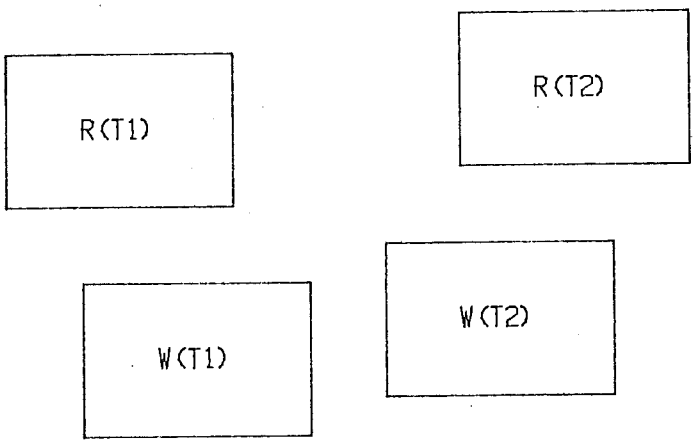

Fig 3.2 Independent Transactions

Figure 3.2 shows two transactions whose read and write sets do not overlap. No schedule of the operations of T_1 and T_2 can give rise to dependencies between them, so any schedule will produce the same final state.

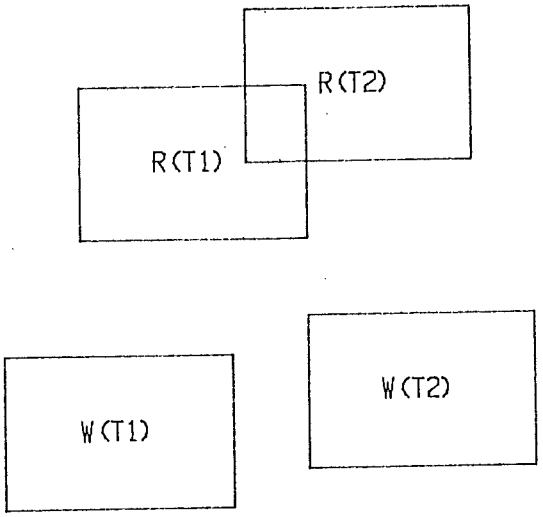

Fig 3.3 Overlapping Read Sets

The same is true where only read sets overlap, as in figure 3.3, because no future transaction would be able to decide whether T_1 or T_2 had run earlier. Since there are no dependencies between them, any schedule of T_1 and T_2 will produce the same consistent state.

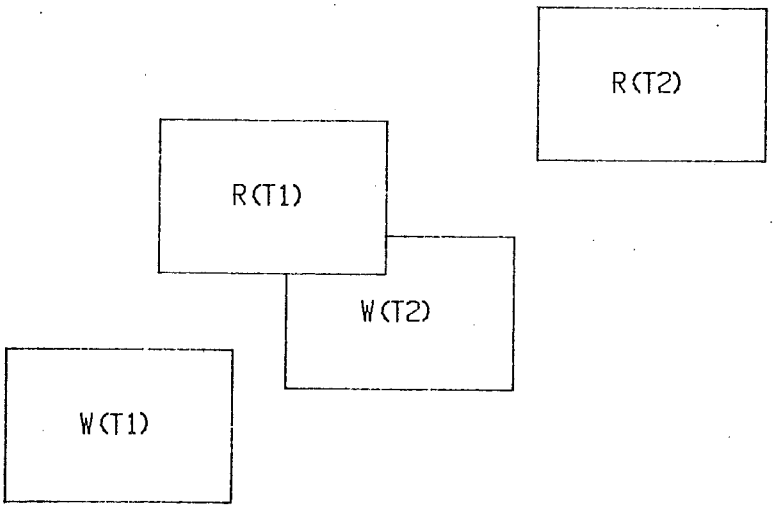

Fig 3.4 Overlapping Read and Write Sets

In figure 3.4, an ordering is necessary because $R(T_1)$ and $W(T_2)$ overlap. This will have been detected as the two sets grew during the steps of the schedule until a read operation by T_1 or a write operation by T_2 caused the sets to intersect. At this point, it would have been necessary to choose whether to allow T_1 to read T_2's new value ($T_2 < T_1$), or the previous value ($T_1 < T_2$). Either choice is equally good, but it must be adhered to for all elements in the intersection. If this were not done, then we would have $T_1 < T_2$ on some elements of the intersection and $T_2 < T_1$ on others, dependencies which could not be produced by any serial schedule of T_1 and T_2. What is being done, of course, is to guarantee that T_1 sees a consistent state containing either all of T_2's changes or none of them, but not an intermediate inconsistent state. The covering of $W(T_2)$ by $R(T_1)$ in figure 3.4 is meant to indicate a choice of $T_2 < T_1$.

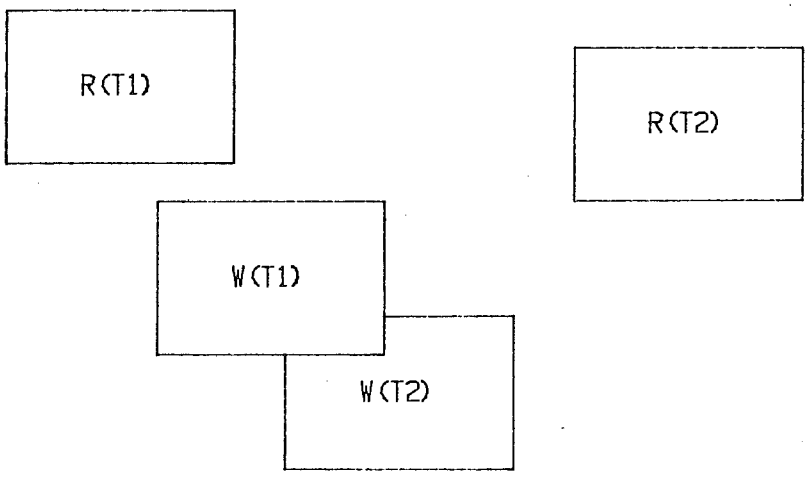

Fig 3.5 Overlapping Write Sets

Figure 3.5 shows the essentially similar case of overlapping write sets. The problem here is not in any ordering between T_1 and T_2, since neither writes a variable read by the other. Rather, a transaction T_3, whose read set was contained in the intersection between $W(T_1)$ and $W(T_2)$ would not see a consistent state. If some variables had been written "last" by T_1 and some by T_2, then we would have $T_1 < T_3$ and $T_2 < T_3$, whereas in a serial schedule, one or other but not both of these dependencies would hold. In this case, it is necessary to make all the writes of one transaction before those of the other to ensure consistency, and figure 3.5 shows T_2 to have run earlier than T_1 from the point of view of all later transactions.

Of course, for any pair of transactions which run concurrently, any of these types of interference may occur. Figure 3.6 gives an example in which all three have arisen.

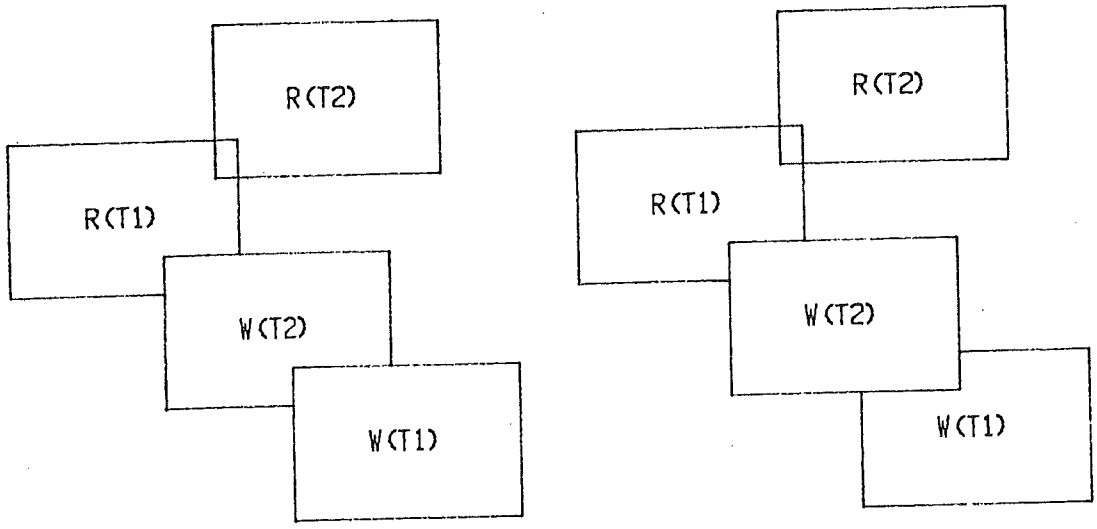

Fig 3.6 a. Inconsistent Orderings b. Consistent Orderings

In figure 3.6a, the ordering decisions have not been compatible. The overlap between $R(T_1)$ and $W(T_2)$ has resulted in $T_1 < T_2$, while that between $W(T_1)$ and $W(T_2)$ has resulted in $T_2 < T_1$, so that figure 3.6a could not have been the result of T_1 applied after T_2 nor of T_2 applied after T_1. In figure 3.6b, the consistent ordering $T_1 < T_2$ has been chosen.

The problem of maintaining consistency in the presence of concurrency involves detecting when orderings between transactions are necessary, and at each conflict imposing an ordering which is compatible with those chosen previously. Maintaining compatible orderings means preventing the creation of cycles of dependencies $T_i < T_j < \ldots < T_k < T_i$ which are impossible in a serial schedule.

Two methods for resolving this synchronisation problem are reported in the literature. Locking is the conventional method, and is examined in detail by Eswaran et al. Transactions are required to

lock each object before it is read or written. Defining two classes of lock, read and read/write, allows transactions with only overlapping read sets to be unordered. In addition, every transaction must acquire locks on all objects read and written before it releases any lock. In terms of the simple model used here, the requirement that each object be locked before it is read or written simply means that whenever there is a read/write or a write/write conflict between two transactions, some ordering will be imposed, since one transaction will be delayed while the other, earlier, transaction proceeds. The second condition states that these individual decisions must be self-consistent. A transaction which has successfully acquired locks on all objects accessed is "earlier" than all its delayed competitors, and so cannot be part of a dependency cycle. Bernstein et al. discuss simplifications to a general locking strategy which can be made if the set of transactions is fixed and their interactions are predictable [Bernstein80].

A second novel solution has been proposed by Reed [Reed78, Reed79]. This is based on explicitly representing every object as a sequence of versions as shown in figure 3.7. Writing to an object inserts a new version in the sequence rather than replacing its contents. These versions are ordered by "pseudo-time", which is essentially like real time except that while two clients may attempt an operation at the same instant of real time, they are constrained to choose different moments in pseudo-time.

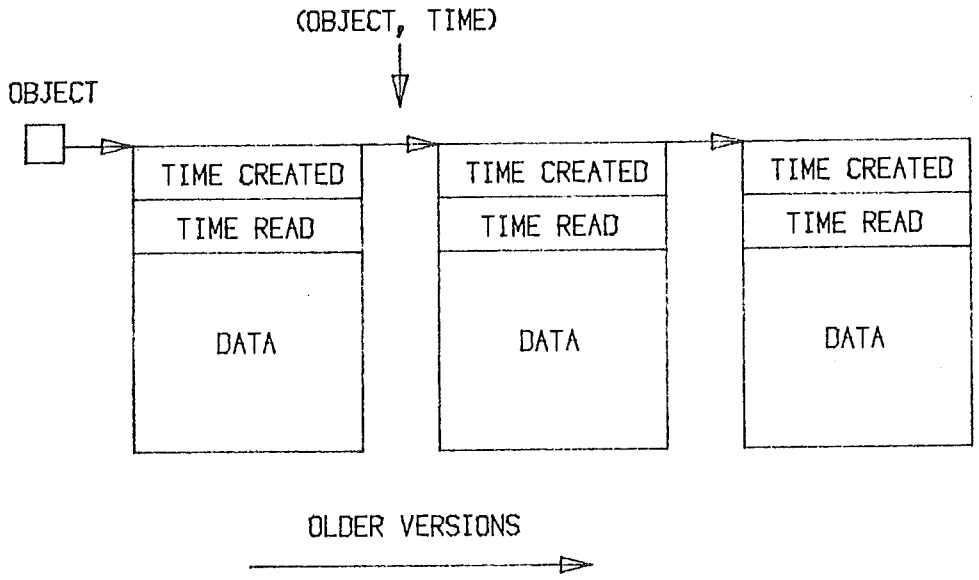

Fig 3.7 Explicit Object Histories

In Reed's scheme, a transaction begins by choosing the instant of pseudo-time at which it wishes to execute, perhaps by appending a client identifier to the value of a real time clock. All read

accesses are made by selecting the object version which is most recent at the chosen pseudo-time. Write accesses are made by attempting to insert a new version in the object history at the transaction's chosen moment of pseudo-time. However, the insertion will be refused if the preceding version has already been read at a later pseudo-time by some other transaction. This scheme is original and elegantly simple, but contains unexplored implementation difficulties. Some of these, such as discarding old versions of objects after a suitable time interval, are considered by Reed, but others do not seem straightforward. In particular, maintaining representations of large objects as a sequence of versions might imply inefficiencies in the use of space or in access time, or in both.

An advantage of Reed's scheme over locking is that the write sets of two transactions cannot interfere. Since each executes at a different moment of pseudo-time, they can be allowed to proceed in parallel; any object in the write set of both will simply have two new versions inserted in its history at differing moments in pseudo-time.

In both the locking and pseudo-time schemes, it is possible for a set of concurrent transactions to interfere in such a way as to make it impossible for all of them to produce consistent results. In the locking strategy, this is the familiar problem of deadlock, in which a number of transactions refuse to release locks they already possess while waiting to acquire a lock held by another. This situation cannot be prevented since in general the read and write sets of a transaction are unpredictable. Nor can it be resolved to the satisfaction of all transactions, since each insists on running earlier than its competitors.

In Reed's scheme, the equivalent situation arises when the operations performed by two transactions in real time are out of order in pseudo-time. Much of the attractive simplicity of Reed's scheme is due to the fact that by choosing a moment of pseudo-time, a transaction orders itself with respect to all others; it intends to see the results of all transactions with lesser pseudo-times, and to make its results visible to those with greater pseudo-times. Real time orderings of reads and writes may make this impossible, however. Consider two transactions T_1 and T_2 with $T_1 < T_2$ because T_1 has chosen a smaller pseudo-time. Suppose there is an object O in $W(T_1)$ and in $R(T_2)$. If for some reason, T_1 is slow in executing, T_2 will read O in real time before T_1 writes it. The read must be allowed, because it cannot be known that T_1 intends to write O. When the write is attempted, however, it must be prevented, because T_2's having read O first makes $T_1 < T_2$ impossible. Thus, the arbitrary ordering imposed by the choice of pseudo-times may be impossible to achieve due to the real time ordering of reads and writes. As Reed points out, this

consideration is a good reason for making the ordering of events in pseudo-time as similar as possible to their order of execution in real time.

Thus in both schemes, situations can arise in which a transaction cannot be allowed to complete because it has picked a bad time to run. In a locking scheme, the detection of a deadlock requires one of the transactions to be removed and its locks broken. Refinements to the locking protocols can reduce the probability of a deadlock, as shown in Sturgis80. In the pseudo-time scheme, the neater detection of the mis-ordering must be followed by the equally drastic action of cancelling a transaction.

3.2 Faults, Errors and Error Recovery

The inability of a transaction to order itself with respect to others is only one reason for it not being able to complete an intended sequence of changes. It may serve, however, as an introduction to the larger problem of maintaining consistency in spite of various types of malfunction. The general ideas in this section are derived from those of Randell [Randell78a, Randell78b, Randell79], and are presented in the context of a model appropriate to a file server. The aim of this section is to define a general class of errors which a file server may be expected to deal with automatically.

The previous section introduced the idea of a transaction for grouping the changes intended on a set of objects. A transaction makes a set of changes to its write set using the values of the objects in its read set, and some external data.

By taking a more general view, we can model the behaviour of a file server expected by a client. If the client owns n files and indices, then these will represent some abstraction, such as a filing system, a data base, or perhaps a trivial accounting mechanism such as in the initial example. The values of these n objects can be represented as points in an n-dimensional space, each point defining a particular choice of values. Only a small number of these points will represent valid states of the abstraction. In a relational data base organised as one relation in each of n files, for instance, there will be relatively few choices for the file contents which give meaningful states of the data base.

The transactions requested by the client can be represented as state transitions in a finite state machine, if we assume that any potential concurrency is ordered by one of the methods presented in the last section. Each transition takes some initial state of the n

objects to a final state which differs only in a subset of the objects, the write set of the transaction. This simple finite state machine model is shown in figure 3.8. Each transition of the machine is triggered by the arrival of a transaction and some qualifying external date, shown at the tail of the arrow. A transition of the machine causes a state change to new values of the objects, and - as an extension to the earlier definition of a transaction - an output. This output may vary from a simple "done" to the description of an airline ticket to be issued to a customer, and is shown at the head of the arrow.

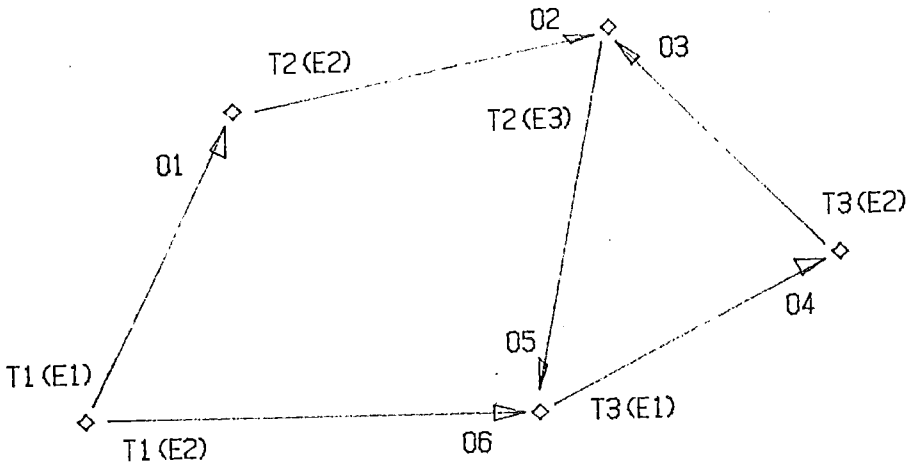

Fig 3.8 Finite State Machine Model

Real file servers do not behave in this simple and predictable fashion. Concurrent transactions may fail to order themselves successfully, power failures can occur and discs may spontaneously destroy themselves, to name only a few deviations from the ideal. These events may cause an invalid state to be entered and incorrect behaviour of the finite state machine. In the following discussion of errors, we shall maintain our optimistic view of transactions. Any transaction which reads a consistent state will produce a consistent state and generate a correct output in the absence of errors.

Describing incorrect behaviour is difficult, because it encompasses every type of behaviour which is not specifically correct. For a particularly simple implementation of the finite state machine, it might be possible to enumerate all the the programs and electronics required to function correctly, and to list the events which might prevent this from happening. For even moderately complex machines, this cannot be done; any better definition of incorrect behaviour than "that which is not correct" requires an exhaustive list of what might cause it.

In very general terms, however, some progress can be made by allowing the finite state machine to behave non-deterministically. Specifically, we will allow two types of random behaviour:

1) In a state S, the application of a transaction T can result in any state. The action of T is governed by a probability density function, in which the ideal result T(S) is only the most likely outcome. There is a finite probability of a transition to any conceivable state.
2) The finite state machine is not constrained to change state only on the arrival of a transaction. In any time interval, there is a finite probability of a spontaneous transition from the current state to any other state.

By comparing the action of this non-deterministic machine with the ideal machine from which it is derived, we can describe its incorrect behaviour. The following definitions are derived from Melliar-Smith75:

Definition: A fault is an invalid transition produced either spontaneously or when a transaction fails to achieve its ideal outcome. A fault produces an error in the incorrect change to the machine state.

Any automatic scheme for dealing with faults must begin with error detection, the signalling of a deviation of the machine's state from that of the ideal machine. This will be based on checking constraints of the set of valid states, either inherent ("there are never more tickets issued than seats available"), or added with explicit redundancy. Once an error has been detected, there are two complementary actions which can be taken. Error recovery removes the effects of a single fault be restoring the machine state to that of its ideal counterpart. Fault treatment attempts to reduce the likelihood of the fault recurring. In terms of the model, we can distinguish two general classes of faults. Transient faults are caused by choosing other than the most likely outcome when sampling from the probability distribution, and will happen from time to time on a statistical basis. Hard faults, on the other hand, are caused by an alteration of the probability distribution, such as might be produced by a short circuit. Fault treatment attempts to prevent faults recurring by changing the probability distribution, and is thus more appropriate for hard faults than for transient faults.

The most satisfactory way of dealing with incorrect behaviour is by eliminating its causes. This is termed fault intolerance by Avizienis [Avizienis78]. Program proving techniques, for instance, may ultimately allow a more justified reliance on program correctness.

Similarly, improvements in technology may make some faults so improbable that they can be ignored.

In the interim before the arrival of perfect hardware and software, however, faults must be accepted as a practical hazard. In these circumstances, replication seems to be the best solution. The action of the ideal machine can be simulated by using a number of identical non-deterministic machines whose faults are independent. Choosing the majority outcome at every stage will give a close approximation to ideal behaviour. What is perhaps better, the dissenters can be immediately identified as faulty, and can either be switched off or reset to a correct state. Replication thus gives particularly simple forms of error detection and error recovery, and simplifies fault treatment by precisely describing the error.

Unfortunately, replication is not the universal solution to all faults. There are two reasons for this. The first is that the replicated copies will be equally susceptible to incomplete input sequences. Where transactions are defined as a sequence of reads and writes, an inability to complete a transaction whether due to communication failure, deadlock, or faults in the client program, will leave all identical machines in the same intermediate state between the initial and final states of the transaction. The second shortcoming of replication is that although it might be possible to make the failure modes of the copies independent, it may not be feasible to do so. Power failures, for instance, may affect all copies of a machine if it is too expensive to give each a private power supply. Similarly, it may be too expensive to write to five disc units simultaneously to eliminate the effects of random changes to the same block on any two of them.

There is thus a pragmatic trade-off in designing error recovery mechanisms. The set of errors must be partitioned into those which can be recovered from automatically at acceptable cost, and the remainder which will cause disaster. This classification is implicit in most of the published work on data base and filing system recovery techniques [Verhofstad78].

Randell discusses two methods of error recovery [Randell79]. Forward error recovery operates by examining the current erroneous state to determine the damage which has been caused by a fault. The erroneous state is then corrected by altering the damaged parts. For clients of a file server, forward error recovery would require reading the files in the write set of a failed transaction to discover the damage which had occurred, and what changes the transaction was attempting to make. Recovery could then be performed either by undoing the transaction's erroneous changes or by completing the remaining ones. An obvious difficulty with forward error recovery is the inference which may be required to determine how the recovery

should be done. This will depend on the particular changes which the failed transaction was attempting and the external data which it was using, and makes forward error recovery unsuitable as a general technique for a file server.

Backward error recovery eliminates most of the problems of inference by saving the previous history of the finite state machine as a list of recovery points. As each transaction completes, the state in which it leaves the machine is added to the list. On detection of an error, the only inference needed is in deciding when the fault occurred. Then, a recovery point previous to that time is chosen and installed as the machine state. If this is not the most recently recorded recovery point, then any transactions performed since then must be rerun to arrive at the correct state.

3.2.1 Practical Error Recovery In Large Systems

The problem of providing automatic error recovery in systems where the machine state is in the order of 10^{10} bits is a special case of the general problem discussed in the previous section.

Systems of this type, such as file servers and data bases, may be characterised as having an enormous number of possible states, and an unpredictable set of transactions to change the state. Forward error recovery may be eliminated as a general technique in these circumstances, because the actions which must be taken depend on the detailed behaviour of the failed transaction. Backwards error recovery is thus universally used in systems with a large amount of state information. To be a practical possibility, however, it depends on rapid error detection to prevent the propagation of errors.

Consider a banking data base. As well as having transactions to calculate interest, and to transfer money between accounts, it might also be able to transfer money out of the system, by crediting a bank account in another country. The consequences of an error in this type of transaction are particularly serious because the output of the transaction, once issued, can not be changed. If the transaction reads erroneous data, perhaps because a previous transaction failed leaving an incorrectly large amount of money in a bank account, then an invalid transfer might occur. Detecting the error at some later stage is insufficient, because even if the transaction is retried with a different result, the money transferred out of the system may not be recoverable.

Thus in general any errors present in the state must be detected before they propagate to the outside world. Before a transaction issues an output, it must be known that the transaction's read set was free of errors and that the output has therefore been correctly calculated. In discussing this problem Davies has said "no process

may be allowed to commit its results with any greater degree of certainty than the process's inputs" [Davies79]. This may be rephrased "in a system which takes irreversible actions, a transaction must not read erroneous data".

To make error recovery possible, therefore, it is necessary to assume that no outputs are generated between the occurrence of a fault and detection of the error. Allowing an output to be generated is thus an assertion that all data read were correct, and that the transaction has constructed a new consistent state. This assertion as called committing the transaction, and is a declaration that no errors will propagate to other transactions as a result of this transaction.

Substantial simplifications result from assuming that rapid error detection prevents error propagation. Since a transaction only commits if its read set is correct and its write set has been changed without error, any detected error has occurred by assumption after the last committed transaction. Thus, the state which should be restored after an error is that produced by the last committed transaction. It is therefore only necessary to retain a single recovery point at any time rather than a list of recovery points. As each transaction commits, its final state can define the current recovery point. This algorithm is now generally known as the two-phase commit protocol [Gray78], though it seems to have first been applied in the filing system of the CAL-TSS operating system [Sturgis74] under the name of intentions lists. The name refers to the separation of a transaction into two phases. In the first phase, when faults may occur, the changes to the state are made tentatively in a reversible manner. Only after the last change has been made without error can the transaction enter its second phase by committing the changes. At the time of committing, two consistent versions of the system state are available, and it must be ensured that the same version is chosen for each of the changed objects. Many of the subtleties in published versions of the two-phase commit protocol are concerned with guaranteeing that the same version is chosen for each object when these are stored on different computers with independent faults connected by an error-prone network [Thomas79, Sturgis80].

In systems with enormous amounts of state information, automatic error recovery remains a substantial technological problem even after error propagation is eliminated by assumption. The reason for this is that recording a copy of the state produced by a transaction is orders of magnitude more costly than executing the transaction itself. In a data base with 10^{10} bits of storage, for instance, an individual transaction may change only a few hundred bits. As a result, copying the entire data base is not an operation to be attempted after every transaction. Rather, recovery points are recorded at infrequent intervals, and thereafter a list of changes made by committed

transactions is maintained. When a transaction commits, a record of changes to the system state which it has made is appended to this journal or audit trail [Gray78]. To recover from an error, it is simply necessary to reinstate the last recovery point, and then apply to it all the changes in the journal.

3.2.2 Interruptions

By assuming that no transaction commits between the occurrence of a fault and detection of the error, recovery mechanisms can be built which will recover automatically from most errors of practical interest. These include losses due to using bad areas of disc storage, and even wholesale destruction of information, such as occur in a disc head crash. An excellent description of a mechanism of this type is to be found in Gray78.

As mentioned above, the design of an error recovery scheme is a matter of balancing the probability of occurrence of errors, the seriousness of their consequences, and the cost of a mechanism for preventing these consequences. In data base applications, the consequences of unrecoverable errors may far outweigh the cost of extra hardware and software to maintain the recovery points and the journal, even if it is exercised only once in the lifetime of the system. In other cases a mechanism which will handle a smaller class of errors more cheaply is appropriate. The Cambridge file server, for instance, was designed for use in a research community in which the cost of losing a week's work every few years was more acceptable than providing expensive hardware to recover from these errors. As a result, a lightweight mechanism which recovers automatically from the large majority of errors was designed, and was supplemented by a discipline of recording a weekly recovery point for the rarer more serious errors.

The process of defining these smaller classes of errors is one of making assumptions which restrict the amount of state information damaged in an error. By doing this, the amount of information needed to recover from the error is limited, and therefore also the storage and work needed to record it.

To make any simplifications on the general technique, it is necessary to exclude spontaneous state transitions. If any part of the state may be damaged spontaneously at any time, then a complete copy of the most recent committed state must always be maintained. This assumption does not prevent the machine from halting at arbitrary times, but it must only refuse to accept further transactions and must not change state.

Similarly, if the state produced by a transaction can be any state whatsoever, there is the possibility for arbitrary damage, and so the need for a complete copy of the most recent committed state. The majority of errors in a working system have a more predictable pattern of damage, however. Damage is usually confined to the objects in the write set of the transaction in progress.

Definition: A fault during the execution of a transaction is an interruption if the state it produces differs from the initial state of the transaction only in the members of the write set of the transaction.

Consider a non-deterministic finite state machine in which the only errors which occur are interruptions, and in which there is no error propagation. This machine has the important property that if an error is detected, then the damage is limited to the members of the write set of the current transaction. Spontaneous transitions, such as those which might be caused by a disc head crash, are assumed not to occur. The information which must be maintained for error recovery is therefore also limited. Specifically,

1) When no transaction is in progress, no errors can be produced. Therefore, no recovery information need be maintained.
2) When a transaction is in progress, it is sufficient to maintain copies of the initial values of its write set. In the event of an interruption, error recovery consists of restoring the write set to its initial state.

Interruptions are thus particularly tractable by automatic means, and the limited amount of information required to deal with them means that maintaining it may not impose severe costs in hardware or execution time.

Interruptions account for a large majority of transient faults which occur in practical systems. These include the states produced by most client crashes, deadlocks, power and communication failures and the software and hardware failures known as "system crashes" and in general all events which cause a transaction to be only partially executed. They do not include faults due to incorrect programs which alter one object when told to change another, spontaneous losses of information, and any number of more catastrophic events. In the remainder of this thesis, it will be explicitly assumed that the only faults with which the file server will deal automatically are interruptions.

3.3 The Storage Model

The finite state machine model presented in the last section is not particular to the behaviour of a file server as seen by its clients. More generally, it applies to any storage abstraction which is subject to occasional erratic behaviour. In this section, we will apply it to a model of disc storage, and use it to compare a number of error recovery techniques.

A client of the file server specifies a transaction as a sequence of primitive file operations. Within the file server, each operation is not primitive, but consists of a transaction on the set of disc blocks which define the file or index. Again, a disc write operation is itself not atomic when considered from the point of view of the disc hardware, but consists of a sequence of bit changes. Thus, there is a natural hierarchy of finite state machines which reflects the levels of abstraction in the file server. A primitive write operation at one level is executed as a complete transaction at the next lower level.

When an interruption occurs, it will affect the operation of each machine in the hierarchy. A power failure, for instance, may interrupt a client transaction, the file write it was engaged in, and the disc transfer which was in progress. It may, however, occur when the disc is temporarily idle. In general, halting the hierarchy of machines at an arbitrary time will find the higher levels in mid-transaction, but may find the lower levels between transactions, and therefore in a consistent state.

The difficulty of error recovery at each level is determined by the behaviour of the lower levels. The models of disc storage used implicitly by Rappaport [Rappaport75] and Giordano [Giordano76] assume that disc writes are atomic with respect to interruptions. All interruptions at higher levels occur either before a disc transfer starts, or after it completes, but never during it. This strong assumption makes error recovery at the file level simpler, since no interruption can cause the loss of a disc block.

Lorie [Lorie77] permits interruptions at the disc level and the possibility of loss of information. His scheme, therefore, maintains redundant information on disc to allow reconstruction in the event of an interruption during a disc transfer.

Sturgis *et al.* [Sturgis80] are explicit about the assumed characteristics of disc-level transactions. They assume that disc storage has the "weak atomic" property, in that a disc write can have three outcomes: 1) success, 2) no effect, 3) detectable corruption, and that success is only reported if the transfer occurred correctly.

Contents of disc blocks are also explicitly assumed not to deteriorate once written successfully. These conditions are equivalent to restricting the faults allowed to interruptions as defined in this chapter. Sturgis et al. also describe a method for building stable storage from disc storage. This is done by recording disc blocks in two copies so that each write to stable storage is performed in two disc writes. Recovery from interruptions is automatic, and relies on the fact that at least one of the two disc copies of a block of stable storage will be readable at any time. Transactions at higher levels can therefore assume that writes to stable storage are atomic with respect to interruptions.

The model of disc storage assumed by the Cambridge file server also restricts disc errors to interruptions. The addition of the required redundancy to allow automatic error recovery will form the subject of the next chapter.

3.4 Interactions Between Concurrency and Error Recovery

The necessity for a backward error recovery mechanism has consequences on the ordering of transactions for control of concurrency. Consider two transactions T_1 and T_2 such that $W(T_1)$ and $R(T_2)$ overlap. Suppose that at the time the overlap was detected during the concurrent execution of T_1 and T_2, the scheduling decision $T_1 < T_2$ was made. Thus, changes to T_1's write set are to be made visible to T_2. Given the possibility of errors during the execution of T_1, however, the decision $T_1 < T_2$ cannot be made unambiguously. T_1 may be erased completely and have all its writes undone during error recovery, so that judgement on the decision must be suspended. Under a locking protocol, for instance, if T_1 unlocks an object in $R(T_2)$, T_2 must not be permitted to read it while T_1 is still susceptible to errors. Otherwise, even though $W(T_1)$ is backed up, an inconsistency will be propagated to $W(T_2)$ unless T_2 is also subjected to backwards error recovery. Thus in general, whenever a relationship $T_1 < T_2$ is decided upon, it must not be allowed to occur until T_1 has committed; T_2 must wait until T_1 is known to have run without error before reading its results. With this refinement, backwards error recovery of T_1 has no effects of T_2 because no decision will have been taken on dependencies between them.

This method is followed in Reed's pseudo-time mechanism. Reading an object version is delayed if the transaction which created the version has not yet committed. The version will ultimately be read if the transaction commits successfully, but otherwise the next older version in the object history will be returned.

Eswaran et al. explicitly ignore the effects of errors, and the locking protocol they define must be elaborated slightly to deal with them. They require that a transaction be two-phase. In its first phase, the transaction may acquire locks, but not release them. In the second phase, it must release locks but acquire no new ones. Clearly, releasing a lock allows results of the transaction to become visible to other transactions, so that before releasing its first lock, the transaction must have passed the point of commitment. In particular, all writes must have been completed by this point if errors can still cause the transaction to fail. Otherwise, the transaction will have exposed parts of its result state to successors, but will be unable to create the complete and consistent write set.

Thus to cope with interruptions as well as concurrency, a transaction must neither acquire new locks nor write any objects after the first object in the write set has been unlocked. The unlocking of this first object marks the point of commitment of the transaction, and following it, the transaction may only (uselessly) read from objects and release further locks.

3.5 Atomic Transactions

The abstraction we have now described is often called an atomic transaction [Gray78, Randell78]. This phrase attempts to capture the indivisibility of a transaction; an atomic transaction either runs to completion or has no effect whatsoever and no intermediate states are visible outside it. The notion of an atomic transaction seems the closest possible to the mathematical concept of a function, which is instantaneous and arbitrarily complex, given fallible hardware.

In the light of the foregoing discussion it is more accurate to say that a transaction is atomic with respect to interruptions. The mechanisms we have been discussing here provide atomicity with respect to concurrency, and atomicity with respect to interruptions.

3.6 A Transaction Abstraction For the File Server

This general excursion into the problem of assisting clients of a file server in maintaining consistency in a complex data structure has outlined the desirable characteristics of a transaction abstraction. The minimum requirements seem to be to provide for atomicity with respect to concurrency and interruptions. In the event, however, the abstraction actually provided was less general than the ideal, and

allows only a single object in the write set of a transaction. This is a result of several factors: pragmatic assessment of the needs of the client operating systems, implementation uncertainties with a complete mechanism, and the author's lack of understanding of the general problem.

Concurrency control is based on conventional interlocks. The pseudo-time mechanism is very attractive, but as yet untested in an environment requiring very rapid response. In the absence of a detailed implementation, it must remain a valuable contribution to the understanding of the consistency problem rather than a pattern to be imitated.

In the file server, there are in fact two kinds of unique identifiers. The kind already introduced are permanent unique identifiers (PUIDs). A PUID is created with an object, and remains bound to it for the object's lifetime; it may be thought of as a capability for the object. A temporary unique identifier (TUID), is a capability for an interlocked file, and is created by the open request:

open (PUID object, BOOL for writing) TUID

The open request attempts to establish an interlock on the selected file or index according to a multiple-reader-or-single-writer discipline. A TUID identifies both the object and the particular interlock set on it; when several requests are made to open a file for reading, different TUIDs will be returned for each request. These TUIDs denote distinct, but overlapping interlocks on the same object.

Once a TUID has been obtained, it can be used in all the file and index operations listed in chapter two. All write operations, however, make changes reversibly, so that in the event of an interruption, an object opened for writing reverts to its state at the time of opening. A TUID cannot, however, be used in a retain operation, since it would not be sensible to retain a name for a temporary object in a permanent one.

TUIDs are destroyed by the close operation, which is used to remove the interlock and to commit or abort the transaction on the object.

close (TUID object, BOOL commit)

If commit is TRUE, then any changes made while the object was open are made irreversible in a single atomic operation. If commit is FALSE, then all these changes are removed. The TUID supplied as argument is then invalidated by removing the interlock it denotes. It is up to the client to decide whether or not to commit the transaction, and under normal circumstances, a value of TRUE is supplied only if all operations in the transaction were successful, or if any errors

detected were successfully corrected.

Interlocks, of course, must not be allowed to persist indefinitely, and the file server must have some means of detecting and removing the interlocks set by a failed transaction. To do this, when a TUID is first generated, a timer is set on it for some small number of minutes. Whenever this TUID is received from a client in a request, the counter is reset to its initial value. If it ever expires, the file server decrees that the transaction to which it belongs is dead, and generates a close operation with commit = FALSE. If the transaction is still alive, the client will find that its TUID is now invalid, and it should under normal circumstances close all remaining open objects with commit = FALSE. Note that nothing prevents a looping transaction from holding a lock indefinitely. An alternative strategy, whereby locks become vulnerable to pre-emption after some initial period, is used in the Juniper file server [Sturgis80].

For efficiency reasons, a client may wish to have an object move through a sequence of consistent states in a transaction without having to compete each time for the object interlock. This is similar to the idea of checkpointing in databases, and limits backward error recovery to the most recent consistent state rather than the initial state found by the transaction. An error thus causes less of the work done by a long transaction to be discarded. To define an intermediate consistent state, the ensure command is used.

ensure (TUID object, BOOL commit)

This operation either commits or removes all changes to the object made since the last ensure or close command. It is identical to a close operation except that it does not invalidate the TUID supplied as its argument.

The three transaction operations introduced here allow a client to construct transactions in the described in section 3.3. A transaction begins by open operations on the members of the read set and the single object in the write set. The appropriate file or index write operations then change the object open for writing to its new value. Finally, the interlocks are released by a sequence of close operations, in any order. The transaction commits when the object open for writing is closed with commit = TRUE.

Note that nothing prevents a client from opening several objects for writing in a transaction, thus having a number of elements in the write set. However, because of the semantics of the close operation, it is impossible to make an atomic change to all of them. The close requests must be made in some order, and if the client were to crash after closing some of the objects with commit = TRUE, the file server would eventually close the remainder with commit = FALSE.

This restriction has not in practice led to any serious difficulties. Most of the client operating system transactions involve a single object in the write set, and those which do not, such as directory updates in the CAP [Dellar80b] have been adequately implemented as a sequence of transactions.

Careful ordering would not be sufficient for data base clients, however. In a transfer of money between accounts using a debit transaction and a credit transaction, there would be an instant at which there was either an excess or an insufficiency of money in the data base. For such cases, it would be possible to build a client-based multiple-file transaction method following Paxton's example [Paxton79]. A file could be used to hold the changes intended by the transaction, by recording the adjusted balances of the two accounts. At each restart of the data base client, this "intentions" file would be examined, and any updates found in it applied before resuming normal service. Thus a multiple-object transaction mechanism could be bootstrapped from the simpler mechanism by using a special file to hold higher level intentions. The same result can also be achieved more efficiently by extending the actual file server transaction mechanism to many objects in the write set, as will be described in chapter six.

3.7 Repeatability

As opposed to the file and index operations defined in chapter two, the operations just defined for the control of transactions are not obviously repeatable. There would seem to be a problem if the reply to an open, ensure or close request is lost in the network.

The case of the open request is merely inconvenient. If the TUID contained in the reply to the open request is lost, then a retry of the operation will fail if the interlock is for writing. This does not imply a loss of synchronisation between client and server, as would be the case in the loss of a reply to "withdraw 50 pounds from account 3751", because it is clearly safe to retry the open request. If later attempts fail, then the client must wait. Eventually, the file server will time out the lost TUID, and the next attempt will succeed. It is only tiresome that the loss of a reply can cause a delay of some minutes.

A solution to this problem would be to admit defeat in the attempt for complete repeatability by introducing a simple virtual circuit for

the special case of open requests. If the file server records the machine number and port* number of each client requesting an open operation, then a retry by the same machine specifying the same reply port number could be recognised as such and the same TUID could then be returned.

Whether this addition is worthwhile is to be judged on the basis of frequency of occurrence of the error. If loss of messages in the network is about as likely as a crash by a client machine - in which all TUIDs will be lost in any case - then it is probably not worthwhile. In the actual implementation, a timeout of TUIDs after a few minutes has proved sufficiently short not to be troublesome.

The case of ensure and close is more fundamental, however. If the reply to a close operation is lost, then the client will not be able to determine whether the transaction committed or aborted. If a repetition of the request is successful, then all is well; the initial request was lost. If a repetition eventually produces the reply "invalid UID", however, then this may either be because the first attempt has succeeded, or because the file server decided to break the interlock after a timeout. (In the actual implementation, a file server crash also causes interlocks to be broken. This could be avoided - at a cost - by recording interlocks in non-volatile storage. It does not make the current problem fundamentally more difficult, however.) Worse, the client cannot determine whether the transaction committed by reading the object. It is possible that some other client has since modified the object while the first client was waiting for the lost reply to its request.

It is important to note that this essential uncertainty is not due to the separation of client and server by a local network. Even in a single machine implementation, if the procedure call to commit a transaction never finishes due to a crash, the same uncertainty arises. The user at a terminal in the centralised case, and the client program in the distributed case both need to know whether the crash occurred before or after the atomic commit action. Unless the transaction mechanism guarantees

1) never to time out interlocks, and
2) always to re-establish interlocks in effect at the time of an interruption

then examining the object in question will not necessarily solve the problem due to the possible effects of intervening transactions. If the transaction mechanism <u>does</u> guarantee these two conditions, however, then lost and unbreakable interlocks are possible.

* essentially a process identifier in the client machine, whose use is defined in chapter 5.

This situation seems inevitable in storage systems which do not maintain histories of changes. In Reed's scheme, a client in this predicament would always be able to enquire "was a version of object X created at pseudo-time T?". If the transaction at pseudo-time T aborted, then all of its object versions would have been deleted, and the answer would be no. Because no other transaction can affect the version at time T but can only insert versions at other pseudo-times, so the uncertainty can always be resolved.

Adding a certain amount of historical knowledge to a system which does not explicitly maintain object versions can help, however. If the file server maintains a list of recent TUIDs together with a record of the decision to commit or abort which it has taken, then clients can be allowed to enquire about the fate of a transaction. A mechanism much like this is used in the Juniper file server [Sturgis80]. Even so, to be effective, the method requires recording the TUID of a transaction in a place which will survive all errors which are meant to be recoverable, so that the correct question can still be asked when service is resumed.

3.8 Optimisations

The fact that the file server was designed for use by operating systems caused two simplifications to be made to the transaction mechanism.

In a virtual memory system such as that of the CAP, the full transaction mechanism seems rather elaborate for the task at hand. The virtual memory system may be expected to organise its own interlocks, so that when it swaps a file to or from the server, no interference is possible with other clients, or within its set of processes. Engaging in a full transaction exchange such as

```
tuid := open (file, TRUE)
write file (tuid, offset, length, data)
close (tuid, TRUE)
```

therefore involves two essentially useless commands. The network transmission time of these commands is liable to be negligible, but scheduling delays in both client and server to prepare for transmission and reception are probably more costly. If the virtual memory manager process must run three times in quick succession for every segment transfer, this may contribute significantly to the load it imposes on its processor.

For this reason the file server interface provides a simple shorthand for the common case. Whenever a file or index request arrives specifying a PUID instead of a TUID, it is assumed that the

client means to perform a transaction consisting of a single operation. The file server then brackets the operation by the appropriate open and close operations, which are generated internally. Thus the full transaction mechanism is in fact performed, and the transaction will be refused if a conflicting interlock is outstanding. With this optimisation, PUIDs and TUIDs are almost always interchangeable in any file server operation. It does not of course make sense to retain or open a TUID, nor to ensure or close a PUID.

The second beneficial effect of designing a file server for operating systems is more fundamental. It is often the case that precautions for correct behaviour in the event of an error are completely unnecessary. Compilation, for example, is often performed in two transactions. The first is the actual run of the compiler; its read set contains the source code and perhaps some environment description files, and its write set consists of a file created expressly to hold the translated binary program. The second transaction is a directory update which replaces the old binary version by the one just made by the compiler. The important point about this example is that no error recovery is required by the compiler transaction. If an interruption occurs, the inconsistent output file will simply be discarded when the compilation is repeated. Any extra expense involved in making sure that the binary file can always revert to its initial empty state is wasted.

The second directory transaction shows the opposite case. Since the correct functioning of the operating system depends on the consistency of its directories, all changes to directories must be made atomically. When inserting the new binary file, for instance, no error should expose a half-written directory.

This example leads to the general observation that a transaction may know that any inconsistencies which might be produced in its write set are not dangerous. This may be because objects in the write set will be discarded if the transaction fails, or because they contain information local to the transaction which will never be read by any other, or simply because the higher levels of software are prepared to deal with inconsistencies, perhaps by typing "bad binary file". This information would not be important if atomic updates could be made so cheaply as to make considerations of cost unimportant, but this is not the case in any reported atomic transaction mechanism. A file server which imposes a blanket guarantee of error recovery for every transaction will thus be doing its clients a disservice if there is a perceptible performance cost. The pitfall to be avoided here is common in operating system design; the client of a sophisticated service should not be obliged to pay for facilities he does not use.

To the author's knowledge, the Cambridge file server is unique in allowing clients to specify whether precautions against interruptions are to be taken or not. This is done at the time of creation of a file, on the assumption that the need for error recovery is governed by the information to be contained in the file. The create file operation takes an extra Boolean argument which specifies whether the file is to be normal or special. Special files are those which have been described to this point. They have the guarantee that a transaction error will always cause the file to revert to its last consistent state. Normal files do not have this guarantee and a transaction error can leave a file in a partially written state. The performance advantage in updating normal over special files is due to the file server's knowledge that for a normal file, modifications can be made irreversibly as they are requested without the need to make them conditional on the transaction committing.

Once normal and special files have been created, there is no distinction in the way they are used. Clients are expected to know whether a file is normal or special by context, if this is of interest to them. The only way to observe the difference at the file server interface would be to open a file for writing, write to it, and then close it with commit = FALSE. A special file would be in its original state, but the normal file might be modified.

In the light of experience, the distinction between normal and special files has proved to be very useful. The large majority of files kept by the file server are normal, with consequent gains in performance. The special files mostly contain the files used to hold directories for the client operating systems. Indices are always special, the choice being made by the file server rather than by the client. The reason for this is that indices are used in storage control, as described in the previous chapter, and an error during an index update must not cause any objects to become detached from the root index.

In summary the complete file server interface can now be presented. The type UID specifies a value which is either a PUID or a TUID.

3.8.1 File Operations

```
create file (UID index, INT entry, INT size,
             INT uninit, BOOL special) PUID

read file (UID file, INT start, INT length) DATA

write file (UID file, INT start, INT length, DATA to write)

read file size (UID file) INT
```

change file size (UID file, INT new size)

3.8.2 Index Operations

create index (UID index, INT entry, INT size) PUID

retrieve (UID index, INT entry) UID

delete (UID index, INT entry)

retain (UID index, INT entry, PUID object)

read index size (UID index) INT

change index size (UID index, INT new size)

3.8.3 Transaction Operations

open (PUID object, BOOL for writing) TUID

ensure (TUID object, BOOL commit)

close (TUID object, BOOL commit)

Chapter 4

Implementation Issues

The similarity of files and indices suggests that both can be implemented out of a common underlying storage abstraction. We require this abstraction to represent both small and large files efficiently, to provide random access to words in files and UIDs in indices, and to allow large tracts of material to be read and written sequentially at high speed. It must also be designed so that updates can be done atomically or not, since all indices but only some files require atomic updates.

This storage abstraction obviously incorporates the essential implementation decisions of the file server, and in this chapter, the reasons for the structure chosen, and their consequences, will be discussed. The central topic will be the atomic transaction mechanism, and the representation of information in such a way as to guarantee its preservation over interruptions.

In the file server the abstraction provided at this internal interface is called an object. An object is simply a storage container consisting of a sequence of blocks of fixed size whose contents are uninterpreted. Each object has a number of attributes. These include a use count of index references to the object, the maximum accessible block number, and the maximum word offset accessible within the final block. Also stored with each object is the uninitialised data value which is returned when words which have never been written are read. For objects used as files, this value is set by the client at the time of creation of the file. For indices, it is always set by the index manager.

4.1 Representation of Objects

Each object is represented on disc as a tree of disc blocks. The tree can have one, two or three levels depending on the size of the object. When first created, every object consists of a single disc block as shown in fig 4.1. This block contains the object's attributes in a few reserved words at the beginning, and the remainder is cleared to the uninitialised store value. This is done even for objects created with a large maximum size.

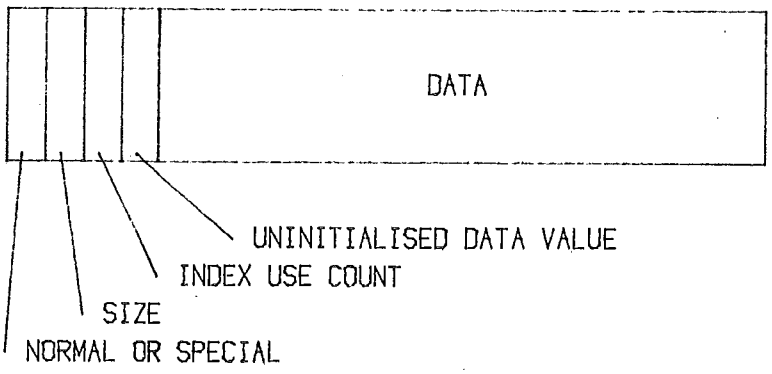

Fig 4.1 One-level Objects

As the object is used, transactions may attempt to write beyond the limit of allocation, but within the maximum size. As soon as this happens, the data part of the root block is copied to a new disc block, and the root block is used to hold an array of disc addresses pointing to the leaves of what is now a two-level tree. The attribute information remains in the first words of the root block. From this point onwards, update operations may cause additional blocks to be allocated as they are written, and their disc addresses will be inserted in the corresponding entries of the root block. This change to the tree structure is shown in figure 4.2.

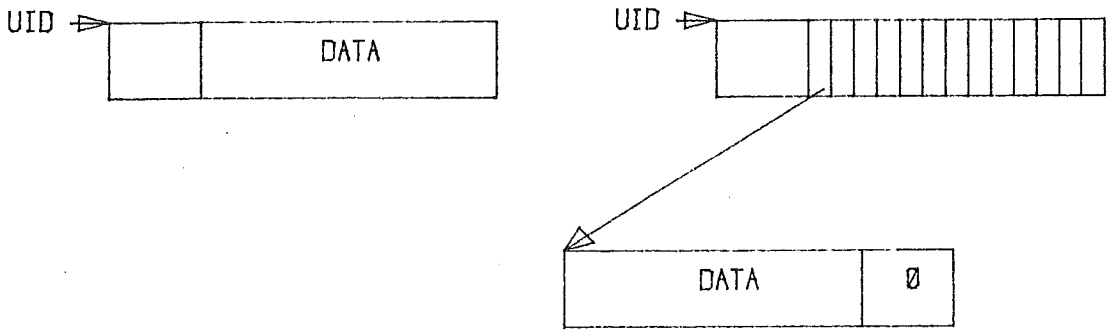

Fig 4.2 One- to Two-level Object Change

Again, at any time it is possible for a transaction to attempt to write to a block whose number is greater than the number of slots in the root block but less than the object's maximum block number. To handle this, an automatic switch is made from a two- to a three-level tree. The disc addresses in the root block are again copied to a new disc block and the root block is changed to have its first entry point to this new block, and all other entries empty. Accesses to data blocks of the tree must now be made through two levels of map blocks, as shown in figure 4.3.

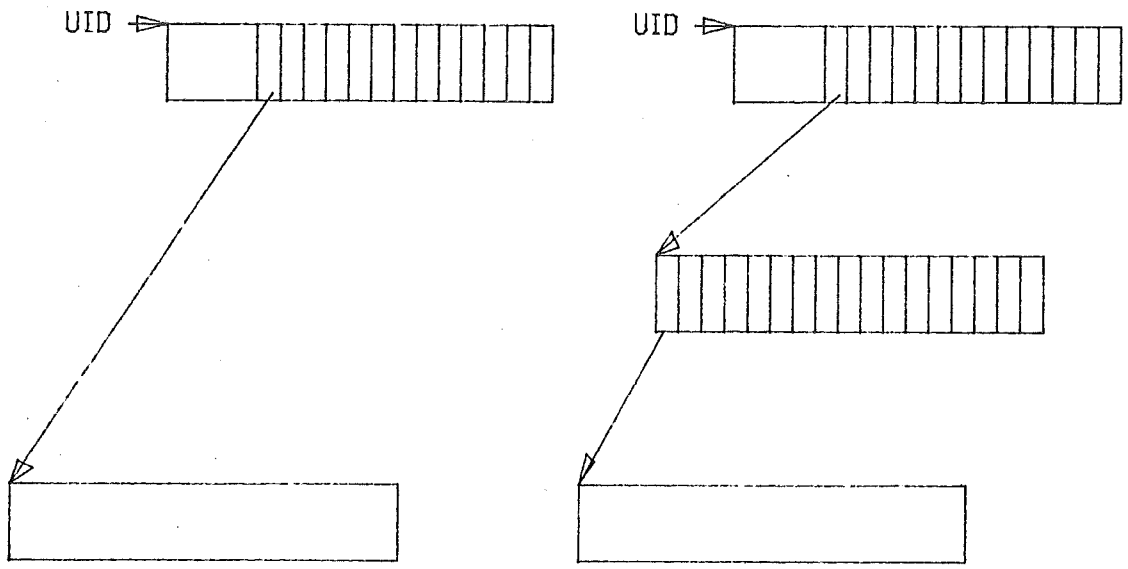

Fig 4.3 Two- to Three-level Object Change

This representation of objects is based on a number of considerations. A file server underlying a general purpose operating system such as that of the CAP will be required to maintain a substantial number of objects in existence, many of which are very small. An examination of the CAP filing system's original implementation on a local disc showed that of the 2000 or so objects maintained, 33% occupied less than 1024 bytes and 27% occupied less than 512 bytes. Thus, the unit of allocation was required to be small, on the order of the disc block rather than the track or the cylinder. The actual implementation uses two sizes of disc blocks. Leaf blocks of the tree are 2048 bytes in size, and the upper-level map blocks of the tree are 484 bytes long. At the time of creation, an object is always allocated a small block as the root of the object tree. Small and large blocks are scattered uniformly over the disc surface, in the ratio of about one small block to two large blocks. The intention of this strategy is to use large blocks only for the data of two- or three-level objects, and to be able to use a large block if a request for a small block cannot be satisfied. The maximum possible file size with this organisation is about 28Mbytes.

When creating a file, it is often not known how much space will be needed. A compiler, for instance, will have no idea at the start of a compilation of the size needed to hold the translated program. Any estimates of size given at the time of creation of a file should probably not cause allocation of the requested amount of space since in many cases this estimate may be wrong by orders of magnitude. Instead, it is much more economical in space to allocate blocks to a file only as they are needed. Until a block of an object is actually

written by a client, it can be marked not present in the object tree. Any attempt to read the block will simply return a block's worth of the uninitialised store value. A tree structure is appropriate here because blocks can be allocated one by one to allow sparse files to be represented efficiently. Allocating block n does not require allocating all blocks less than n, but only the map blocks on the path from the root of the tree. Furthermore, a tree structure allows the .apid random and sequential access required by file transfers.

The decision to allow virtual sizes which do not reflect actual allocations is undoubtedly convenient for clients, but does have performance implications. During a file write it will be necessary to allocate blocks as data arrive from the client. The details of accepting a transfer of indefinite length at high speed as described in section 5.11 already require enough bandwidth that the additional search for free blocks may reduce the transfer rate. This effect is largely governed by the distribution of free blocks over the disc surface.

4.2 Resolution of UIDs

The set of objects maintained by the file server consists of a rather large number of object trees, and the sheer size of this number makes resolution of unique identifiers a potential problem. The file server must maintain the binding between each UID and its tree representation, and it must be able to find the tree quickly given its unique identifier.

Perhaps the obvious way to resolve a UID is to look it up in a large table, much as was done with Hydra capabilities [Wulf74]. The table could keep the disc address of the root of the object tree and perhaps any other pertinent information such as its size and index use count.

There are two problems with this method. With thousands of entries the table will be very large and certainly too large to keep entirely in fast store. A cache can be used to arrange that some parts of the table can be kept in fast store while most of it remains on disc, but with largely random UIDs, there seems little chance of obtaining the locality of reference which would make this arrangement acceptably efficient. It would be likely to become a performance bottleneck.

The second problem with this central table is its vital importance to the functioning of the file server. Any damage to the table would in all likelihood cause the loss of a number of objects. To avoid this clearly unacceptable possibility, the table must be stored as a special file whose existence is known only to the file server. Object

creation is therefore likely to become expensive, with atomic updates required to both the UID table and an index.

The advantages of a central table are twofold. Firstly, it collects all UIDs together in a single structure so that a garbage collector, for example, could discover its universe of objects very easily. Secondly, it allows objects to be moved. Because each access to an object passes through the UID table, an object can be moved by creating a new copy elsewhere and then changing the UID table binding for it.

In a file server, the usefulness of moving objects is not convincing. Janson [Janson76] describes a mechanism in Multics for moving a file to a new disc pack when its current pack becomes full. An equally consistent approach is to treat this as an error, and to abort a transaction which detects a full pack. Or, if objects are moved only rarely, it would be acceptable to leave a "forwarding address" at the original location.

If the inability to move objects is accepted, then a very simple approach can be taken to the resolution of UIDs. A UID can consist of a disc address concatenated with a random number. To resolve a UID, it is only necessary to read the block at the disc address and to check that it is the root of an object tree. If the object's UID is also stored in the root block of every object, then the UID supplied by the client can be validated by a simple comparison*. In the large majority of cases where the UIDs match, the object is directly accessible through its root block. UIDs containing embedded disc addresses are similar to names with "hints" [Lampson74]; every name has an associated address where the object is likely to be found. The difference is that a file server object is to be found at the address given in its UID or not at all, and thus there need be no other method for resolving the UID if the "hint" fails. Figure 4.4 shows the format of file server unique identifiers.

* The UID is actually stored, and validated on each access, in a cylinder map introduced in the next section.

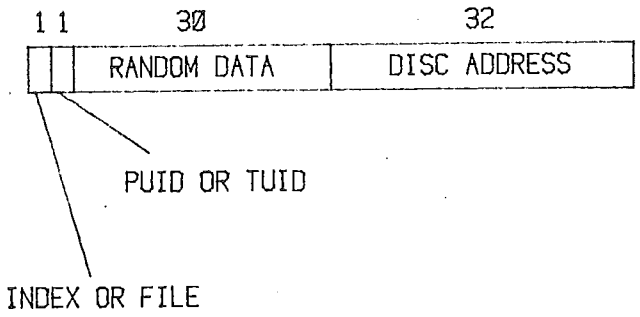

Fig 4.4 Unique Identifiers

4.3 Consistency of Disc Information

The set of object trees is the structure used by the file server to find information quickly. Another structure must be maintained to describe the pool of free blocks not part of any object. This allocation table, often implemented as a bit map indexed by block address, indicates whether each block is in use or free for allocation. Allocating a block involves searching the allocation table for the nearest free block to some optimal position, marking the block in use, and inserting its address in an object tree. Deallocating a block is done by erasing an entry in an object tree and marking the block free in the allocation table.

A consistency problem arises immediately when considering allocating and deallocating disc blocks in the manner just described. The difficulty is caused by the fact that the allocation table is redundant, and is needed only for efficiency of allocation. If there were no allocation table it would be possible - though so inefficient as to be impractical - to choose a block by picking a candidate and then scanning all the object trees to find out if it already belonged to some object. The allocation table exists only to describe disc blocks which are not part of any object tree, and must at all times indicate that only blocks in no object tree are free for allocation.

In the normal course of events, the object trees and the allocation table will be kept in step as blocks are used and freed. However, interruptions can occur during the allocation or deallocation sequences, in which case the object trees and the allocation table will disagree about the allocation state of a block. Two types of disagreement are possible. A block can appear in no object map but be marked in use, or a block can appear in an object map but still be marked free. If allocation and deallocation are performed as

described above, only errors of the first kind - clearly less serious than those of the second - can occur. Blocks will occasionally fall out of use because they are erroneously marked as belonging to some object, but no block will be allocated twice.

There are two fundamental approaches to ensuring that these structures stay synchronised. Either a mechanism must be provided to update both atomically or one must be designated the authority and the second redundant to be corrected by periodic recomputation.

If one of the structures is assumed to be authoritative, and one redundant, then an economical solution is possible. From time to time, the allocation table would be recomputed by enumerating all known object maps and marking blocks in them as used. This could be done either when convenient, if the accumulated errors merely caused free blocks to be considered in use, or whenever an unsafe state might have been produced as a result of a interruption.

Many operating systems use this technique to ensure consistency by recomputing the allocation tables on each restart. The complete scan of all known objects which is necessary also gives added assurances about the correctness of the structures maintained.

An assumption in the above discussion has been that changes to an object tree or to an allocation table are atomic, and that interruptions in the allocation and deallocation sequences will occur only between these changes. In reality, all object trees and the allocation tables will be kept on disc because there will be so much structural information that it cannot all be held in main store. Changing one of these structures is done by reading the appropriate disc block into volatile memory, modifying it there and writing it back to disc. Since an interruption causes the loss of the contents of volatile memory, the essential problem of consistency is to maintain the disc copies of the structures in step. There is a particularly dangerous class of interruptions, therefore, which occur while a disc block is in the process of being written to disc. These interruptions can cause the block to become unreadable so that it contains neither its old contents nor its intended new contents. In the present situation, if an interruption were to occur while writing out one of the map blocks in the object tree, the most likely result would be the loss of all the blocks pointed at by the block, and the partial or complete destruction of the object to which it belonged.

If the contents of a disc block must not be lost even given a power failure during a transfer to it, then there are severe restrictions which must be obeyed when the block is to be written. A transfer can only be permitted if either the block's old contents or its intended contents can be reconstructed in the event of an error. In other words, the block must be redundant. Since we assume the loss of all volatile store in an interruption, the block must be recomputable as a

function only of the contents of other disc blocks.

This restriction is of necessity expensive to observe. Having written a disc block whose old contents were reconstructible, it is necessary to make the new contents reconstructible before allowing a further update to the block. This will require at least one more disc transfer since all disc blocks must be redundant in terms only of other blocks. Thus maintaining the redundancy of disc blocks will require at least a doubling of the number of disc transfers performed. If this is done for every disc block of information, the potential bandwidth of the file server will be halved. This is a heavy price to pay for the ability to recover from a relatively rare form of interruption. In the Cambridge file server this expense is reduced by defining three conditions under which disc blocks may be written.

1) Data blocks of normal objects have no precautions taken to avoid their corruption. They can be written without regard to the possibility of reconstruction in the event of an interruption.
2) Any block which will be returned to the free pool after an interruption can be written without special precautions.
3) All other blocks may be written only if their old contents or their intended contents can be recalculated from the current contents of other disc blocks.

These three rules decide how updates are made to data blocks of objects, to the pointer blocks of object trees, and to the blocks of the allocation table.

Data block updates are governed by whether the object was created as special or normal. For normal objects, data blocks can be written in place since the client has explicitly defined that no precautions need be taken against loss of the contents of the object. For special objects, the risk of losing a data block cannot be accepted. Data blocks are therefore changed by writing the new contents to a free block, and then replacing the old block's address in the object tree with the address of the new block. Writing the contents of a previously unoccupied tree position in both normal and special objects is also done in this way; a free block is chosen, the new contents are written to it, and only then is its address inserted into the object tree. In both the above cases, a failure when writing the new block will cause it to be returned to the free pool on recovery.

There remains the method for changing map blocks in an object tree and the blocks of the allocation table. As presented to this point, there is only partial redundancy between these structures. The bit map can be recomputed since it describes all blocks not contained in any object tree. According to the third condition given above, it is thus safe to allow blocks of the bit map to be written in place.

The object map blocks, however, are not redundant, because there is no structure which could be used to recompute them. Serious errors are possible whenever a map block must be be changed to hold a new disc address.

One possible solution to this problem is used in the Juniper file server [Sturgis80] and was described in section 3.3. Blocks which must always be redundant can be allocated two permanent copies which are always written in the same order. The second copy is only written if the first one is known to have been written successfully. Since an interruption can corrupt at most one disc block, there is always at least one readable copy in each block pair.

An alternative mechanism is used in the Cambridge file server. The allocation table contains not just a single bit per entry indicating whether the block is in use or not, but a description of the current use of the block. For each block in use, the entry contains the UID of the object to which it belongs. It also identifies the position in the tree occupied by the block by a pair of numbers, the level number within the tree and the sequence number within the level. As a result of this increase in the amount of information contained in the allocation table, the table is subdivided into one table per cylinder for each disc. These Cylinder maps are indexed by sector number and define the use of each block on the cylinder. Figure 4.5 shows the format of a cylinder map. Each cylinder map fits into a single disc block of 256 words*.

* Figure 4.5 shows each cylinder map entry containing the UID of the owner object. In practice, space is saved by storing only half the UID in each entry. Root block entries contain the random half of the UID, and entries for other blocks contain only the disc address of the root block. This is logically equivalent to storing the complete UID, because given a disc address it is possible to find the UID of its owner, perhaps by consulting two cylinder map entries. This optimisation reduces entries to four words each, but is ignored here in the interest of simplicity.

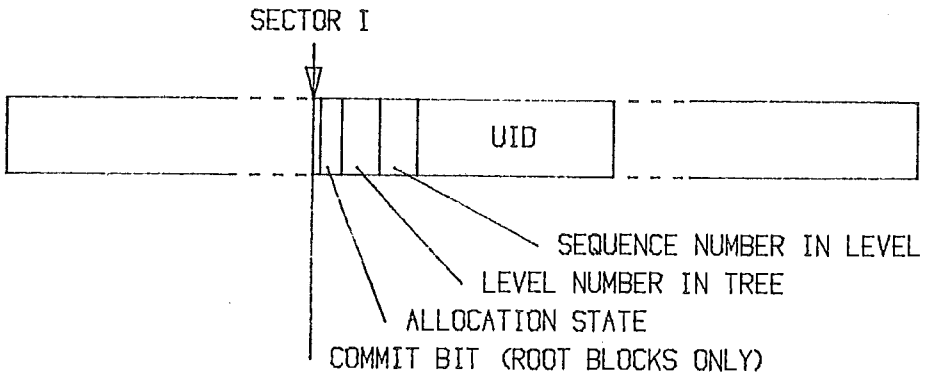

Fig 4.5 Cylinder Map Format

Because of the added information in the cylinder maps, all object map blocks in the object trees are redundant. If an object map block is corrupted during a disc transfer, then by scanning the cylinder maps, it is possible to find all blocks which were its children in the tree and thus to reconstruct the disc addresses which it held. As previously, a cylinder map entry can also be reconstructed by beginning at the root index, scanning all indices to find all valid objects, and scanning all object trees to enumerate the blocks in use. Any blocks not visited during the above sequence can be marked free in the cylinder map. The necessity to be able to find all object trees in the event of losing a cylinder map is one of the reasons why indices must be held in special objects.

A brief comparison with the pair-redundant strategy shows that the cylinder map form of redundancy is more efficient given the tree structure chosen for objects. In a pair-redundant scheme, it would be necessary to duplicate only the root block of each object tree. Changes to all but the root block of an object could be made by writing to a new block and changing the disc address in the parent. The root block would be written in place, because its address is embedded in the object's UID, but in two copies to avoid loss by an interruption during a disc transfer. Thus, each object on a cylinder would require an extra block for the second copy of the root block, which is substantially more than the one cylinder map needed in the scheme proposed here.

4.4 Transaction Sequencing

By introducing an effective duplication of the structure of all information maintained - once in the hierarchy of object trees, and once in the list of cylinder maps - it is possible to guarantee that

no interruption, even during a disc transfer, can destroy essential information. This is, however, only a part of what is needed. These assurances must be used to build a mechanism for making atomic transactions. A write operation to a special file, for instance, may modify the contents of tens or hundreds of blocks of the file, and the changes to these blocks must be synchronised so that all are made or none are.

The method used by the file server is based on the ideas that a transaction must create a complete list of intentions to change an object before it is allowed to make the changes [Sturgis74]. Instead of having just two states, blocks in the file server can be in one of four allocation states. In addition to the two conventional states allocated and deallocated, there are two additional halfway states known as intending-to-allocate and intending-to-deallocate. These intermediate states express an intention to change the allocation state of a block if the transaction requesting the change commits. A block which is marked intending-to-allocate has been tentatively assigned to an object. The cylinder map entry for this block defines the object, and the position within the tree which the block should occupy. A block marked intending-to-deallocate is currently part of an object, but is to be removed from the object and freed if the transaction succeeds.

In both of these intention states, the decision whether to perform the intention or not depends on a single bit. The cylinder map entry for the root block of every object contains a commit bit which is the authority for deciding whether to perform or to reverse any intentions which may have accumulated on the object.

The steps necessary to perform an atomic update on an object can now be described. The first step in the sequence is to obtain an exclusive interlock on the object. This is needed to ensure that any intentions created on the object are due to a single transaction. In the general case, a client will have obtained a TUID for the index or special file, and can now request a sequence of updates to the object. For a special file, the sequence will consist of file write requests, but for an index it can be a mixture of delete, retain, create file, and create index operations. In either case, the file server must make either all of the updates to the object or none of them.

At the object level, all these updates will have been translated into requests to change the contents of one or more blocks of the object. Index operations always change only one block, but a file write may extend over many consecutive blocks of the object. For each block changed, however, the same procedure is followed. The block which currently occupies the block position is read into a buffer if necessary and its contents are changed. When it is to be written out, however, a free block is chosen and marked intending-to-allocate. The

object UID and the tree position intended for the new block are inserted in its cylinder map entry. The new data are then written to the new block. The old block, if there was one in this block position, is marked intending-to-deallocate in its cylinder map entry, and the object tree is changed to point to the new block.

At the end of this procedure, there are two blocks, each claiming to occupy the same tree position in the same object. One is marked intending-to-allocate, and the other intending-to-deallocate.

This is repeated for every block to be updated by the transaction. It may happen, of course, that two or more operations in the transaction update the same block of the object. In this case, the first operation will create a pair of block intentions as described above, but all further operations will write in place to the new copy of the block. This is clearly acceptable, since any interruption causing the corruption of this block, such as a power failure, will cause the entire transaction to fail and the original block to be reinstated in that block position.

The essential property of this mechanism is that at all times, two states of the object are maintained. One is the current state as described by the object tree. Whenever a leaf in the tree is replaced, the corresponding pointer block in the object is updated to point at it. The other state maintained is the original state of the object. This is described by the set of blocks marked intending-to-deallocate in their cylinder map entries. At any time it is possible to reinstate the original blocks in their tree positions.

To commit the transaction when all updates have been successfully performed, the commit bit for the object (held in the cylinder map entry for the root of the tree) is set. Since the decision whether or not to perform all intentions on the object depends solely on the value of this bit, setting the bit changes from a state in which all updates are tentative to one in which all updates will be done before a new interlock is allowed on the object.

Once the commit bit is set, the updates will ultimately be performed. Before the object is ready for a new update, however, all intentions on it must be removed. This is done by altering the cylinder maps which were earlier marked with tentative changes. Any block belonging to the object marked intending-to-allocate is marked allocated. Any block marked intending-to-deallocate is marked deallocated. Only when all intentions have been performed can the commit bit be reset to indicate that all intentions are to be reversed.

There remains the problem of ensuring that interruptions are correctly dealt with. For the purpose of the present discussion interruptions can be divided into two classes. Some, such as communications errors or crashes in client computers, will result in

the file server being unable to complete a transaction but still having the entire description of that transaction in its volatile memory. This state information consists of the UID of the object being updated and a list of the blocks on which intentions have been created. The second type of interruption causes the loss of the file server's volatile memory, and of all knowledge of transactions in progress. These interruptions are more difficult to deal with, since only the contents of the discs are available to determine the state at the time of the interruption.

Errors of the first kind are straightforward to deal with. If the list of blocks on which intentions have been created is not lost, then the states of these blocks can be put back to their values before the transaction started. Blocks marked intending-to-allocate are changed to deallocated, and those marked intending-to-deallocate are marked allocated. These intentions also describe precisely the changes which must be made to the object tree to restore its pointers to their original state. For each block marked intending-to-allocate, it is necessary to remove its disc address from the object tree. Correspondingly, for each block marked intending-to-deallocate, its disc address must be written back into the object tree at the position indicated by its cylinder map entry.

Before discussing how recovery is done for interruptions of the second type which cause loss of the contents of volatile memory, it is necessary to describe how the atomic update mechanism makes changes to the disc representation of an object. At every step the restart algorithm must have enough information to perform recovery, even though a disc block may have been corrupted at the time of the interruption.

The actual sequence of operations performed during an atomic transaction is as follows:

1) After an interlock has been obtained, the root of the object tree is read into a buffer.
2) For each block to be updated, the cylinder map for the old block is read into a buffer, and the old block is marked intending-to-deallocate. A free block is chosen in this cylinder map if possible, or in another one nearby, and its cylinder map entry is marked intending-to-allocate. The new data are written to the new block, and the new block's disc address is written into the appropriate object map block held in a buffer.
3) When the transaction is to be committed, all the cylinder maps modified in step 2 are written to disc.
4) All modified map blocks in the object tree are written back to disc.
5) The commit bit is set in the cylinder map entry of the root of

the object tree, and this cylinder map is written to disc.

6) A reply is sent to the client indicating that the transaction has completed successfully.
7) The cylinder map entries for all blocks with intentions are changed. Blocks marked intending-to-allocate are changed to allocated, and blocks marked intending-to-deallocate are changed to deallocated. The cylinder maps, which are the same as those modified in step 2, are again written back to disc.
8) The object's commit bit is reset and the cylinder map which contains it is written back to disc.
9) The interlock on the object is released.

In this sequence, there are a number of important orderings. In step 3, when the cylinder maps are written, the object tree is in its initial state on disc. If a cylinder map is lost, reconstruction is possible by examining all object trees for blocks residing on that cylinder. The same is true in steps 5, 7, and 8 when the object tree is now in its correct final state. Conversely, in step 4, when the object map blocks are written to disc, the cylinder maps are known to reflect the new state of the object. Thus they too are redundant when written because by reading all cylinder maps the tree structure of the object can be rebuilt. The result of this ordering of the disc writes is that an object map block is only written when the set of cylinder maps is known to be correct. Similarly, a cylinder map is only written when the set of object maps is known to be correct. This mutual redundancy is used to guarantee that when writing structural information, it is always possible to reconstruct either the state before the transfer, or the the intended state after the transfer in case of an interruption.

A final observation on the ordering of the disc transfers is that it is safe at step 6 to reply to the client indicating that a transaction has completed. The successful writing of the commit bit to disc at step 5 guarantees that even though any number of interruptions may occur, the remaining steps of the sequence will have been performed by the time a new interlock can be obtained on the object.

We can now return to the method for recovering from errors which cause the loss of volatile memory. The job of the restart program is to deal with any unfinished transactions. A transaction which had not yet set its commit bit should have all its intentions reversed. One which had set the commit bit but had not finished clearing up intentions should have all its intentions performed.

To find out if any objects were being updated at the time of the interruption, the restart algorithm makes a scan of all cylinder maps, examining the allocation state of every block. This is a relatively quick process which involves reading one block from each cylinder of

each disc. In the current implementation with two 80 Mbyte drives, this scan takes about 30 seconds.

Whenever a block is found in one of the intention states, an entry is made on a list of blocks belonging to the owner object. At the end of the scan, therefore, it is known which objects were being updated at the time of the interruption, and which blocks in each object have intentions on them.

All that remains to be done is to complete each transaction according to the value of the commit bit. If the bit is off, then the transaction was interrupted before step 5 in the update sequence. In this case the object tree must be forced into agreement with the cylinder maps, the intentions on the object should be removed and the cylinder maps written to disc. If the commit bit was set, however, then the restart algorithm continues the update sequence from step 7; the intentions are performed in the cylinder maps which are then written to disc, and the object commit bit is reset and written to disc. The client, however, may be in doubt as to the state of the object, since the interruption could have occurred immediately after writing the commit bit in step 5. Since the identity of the caller is part of the volatile information which is lost in an interruption, the reply cannot be sent again on restart. The object will be in one of only two possible states, however.

As mentioned previously, there is a chance that a disc block was corrupted at the time of the interruption, and the restart algorithm must clearly cope with this possibility. During the scan of the cylinder maps, therefore, if any of them are found to be unreadable the normal restart algorithm is temporarily abandoned in favour of reconstructing the cylinder map. To do this, the UID of the root index is retrieved from a fixed location on disc inaccessible to clients. From the root index, and by reading only map and data blocks for indices, every valid UID can be found, and so every object tree. These trees are then enumerated to find those which had blocks on the damaged cylinder, and the map entries in it are reconstructed as they are encountered. Commit bits for root blocks can be turned off in reconstructed entries, since by the commit algorithm presented earlier, the object tree is in a correct state. Any intentions on such objects should be dealt with by examining the current state of the object tree, however. Once the cylinder map has been rebuilt, the standard restart algorithm can be restarted.

Similarly, it is possible to find a corrupt object map when attempting to force an object tree into agreement with the cylinder maps. A block corruption in step 6 will cause this to happen. Again, the standard restart sequence is abandoned in favour of reconstructing the broken object map. This time the cylinder maps are known to be correct - they have already been scanned in the initial restart

sequence - and by reading them again the object tree can be rebuilt.

4.5 The Cost of Atomic Transactions

For the purposes of estimating the expense of the atomic transaction mechanism, it is possible to compare the steps needed for two updates differing only in that one is to a normal object and one to a special object. The extra disc transfers involved for the special object will give a measure of the cost of the atomic update mechanism.

In both cases, performing the actual data transfers will require a certain number of disc transfers. One will be needed to write the new data of every block to be changed and perhaps one or two might be needed in the case of a three-level object to access the second-level map blocks. Over and above this, however, the atomic update will require additional transfers. If the blocks to be updated are distributed over n cylinders, these cylinder maps must be read so that the old blocks can be marked intending-to-deallocate. Before the new data are written, new blocks must be chosen. These will be chosen if possible on the n cylinders whose maps are already accessible, but if there are not enough free blocks on them, a further n' cylinder maps must be fetched.

Assume that these blocks are pointed at by m object maps. (Only for three-level objects will m be greater than one.) When the transaction is to be committed, the n+n' cylinder maps which have been modified must be written to disc. Then the m altered object maps must be written followed by the cylinder map with the object's commit bit. At this point, the reply can be sent to the client. To finish off the transaction, the n+n' cylinder maps must be altered and written again, and finally the commit bit must be turned off. The total number of extra disc transfers is thus n+n' reads and 2(n+n'+1)+m writes of which 2n+2n'+m+1 are synchronous with the client, and n+n'+1 are asynchronous.

In practice, these overheads are usually less than they might appear. To a large extent, the extra reads can be eliminated by maintaining a cache of cylinder maps in volatile store, and the remaining transfers will tend to be grouped in a small number of adjacent cylinders.

Another mitigating influence is that the cost of an atomic update is incremental, depending not on the size of the object, but on the amount of information to be changed. Experience has shown that most special files and indices are small, and are stored as one- or two-level trees. Even wholesale replacement of these objects, such as is

caused by CAP swapping a special file out of its store, thus affects only blocks on one or two cylinders.

A typical situation, therefore, is one in which the data blocks of an object which are to be changed have all been allocated on the same cylinder. Assuming a moderate disc loading, there will be enough free blocks on the same cylinder to replace them. Normally, these blocks will be pointed at by a single map block, which is likely to be the root of a two-level tree. The overhead in disc transfers in this typical case, therefore, consists of:

write cylinder map with intentions	1
write object map	1
turn on commit bit	1
perform intentions	1
turn off commit bit	1
	---
	5 transfers

This situation is so common that it is possible to make an advantageous optimisation. It is very frequently the case that all blocks involved in a transaction reside on the same cylinder as the root of the object tree. The policy for choosing the cylinder on which to allocate a new block always favours this 'home' cylinder. In this case it is possible to exploit the fact that there is only one cylinder map involved in the transaction. Since there is no need to coordinate the state changes between blocks on several cylinders an optimisation can be made. When such a transaction is to be committed the sequence performed is:

1) Write the cylinder map containing intentions to disc.
2) Write the changed object map to disc. Step 1 is only taken so that if this transfer fails, the restart algorithm will find the broken object map while attempting to remove the intentions.
3) Without changing the value of the commit bit, perform intentions as if it had been set. Write the cylinder map to disc.

Step 3 in this sequence is the equivalent of turning on the commit bit, performing the intentions, and then turning off the commit bit. Since there is only one cylinder map involved, these operations can be condensed into a single disc transfer.

Thus in a large number of cases, the atomic transaction overhead is three disc transfers, all synchronous with the client. These transfers all occur on the same cylinder, and require a total of about 30 milliseconds. The optimised algorithm is still safe, however. Corrupting the cylinder map in steps 1 or 3 can be recovered from in the normal way. Losing the object map in step 2 will be noticed on restart due to the presence of intentions on the object, and the block can be reconstructed from the cylinder maps as previously described.

4.6 A Special Case

The mechanism presented in this chapter solves two problems. The first is the need to provide structural redundancy so that recovery can be successfully performed even if a single disc block is lost. The second is to make any number of block allocations and deallocations conditional on the value of a single bit. In both cases this is done by adding redundant information to the allocation bit maps.

The addition of the intermediate allocation states solves the consistency problem between the cylinder maps and object maps, because it becomes possible to distinguish blocks which are in use from those merely marked in use but in no object tree. Any blocks for which this is true will be in one of the intention states. There is thus no need to perform the costly scan of all known objects to verify the allocation states of all blocks, because a simple linear scan of the cylinder maps will suffice. In practice, this linear scan will be orders of magnitude faster than the graph scan and has the advantage of not becoming more expensive as the graph structure grows.

Atomic updates on an object are provided by the ability to make all the tentative changes to it conditional on the value of a single bit. In effect this bit can be seen as determining which of two possible sets of disc addresses should occupy the positions of the object tree. If the bit is set, then all blocks marked intending-to-allocate should be inserted in the tree. If it is not set, then the blocks marked intending-to-deallocate should occupy the same slots.

A problem arises when considering one-level objects. These objects consist of exactly one disc block containing the object attributes and the data as shown in figure 4.1. To attempt to update a one-level object by the same mechanism used for two- and three-level objects, one would choose a new free block, write the new copy of the object into it, and set the commit bit in the old copy of the first block. At this point, the method breaks down; unlike blocks at lower levels in the tree, there is no single place where a disc address can be written to indicate the new location of the object. The disc address of the old block has already been distributed to an unknown number of clients embedded in the unique identifier of the object. The first block cannot therefore be moved.

For this reason, one-level objects are treated as a special case by the file server. If the object is normal, writing occurs in place to the root block, with attendant risk of corruption. In the case of a special object, a temporary copy of the root block is created for the duration of the transaction. The actual sequence performed is:

1) choose a free block and mark it intending-to-allocate with a cylinder map entry describing it as the first block of the object.
2) write the new data to the new block.
3) write the cylinder map for the new block to disc.
4) when the transaction commits, write the new data to the original root block and
5) mark the new block deallocated and write its cylinder map to disc again.

Five disc transfers are required, the same number as for an update to a single block of a two- or three-level object.

This sequence requires the cooperation of the restart algorithm. A crash before step 3 will cause the attempt to update to be ignored, since the new block will be marked deallocated on disc. A crash after step 3 but before step 5 is recognised as such because the only time an intention to allocate can occur on a root block is during this sequence. (Creation of an object causes the first block to change from deallocated to allocated without passing through the intending-to-allocate state.) The restart algorithm continues from step 4 in this case.

This sequence is also safe against block loss. The loss of a cylinder map is handled in the normal way. If the new data are lost in step 2, this fact will be ignored by the restart algorithm since the new block will still be marked deallocated on disc. If the root block is lost in step 4, the restart algorithm will correct it by writing the new data again.

Chapter 5

Structure of the File Server

In the previous chapter a design for the structure of the information maintained by the file server was presented. This is one among many possible designs, chosen on the grounds of efficient space utilisation and the possibility of inexpensive atomic transactions. However, the disc structure chosen retains sufficient information at all times to be proof against arbitrary interruptions.

This chapter describes the structure of the file server program. It is to a large degree a description of how the interface defined in chapters two and three and the algorithms defined in chapter four can be made to work on a practical basis. Sections 5.1 and 5.2 describe the hardware environment, and section 5.3 describes a methodology for constructing the file server as a set of modules. Section 5.4 gives a rapid overview of the program's operation as a set of cooperating module instances. The remainder of the chapter, from section 5.7 onwards, is a more detailed description on a module-by-module basis, and may be omitted by readers not interested in the finer points of the structure.

The hardware used by the file server consists of a 16-bit minicomputer with 64K words of store and three attached peripherals. The disc controller provides block-at-a-time access to two 80Mbyte removable discs. Character-at-a-time access is provided to a teletype. The third peripheral connection is to the Cambridge ring via a direct memory access mechanism.

5.1 The Cambridge Ring

Wilkes [Wilkes79a] describes the operation of the Cambridge ring, a high-speed local area network. Each attachment to the ring comprises two units, the repeater and the station. Repeaters manage the analogue propagation of signals in the ring, each one receiving from its upstream neighbour and transmitting the regenerated and possibly modified signal to its downstream neighbour. Transmission in the ring is unidirectional, since each repeater receives from only one other, as opposed to bidirectional as in Ethernets [Metcalfe75, Shoch80]. A full discussion of design alternatives for local area

networks is given in Hopper78.

In the ring, a fixed number of fixed-size slots are constantly circulating. Each slot is either empty, or contains a single ring packet, as determined by a full/empty bit at the start of the slot. A ring packet contains a bit used by a distinguished monitor station to detect lost packets, some addressing information which identifies both the source and the intended recipient of the packet, and a small number of data bytes, two in the present system. The packet is terminated by two hardware response bits, whose use is described below, and a parity bit used for error detection.

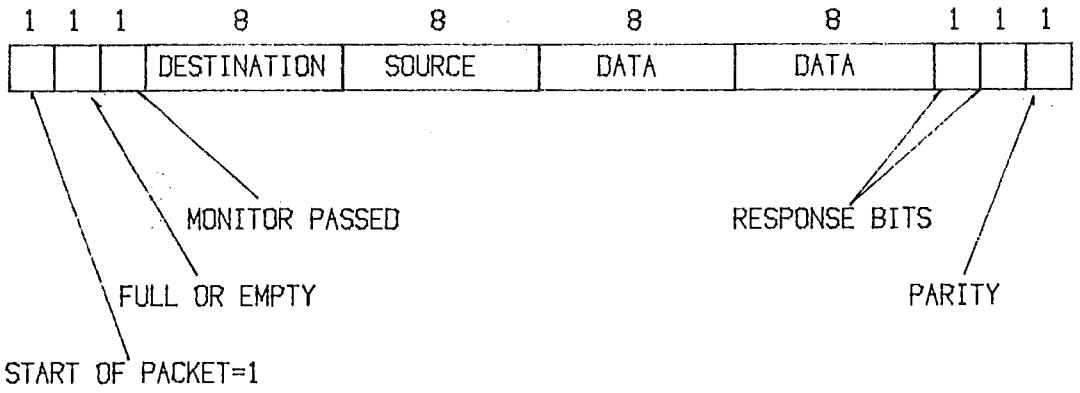

Fig 5.1 Ring Packet Format

The station half of a ring connection is concerned with the digital interface between the ring and the host connected to it, usually a computer. Every station is full duplex and provides control signals for the simultaneous reception and transmission of ring packets. Each half of the station contains a data register which provides a one-packet buffer. On the transmit side, this is used to hold the contents of the next ring packet to be transmitted. On the receive side, it is used to hold a packet accepted from the ring but not yet read by the host. There is also a select register in the receiver interface which is used by the host to select stations from which it wishes to receive ring packets in a manner to be described below.

To send a packet, the host loads the data to be sent into the transmit data register. The station then begins examining the slots passing by it on the ring. When an empty slot is detected, the station marks the slot full, loads the destination station number, its own station number, and the data register contents into the slot, and finally initialises the response and parity bits.

The packet is then passed from repeater to repeater until it reaches the one connected to the destination. If the destination station is ready to receive a packet from this source, the data will be copied into the receive data register, and the response bits of the

packet will be marked accepted. As the ring packet returns to the sending station, it is marked empty and the response bits are passed back to the host.

Other outcomes are possible, however. If the receiving station is switched off or no such station exists, the packet will return to its sender with the response bits still set to ignored. Alternatively, the receiving station may not be willing to accept a packet from the sender. This is determined by the select register, whose possible values are 0, meaning receive from no one, 1<=n<=254, meaning receive from station n only, and 255, meaning accept from anyone. If the destination is unwilling to receive from this source, the packet is marked unselected as it passes, and the data are not copied to the receive data register. The final possibility is that the receive station register setting allows reception from this source, but a previously accepted packet has not yet been read by its host and is still blocking the receive data register. In this case, the packet returns to the sender marked busy.

The response bits provide a low level acknowledgement to the transmitter. In practice these responses are used to match transmission speed to reception speed in higher levels of protocol. In general, a packet returning marked busy is not regarded as a serious error, and the normal response is to retransmit it immediately or after some delay. Unselected and ignored packets, however, are more serious errors which usually cause abandonment of the current unit of transmission at the next level of protocol.

The ring protocol just described provides for unidirectional communication of 16-bit packets between physical machines. Information is provided for flow control (the response bits) and error control (a host is informed whether each transmitted packet returns unchanged) but no actions are defined to deal with abnormal transmission. A station will indicate the failure of a packet transmission to the host but will not take any remedial action.

5.2 The Basic Block Protocol

At the software level, most communication takes place between processes rather than machines. In the ring packet protocol, there is no way for a receiver to distinguish packets sent by two different processes in the same transmitting machine, because only the identity of the transmitting station is included in the ring packet. Thus, the ring packet protocol implicitly limits the number of conversations between two machines to one in each direction. Furthermore, the ring packet protocol provides for the transmission only of 16 bits between

machines, whereas most inter-process messages require larger units of communication.

For these reasons, a basic block protocol is defined to allow unidirectional transmission of larger messages between processes. The format of a basic block is shown below:

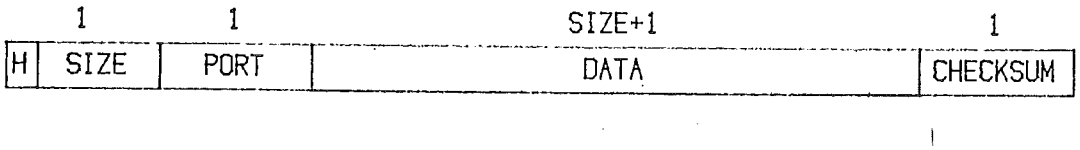

Fig 5.2 Basic Block Format

The first ring packet in a basic block is the header packet. This contains in its data field a header code H and a 10-bit size field. This size field determines how many ring packets of data the transmitter wishes to send. The second packet contains routing information in the form of a port number. The port number is used by the receiving machine to direct the basic block contents to a particular process within it. Following the port number are size+1 data packets which contain the message to be transmitted. Since the size field is 10 bits long, there can be from 1 to 1024 ring packets of data, so that the basic block protocol allows transmission of up to 2048 bytes of data at a time. Longer communications can of course be built up out of sequences of basic blocks at higher levels of protocol.

The final ring packet of a basic block contains a checksum and is used to verify correct reception. Depending on the particular value of the header code H, this packet contains either zero or the arithmetic sum with end around carries of all preceding packets in the basic block. The purpose of the checksum packet is to detect transmission errors and also synchronisation errors in which the receiver takes a random ring packet as the start of a basic block.

The two problems which a protocol must solve are error control and flow control. Error control consists of verifying that all data were received as transmitted. All communication errors must be detected, and possibly corrected, with acceptable probability. Flow control involves ensuring that at all times the transmitter only sends as much data as the receiver is prepared to accept.

Error control can be handled in a relatively relaxed manner due to the inherent reliability of the Cambridge ring. Since the ring error rate is on the order of one bit in $5 * 10^{11}$, the most economical

method of performing error control is for the receiver to acknowledge the correct reception of an entire basic block sequence rather than acknowledging each block individually. For example, in transfers of 10^6 bits, it is better to allow for retransmission of the entire megabit once in $5 * 10^5$ attempts than to double the number of blocks involved in each transmission by using explicit per-block acknowledgements.

Flow control cannot be treated so simply, however. At the time of each basic block transmission the receiver must have sufficient buffer space waiting; if not, the block will be ignored. As in the case of errors, the loss of a basic block for this reason could be accepted if it were a rare event. Handling flow control in this way implies that the receiver must always be able to accept basic blocks more quickly than any transmitter can send them. This will not always be the case in file transfers. During writes to the file server, there are occasions when the scattering of a particular file over the disc surface, or the need to allocate blocks to the file on the fly will not allow the file server to accept large transfers at full ring speed. Conversely, particularly slow clients may not be able to receive at the file server transmission rate. Thus, some form of flow control is necessary.

Fortunately, the ring packet response bits make flow control very cheap to implement at the basic block level. Normally, a station willing to receive basic blocks from a number of potential transmitters will have its select register set to 255, the value meaning "anyone". On receiving a ring packet which appears to be a valid basic block header, the host will then alter the select register so that only ring packets arriving from that particular source are accepted*. The next packet to arrive will be the route packet which defines the process to which this basic block is directed. At this point the receiver has all the information needed to decide whether or not to accept the basic block; some process must previously have expressed willingness to receive a basic block on this port number, and must have provided sufficient buffer space to hold the basic block which is now arriving. During the time necessary to make this decision, the transmitter will be seeing busy responses to the third ring packet of the block, and will under normal circumstances be repeatedly attempting to send it. The receiver can now do one of two things. If an appropriate buffer for the basic block has been found, the data packets can be read into it. The transmitter will see successive packets of the basic block being accepted by the

* A very fast host could leave its station register set to "anyone" and select the buffer in which to write the packet using the source station number. This has not been done in practice.

destination. Alternatively, the receiver can set its select register to zero for a short time. If the transmitter attempts to send his third ring packet during this time, it will return marked unselected; this is conventionally considered a protocol error which causes the entire basic block transmission to be abandoned and restarted. Thus, a slow receiver can indicate that a buffer is not yet ready by temporarily refusing to receive any ring packets at all. After this short interval, it would again set the station select register to "anyone" and begin to look for the start of a new basic block.

A difficulty with this method of flow control is that it is impossible to know for how long the station select register should be set to zero. If this is only done for a short time, then the transmitter may not attempt to send the third packet during the interval, and thus will not detect the receiver's unwillingness to accept the basic block. This "deselection", then, can only be regarded as a hint from receiver to transmitter, which hopefully will be taken most of the time.

This objection does not make the method of flow control unworkable, and it can be overcome by a pragmatic approach to higher levels of protocol. By using the select register to hold off a transmitter which is sending too quickly, a large percentage of the time the mechanism will work and the transmitter will detect the unselected response. In rare circumstances - the rarity depending on the length of time for which the select register is set to zero and the retry frequency of the transmitter - data will be lost by this means.

In the design of the file transfer protocol, these properties of the basic block protocol are assumed and exploited. During a file transfer, no errors detected by the receiver are reported to the transmitter, and the flow control mechanism described above is used to keep receiver and transmitter in step. This simple approach to error and flow control means that during a file read or write operation, a one-way stream of basic blocks flows from transmitter to receiver without any explicit acknowledgements flowing in the reverse direction.

It must be stressed that the simplicity of this protocol is due to two circumstances. Firstly, the low error rate of the Cambridge ring, in common with other local networks, means that error acknowledgements can be made over very large units. This would not be true if a telephone link were in the transmission path, for example. Secondly, the design of the ring provides acknowledgements at the hardware level on which flow control at the basic block level can be built very cheaply. A more expensive scheme would be needed in networks such as Ethernets [Metcalfe75] which do not provide low-level acknowledgements. There is, however, a potential problem with this mechanism if receiver and transmitter are on different rings connected

by a gateway. If the gateway operates on a store-and-forward basis for complete basic blocks, there would be no way for it to propagate an unselected response from the receiver back to the transmitter. If the receiver in a file transfer through the gateway could not accept data at the transmitter's rate, then a more elaborate - and more costly - protocol would be needed to perform flow control.

5.3 The Programming Environment

The file server is written in BCPL [Richards69], and runs as a set of asynchronous tasks under the kernel of the TRIPOS operating system [Richards79]. Since the program structure of the file server is determined to a certain extent by the characteristics of both the language and the operating system, the important features of these will be described here.

BCPL is a block structured language of the Algol family with two major simplifications. Firstly, the language contains precisely one data type, which is identified with a unit of addressing of the underlying machine. Integers, characters and Boolean values can all be stored without restriction in a BCPL variable, whose interpretation is left completely to the programmer. So too can addresses, whose values are unrestricted. The BCPL programmer is free to assign labels, procedures or the values of arbitrary expressions as values of variables, a powerful but dangerous ability.

The second major simplification of BCPL over other Algol-like languages is in the scope of variables. In any procedure activation, only certain variables are accessible; the parameters of the procedure, the local variables of the procedure maintained on the stack, and the global variables accessible to all procedures*. A procedure in BCPL cannot refer to variables declared in textually enclosing blocks, contrary to the normal scope rules of block structured languages. The global variables, however, correspond to the variables declared at the outermost level of nesting of a BCPL program, and for these specially defined variables an exception is made. All blocks and procedures within the program may refer to global variables directly.

The set of global variables known to a program are collected together in the global vector. The definition of a global variable is a simple matter of associating a textual name with an offset in the global vector. The programmer must choose the specific offset within the global vector to be associated with a particular variable. The

* A fourth category, the static variables, is not of interest here.

global vector provides not only a mechanism for communication within a BCPL program, since the textual name of a global variable has the same value in all parts of the program, but also allows communication with separately compiled programs and the operating system. Independently compiled BCPL programs can communicate only by agreeing on the use of global vector offsets. Some are reserved for the addresses of generally useful procedures needed by almost all BCPL programs. Other global vector offsets must be reserved for communication among the separately compiled programs. This is the means by which a procedure defined in one program can be called from a procedure in the other; the same global vector offset is defined in each program as holding the address of the procedure. Finally the programs must also agree on the global vector offsets which are not to be shared. Only careful attention to the use of global vector entries can prevent conflict over the use of an entry, and in a large program, the definition of the shared and unshared parts of the global vector must be done with particular care.

5.3.1 The TRIPOS Operating System

The mechanisms outlined in the previous section allow independently compiled BCPL programs to be combined in a single unit. The common global vector provides the communication area between these programs. Since there is a single procedure invocation stack associated with the global vector, there is a single thread of control through programs connected in this way. This sort of unit naturally represents a process.

The kernel of the TRIPOS operating system described in Richards79 extends this mechanism to allow multiple processes.

The TRIPOS kernel is the coordinator [Wilkes75] of a number of tasks and devices as shown below. Each of these defines an asynchronous activity. Devices are low-level peripheral drivers of a restricted form whereas tasks may run arbitrary BCPL programs.

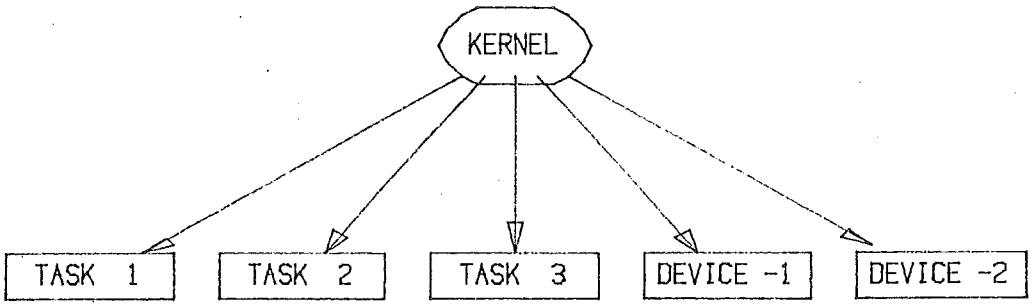

Fig 5.3 TRIPOS Kernel, tasks and devices

Each task is a BCPL program with a stack and a global vector as described above, and executes asynchronously with other tasks and devices. Within each task, the kernel is visible as a number of procedures at well known offsets in the global vector.

Communication between tasks is performed by means of messages*. A message is a vector of words of store whose format is shown below. Like all multiple-word BCPL structures, a message is defined by the address of its first word.

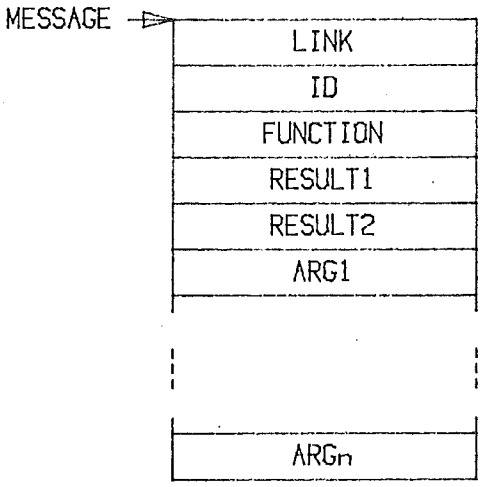

Fig 5.4 TRIPOS message format

The link word is used by the kernel to queue messages sent to a task or device but not yet accepted by it. The id word specifies the task or device to which the message is to be sent. The remaining

* The actual word used by the designers is 'packet', which is avoided here to prevent confusion with the ring terminology.

words in the message are used to define the service requested, and are not interpreted by the kernel. Conventionally, a function code is supplied together with any number of arguments, and two result fields allow information about the operation to be returned to the caller.

Messages are sent by the kernel procedure qmsg, which takes as its one argument the address of the message to be sent. Normally the task which executed qmsg will wait for a reply at some later time by executing reply := taskwait (). This procedure suspends the calling task until there is at least one message waiting on its input queue. The first message received is then unlinked from the input queue and its address is returned as the result of the call of taskwait.

Devices differ from tasks in a number of ways, largely a result of efficiency considerations. A device also represents an asynchronous activity, but its code is written in assembly language rather than BCPL. Devices and tasks communicate by messages, however, in the way just described.

5.3.2 Constructing Larger Systems Using the TRIPOS Kernel

The kernel and devices may be thought of as providing a minimal run-time environment for BCPL programs. They provide the multiprocessing, store management and simple I/O primitives which any large program would need for its implementation. Richards et al. [Richards79] describe the construction of the upper layers of the TRIPOS operating system using these facilities. TRIPOS was constructed by adding layers of abstraction in the form of BCPL programs organised as tasks.

Other systems are of course possible. In this respect, the TRIPOS kernel and the layers of operating system defined using it have the advantages of the "open" operating system for the Alto computer [Lampson79], and of the run-time system of languages such as Mesa [Mitchell79]. In these environments the programmer is free to include only those elements of the operating/run-time system which are needed by his program, the rest being excluded with consequent savings in space. In TRIPOS, the tool needed for this purpose is the system generation program, which constructs a complete store image as a file in the TRIPOS filing system. The programmer selects the binary versions of the programs to be used, the number of tasks and devices required, and the tasks which are to be able to call code segments as procedures defined in their global vectors. The resulting file, which is an image of the program at time zero, can then be started by a bootstrap program which loads the file contents into the machine store and transfers control to the first instruction.

This procedure was used for construction of the file server using elements of TRIPOS. The kernel with its basic services for store and task management was retained, as were a console handler task to provide a stream input/output interface to a terminal, and a debugging task used for examination of the program state after an error. The remainder of the abstractions provided by the TRIPOS operating system were not used.

5.3.3 Modules

Specifying the structure of a program to the system generation program is a matter of defining a number of tasks and of defining the parts of the program which are to be accessible as procedures in each task. Two pieces of code accessible to the same task can invoke each other by means of BCPL procedure calls, but must use TRIPOS messages if they are in different tasks. At execution time, of course, the communication method to be used must be known, but this level of detail should not be bound into the design of the program. As argued in Stroustrup79, the number of instances of a particular piece of code largely depends on the degree of parallelism desired.

Suppose that some part A of the program needs to call part B to have a service performed. There are a number of ways in which this invocation can be done. Firstly, it might be arranged that each task which can execute module A also has access to module B. In this case A can invoke B as a procedure by calling B's entry point directly, and there is essentially one instance of B for every instance of A, since every instance of A might simultaneously be invoking B.

Secondly, B could be isolated within a single task. Each task running A must invoke B by sending a TRIPOS message to it and waiting for a reply. Multiple instances of A, therefore, must queue up waiting for service from the single instance of B. This queueing is explicitly represented in the chain of messages waiting on the request queue of B's task.

The third possibility is to have some number of tasks each running B as its only domain. When a call was to be made, an instance of A would look for a free instance of B to invoke using the message mechanism. There is clearly no point in having more instances of B than of A, but any lesser number can be chosen. The case of having the same number of B's as A's essentially reduces to the first alternative, and having one B is precisely the second alternative.

The choice between these alternatives is largely a matter of the desired dynamic behaviour of the program. The more instances of B available, the higher the potential concurrency in the system will be. At the program design level, however, the only fact of interest is that A requires B to have a service performed. The balance between

concurrency, store usage, and the differing costs of procedure and process invocation are not at issue here, and are likely to confuse the design process.

The extreme position of postponing all decisions as to the number of instances of a module and its method of invocation until the last possible moment is not practical, however. In common with other message-oriented systems [Lauer79], the only mechanism provided for synchronisation in TRIPOS is the queueing of request messages for a task. Any module, therefore, which controls a central data structure must be implemented as a single task, since there is no way for multiple instances to synchronise on updates to the structure. This fact dominates over any considerations of potential parallelism.

In addition, the implementation decision can only be postponed for procedure-like invocations. A call from module A to module B in which module A is idle during B's execution can be represented either as a procedure call within a task or as an inter-task message. If A has useful work to do while B is active, however, there must be two threads of control and hence two tasks involved.

The idea that a program can be designed and largely implemented without consideration of the degree of parallelism involved is a useful one which must not be carried too far. Thinking of a program as consisting of a set of abstract object managers [Parnas72, Liskov74, Jones78] is generally accepted as a useful method for the decomposition of a program into a set of units of manageable complexity with well-defined interfaces. The units of design and implementation can then be mapped onto the underlying process mechanism.

In the file server, the design was approached through the consistent use of a module formalism. The module is simply a mechanism for the clear definition of the interface provided by a unit of the file server which does not oblige a procedure-like or process-like invocation.

All arguments passed between modules on a call are in the form of a TRIPOS messages. The function word which selects the service, however, is slightly elaborated. It consists of a module number and a service number within the module, and uniquely identifies the requested service.

Every module in the file server is programmed as a BCPL procedure in the form shown below.

```
LET M.module (msg) = VALOF

    $( LET ep1 (msg) BE
           $( external procedure 1 code $)
        :     :      :
       LET epn (msg) BE
           $( external procedure n code $)

       LET ip1 (arg1, arg2, ...) BE
           $( internal procedure 1 $)
        :     :      :
       LET ipm (arg1, arg2, ...) BE
           $( internal procedure m $)

       rc := rc.done

       SWITCHON msg ! msg.function & 255 INTO
       $( CASE 0: ep0 (msg); ENDCASE
            :        :          :
          CASE n: epn (msg); ENDCASE
          DEFAULT: rc := rc.wrongfunction
       $)
       RESULTIS rc
    $)
```

Fig 5.5 General Module Structure

This figure shows the structure of a general module M. It is a procedure with one argument which is the address of the request message to it, and one result, a return code value which informs the calling module of the success or failure of the invocation. The mechanism for passing messages between modules will be described shortly. When a module is called with a message argument, the request is performed by selecting the entry point required based on the least significant half of the function code. The value of the global "rc" set by the external and internal procedures is returned as the result of the module call.

This module structure implements the scope rules for procedures which are found in several object-oriented languages [Liskov74, Mitchell79] but at a small execution-time cost. Only the external procedures ep1 to epn are accessible to other modules via the SWITCHON statement. Procedures internal to M, ip1, ... ipm, cannot be called by code outside M. Thus the structure of a module clearly indicates which procedures within it are externally accessible and which are private to the module.

A module which needs to call another can do so using the global variable call.module (All the inter-module call procedures described in this section reside in a library of services available to all tasks). Call.module accepts as arguments the TRIPOS message contents of the request and returns the return code value set by the called module. An example might be

rc := call.module (0, 0, Diseman.write, 0, 0, bfw, da)

which is used to write the contents of a store buffer to disc. In the call, the first pair of zero arguments are the link and task id words of a TRIPOS message, and the second pair are the result fields. None of these values are known by the caller and are thus conventionally supplied as zeroes. The third argument is the function code. All function codes are defined as constants in a file accessed during compilation. This file contains the definitions of all interface procedures such as Diseman.write as a number (module number, entry number). In this case Diseman.write has the value 0601_{16} because Diseman is module number 6 and write is its first entry point.

The procedure call.module is thus supplied with the identity of the service requested, and the request message has been constructed in the correct order as a vector of arguments on the BCPL stack. The address of the message is simply the address of call.module's first argument, the link word.

To perform the call, two arrays are consulted, each of which is indexed by module number. The array task defines the task in which the called module must run. The task entry is one of: zero, in which case the call may be done directly as a procedure call; n, in which case the call should be directed to task n; or the address of a vector containing the numbers of tasks, any of which is prepared to accept the request. These alternatives correspond directly to the possible instantiations of a module described above. An entry of zero allows any number of tasks to be executing the module concurrently. An entry of n allows only task n to execute the module code. An address entry defines a number of tasks which may execute the module code concurrently.

If the module is to be called as a task (because the task array entry for the module is not zero) then call.module simply writes the appropriate task number into the id word of the message and uses the kernel primitives qmsg to send the message, and taskwait to await its return from the called task. The return code will have been written in result1 by the called module, and is returned as the result of call.module.

If the module is to be called as a procedure (because the task array entry for it is zero), then a second array proc is consulted. This array contains the entry point for each module, and like the task array is indexed by module number. Using this array, call.module can

find the procedure to be called - M.module in the preceding figure - and can enter the module directly. The value returned by the module call is also returned as the result of call.module.

From this description it can be seen that call.module implements synchronous calls between modules. Its purpose is to provide a uniform interface to both procedure-like invocation and task-like invocation so that synchronous calls can be expressed in a standard way.

As suggested previously, however, asynchronous invocation must be allowed. This is provided by two procedures act.module and wait.reply. The procedure act.module is used to start an asynchronous request and faults if it finds that the module is to be called as a procedure. Wait.reply suspends the caller until the request message has returned to the calling task. Their use is illustrated by the following equivalent program fragments.

```
$( LET msg = VEC 6
   initialise (msg, 7, 0, 0, M.entryreason, 0, 0, a1, a2)
   act.module (msg)
   rc := wait.reply (msg)
$)

rc := call.module (0, 0, M.entryreason, 0, 0, a1, a2)
```

Fig 5.6 Asynchronous and Synchronous Module Invocation

A file server task is in one of three states when call.module or act.module sends a message to it:

1) It may be busy and uninterested in the arrival of a new message, having been interrupted by a higher priority task.
2) It may be blocked in call.module or wait.reply awaiting the reply to a module invocation of its own.
3) It may be idle, waiting for a new request for service.

In the first case, the TRIPOS kernel notes that the task is busy and puts the message on a queue of unread messages waiting for the task. In the second case, call.module or wait.reply will accept the message but will find that it is not the one awaited. The message will then be queued on a request queue global to the task, (not the one used by the kernel, to avoid looping) and the task will be suspended again. In the third case the task will be suspended in the lowest procedure on its stack, the start procedure. The purpose of the start procedure is to accept request messages, perform a procedure call on the selected module in the task, and return the message to the calling task after writing the result into the result1 field. Before blocking

again, the start procedure examines the request queue to see if any new requests arrived while the called module was active.

Figure 5.7 shows the module structure of the file server. This is represented as an invocation graph where the nodes are module names and an edge from A to B indicates that module A invokes the services provided by B. Superimposed on this graph is the task structure. Modules surrounded by ovals are accessible only to the start procedures of certain tasks. Root, for instance, is accessible to three tasks for reasons of parallelism, while many other modules such as Blockman or Storeman control a central data structure and thus are accessible only to a single task.

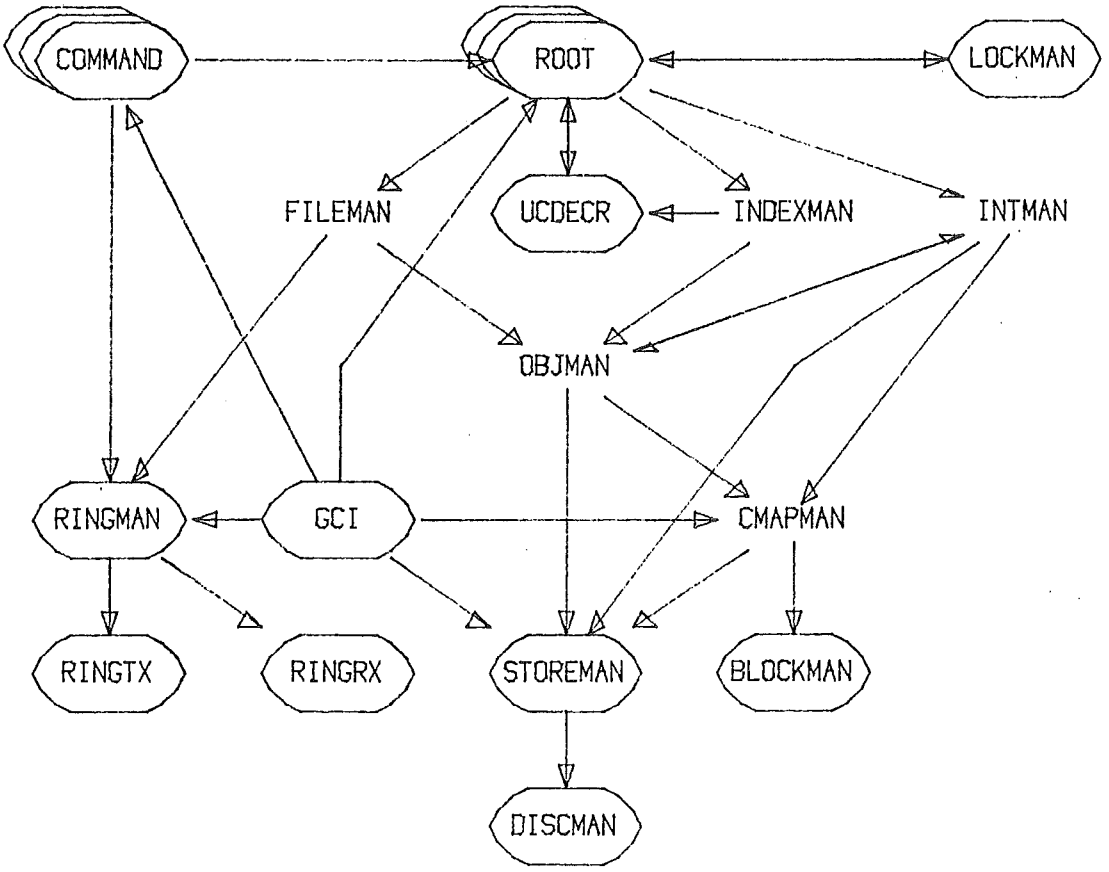

Fig 5.7 File Server Module Invocation Graph

The invocation graph distinguishes the cases of procedure-like and process-like invocation. Any edge which enters a task oval must use the message mechanism of the TRIPOS kernel. Any edge which does not is implemented as a procedure call via call.module. Thus, at the top of the graph, call.module implements the call of Indexman by Root as a procedure call, but sends a message when Indexman calls Ucdecr.

5.3.4 Type Checking

The file server spends much of its time manipulating data structures such as disc addresses, buffer descriptions and unique identifiers. These structures will be passed by reference in inter-module calls and because BCPL is not a strongly typed language, there is the potential for very obscure errors if these addresses are incorrect. To counteract this, a dynamic type checking convention is used at every module interface.

Each object representation which occupies more than one word of store is represented as a BCPL vector. Conventionally, the first word in all such vectors is reserved for a tag which uniquely identifies the type of the object. The type manager for an object writes the correct tag when the object is created, and as a general rule, every module entry point procedure explicitly checks the types of its arguments before performing any operations on them. This added amount of checking at inter-module boundaries has been invaluable in the debugging of a program of this size, and though it was intended to be removed eventually, has proved an excellent method for detecting occasional hardware errors. In the course of experience, most hardware and software errors have shown up quickly as type violations in a module call.

The module abstraction and dynamic type checking impose a certain run-time cost. To estimate these overheads, measurements were made in which a task woke up at regular intervals and recorded the procedure in which the next task to run would execute. The figures for the module invocation procedures are shown in figure 5.8. Each line represents the number of times that the procedure was found active as a proportion of all samples in which the processor was not idle.

call.module	5.9%
act.module	0.7%
wait.reply	0.1%
type	3.1%

Fig 5.8 Time in Procedures as a Fraction of Non-Idle Processor Time

These figures show that the module mechanism accounts for about 7% and the type checking mechanism for about 3% of the useful processor cycles.

5.4 Overview of File Server Operation

The invocation graph in figure 5.7 can be used to give a brief outline of the activity of the file server. All activity is triggered by requests arriving on the ring. In the idle state, the three instances of Command shown in the diagram will be waiting to receive a basic block on the file server's request port number. When a block arrives, it is passed by Ringrx to Ringman, which verifies the checksum and returns Command's message. On being resumed after the completion of its call.module, Command decodes the basic block and issues the corresponding call.module to Root. The instance of Root which receives this message sequences the operation, and maintains all the state information about its progress. Before acting on the request, however, Root obtains an interlock on the object by calling Lockman. It then selects the appropriate Fileman or Indexman function and issues a call.module. Fileman and Indexman are responsible for implementing files and indices using object trees provided at the Objman interface. Fileman's main task is organising rapid file transfers by overlapping accesses to the object tree with ring transfers by Ringman. Indexman interprets objects as a list of UIDs, and calls the use count decrementer Ucdecr whenever a UID is erased from an index.

Both Indexman and Fileman make all disc accesses through Objman, which provides an interface to objects as a sequence of data blocks. All operations on object trees are internal to Objman and are not visible to its callers. While acting on behalf of Fileman and Indexman, Objman calls the store manager Storeman to obtain and release buffers containing disc blocks. It also calls the cylinder map manager Cmapman to alter cylinder map entries, and the intentions manager Intman to record intentions for this transaction. When requested to find a free disc block, Cmapman calls the free block manager Blockman to find a cylinder on which there is free space.

When the Fileman or Indexman call completes, Root decides whether the request it is dealing with is the last in its transaction. If so, then Intman is called to commit the transaction. Intman performs all the steps of the commit sequence described in section 4.4 up to the setting of the commit bit. It then returns, so that Root can explicitly return Command's request allowing a reply to be sent to the client. Root then issues a second call of Intman for the asynchronous cleaning up of intentions which completes the commit sequence, and returns to the idle state.

The only module not directly involved in performing a file server operation is Gci, the garbage collector interface. This is the stub of an external asynchronous garbage collector. When started up in a remote machine on the network, the external collector uses Gci to scan the index structure to find objects not reachable from the root index. Gci's activity takes place in parallel with normal file server operations and requires only a minimum of synchronisation with them.

5.5 Command

The Command module drives the activity of the file server. In the idle state, each instance of Command is blocked waiting for the reception of a basic block on the file server's command port. This port number is well known, and is the one to which all clients direct their requests. The command instances indicate their willingness to accept requests by calling Ringman.receive, supplying a buffer into which a basic block can be received, a ring address which will select any block directed to the file server's request port port, and a timeout value. If the request returns from Ringman with a bad return code because the timeout expired before an acceptable block arrived, or because of a checksum error, Command simply resubmits it.

On reception of a basic block on the command port, Ringman will return the request message to Command, causing its invocation of call.module to complete. Command performs simple checks to reject obviously incorrect basic blocks, and then calls Root to perform the requested service. When this call completes, Command constructs a reply block to the client which includes the return code from Root, and calls Ringman.send to transmit the reply to the caller.

In most operations, Command manages all the network communication with the client, consisting of the single block reception and its reply. The only exceptions are file reads and writes, during which Fileman will engage in ring transmissions or receptions of the file contents. This fact is invisible to Command, which simply receives and transmits a basic block before and after each file server operation.

Command provides three important entry points for use by other modules. Command.stop is called when the operator types "stop" on the console. This causes Command to refuse new requests, and is used to stop normal service in an orderly way. Command.suspend and Command.resume are entry points used to allow the garbage collector interface Gci to synchronise with file server activity. Command.suspend causes all instances of Command to pause before beginning a new operation, and completes when the last instance marks

itself idle. Command.resume signals that normal activity can continue.

The invocation graph shows three instances of Command, and also three of Root. Thus at any time, three independent file server operations can be in progress, subject to the locking strategies to be described below. Command and Root are defined as separate tasks rather than as procedures within the same tasks so that commands can be generated internally. Under certain circumstances, Lockman, Ucdecr and Gci generate file server operations, and the machine level interface of Root is needed for these purposes.

5.6 Root

The name of the Root module derives from its function of sequencing every file server operation through a number of common steps. (In what follows, it is important to distinguish between a transaction and its component operations.)

The most important of these common steps are the synchronisation of different operations on the same object, and verification of the unique identifier presented by the client. To perform synchronisation with other transactions, Root calls Lockman.lockuid. Normally, the request can be satisfied. If the UID supplied was a TUID, then Lockman has checked that it is valid. If it was a PUID, then Lockman will return a newly created TUID which represents the interlock. In either case, the address of the transaction's intentions vector is also returned.

As indicated in the previous chapter, an intention on a disc block is represented by an intermediate allocation state in its cylinder map entry, and every transaction which allocates or deallocates disc blocks accumulates a number of such intentions. The intentions vector of a transaction is used to record the disc addresses of all blocks on which intentions have been made and is scanned to perform or remove all intentions when the transaction finishes.

It would be possible to arrange that the intentions manager Intman ran as a task, and controlled the central data structure made up of the intentions vectors. However, a more economical solution was adopted. Since the intentions accumulated by a transaction are identified with the interlock set on an object, it is natural for Lockman to maintain the intentions vectors. Whenever Root calls Lockman at the start of an operation, Lockman can return the intentions vector associated with that interlock. The address of the intentions vector is then put into a global variable in Root's task, so that it is accessible to Intman. This scheme is efficient without

violating the module structure. The decision of which intentions vector to associate with the operation (one for an existing transaction if a TUID is specified, and a new empty one if a PUID is specified) can be made at the same time as the checking or creation of the interlock. Thereafter, Intman can be called directly as a procedure, since the data structure it manipulates is a global variable in the task in which both it and Root run.

Having obtained an interlock, the Root module must gain access to the object tree. To do this, Root calls Objman.openobj supplying the object's PUID and obtaining in return a buffer window. A buffer window (bfw), is the universal method of describing ring or disc buffers. It defines a buffer by a base address and length, and a window within the buffer by a starting offset and window length. A bfw is thus a convenient way of describing a piece of a buffered disc or ring block. It is also an example of a data structure which carries a type tag to be checked at every module interface.

At this point, Root has all the information needed to perform the operation. The object is directly accessible through the buffered root block of the object tree, and the correct intentions vector has been found for the transaction. Root then makes a simple switch on the function code to execute the specific operation. Fileman or Indexman is called at this point, depending on the function code supplied to Root. After these calls have been made, a common termination sequence is executed. If the operation forms part of a transaction which has not yet completed (because a TUID was supplied, and the operation is not an ensure or a close), Root simply frees the buffered root block, returns the request message, and awaits another request.

If the operation is the last in a transaction, however, then the transaction must be committed or aborted. For a long interlock (defined by a TUID issued to a client), this decision is made by the client and passed as an argument of the ensure or close command now being executed. For a short interlock (defined by the client supplying a PUID on which an interlock has been set for the duration of this one operation only), the decision to commit is taken only if all call.modules performed to this point have succeeded.

In either case Root calls Intman.ensure with an indication of whether to commit or abort. If the decision is to commit, Intman will ensure that the disc representation of the object shows all intentions on the object, and that the object's commit bit is set. Immediately after this minimum of work is done, Root returns the request message to the caller. The instance of Root does not yet return to the idle state however. Intman.cleanup is called to deal with all intentions created on the object. The work done here is simply that which would be done on a restart after an interruption, and is asynchronous with

the client. Finally, Root frees the buffered root block of the object, whose bfw has identified the object to all preceding calls, and returns to the idle state.

5.7 Lockman

As indicated previously, Lockman manages the interlocks which synchronise transactions on the same object. It controls a central interlock table describing the interlocks currently set, and it also controls the pool of intentions vectors.

In a call of Lockman.lockuid, the actions taken depend on whether a TUID or a PUID is supplied. If the argument is a TUID, then it defines an entry of the interlock table. This entry is checked to make sure that the TUID is still valid. If so, then the TUID, the object's PUID and the transaction's intentions vector are returned.

If a PUID is specified, however, then a new interlock is being requested. In this case, two more arguments are examined. One defines whether the interlock is long or short. A long interlock is set by a client explicitly requesting an open operation. A short interlock is requested by Root when a command other than open specifies its argument as a PUID. Long interlocks may be set for an indefinite length of time, and a conflicting request to interlock must therefore be rejected. Short interlocks are held only for the duration of a single call of Root, so a conflicting interlock request can be queued with the certainty of being satisfied quickly. The second qualifying argument to Lockman.lockuid is whether an interlock to read or to read and write is being requested, since Lockman implements multiple-reader-or-single-writer interlocks.

There are two other entry points to Lockman. Lockman.extend is used when Intman tries to record an intention in a full intentions vector, and causes Lockman to allocate a fresh one from its pool. Lockman.unlockuid is called to invalidate a TUID. If this is the last outstanding interlock on the object, then the interlock table entry is released, and the intentions vectors are returned to the free pool. As a cross check, each intentions vector is examined to make sure that it has been emptied by Intman before the interlock is released.

Interlocks can be explicitly set by clients and forgotten, perhaps as the result of a crash. To prevent these interlocks persisting indefinitely, Lockman must control timeouts. Thus, when an interlock is created, a timer is started on the interlock table entry, and each time Root presents a TUID to Lockman.lockuid, the timer of the entry is reset. If a timeout expires, Lockman generates and sends an explicit Root.close request to abort the transaction. Note that there

is a dependency loop here, because Root will immediately call Lockman.lockuid to synchronise on the object. For this reason, and for convenient implementation of the timing mechanism, Lockman is explicitly written as an event-driven task. This means that it always uses act.module rather than call.module, and always waits for the arrival of any message rather than a particular message. Thus, although the dependency loop is present, deadlock will not occur, because Lockman is always ready to receive a new request. This method of removing potential deadlocks may be contrasted with that of Janson [Janson76] one of whose main criteria for a modular decomposition is the avoidance of dependency loops. Janson's method requires a fundamental restructuring of the module functions, where the use of an event-driven task only involves a different, but equally natural programming style.

5.8 Indexman

Indexman's function is simple. Objman provides access to objects as a sequence of fixed-size blocks. Objman.read and Objman.write can be used to examine and update individual blocks, and Objman.changeobjsize to extend or contract the object. Indexman must therefore simply translate the UID offset within an index to a block in object, word within block pair, and then access the appropriate block. The only complications arise due to the side effects of writing index entries. Specifically, Indexman.delete and Indexman.retain can cause a PUID to be removed from an index, and Indexman.changeindexsize can cause many PUIDs to be overwritten. These index operations must result in the deletion of these objects if they have become detached from the root index.

To deal with storage control, a combination of reference counts and garbage collection is used as was done for the CAP filing system [Birrell78, Wilkes79b, Dellar80a]. Maintaining a count of the number of index references to every object will detect most, but not all cases when an object should be deleted. To control use count deletion, Indexman calls the use count decrementer Ucdecr whenever a PUID is deleted from an index. When a PUID is inserted in an index, its use count is incremented synchronously before the index operation takes place. Indexman also calls Gci to inform the garbage collector of new index references in a manner to be described in section 5.20.

5.9 Fileman

Using the object abstraction of a sequence of fixed-size blocks defined by Objman, Fileman implements files as sequences of 16-bit words. Its main purpose is to make the inter-block divisions within a file invisible to clients of the file server, and to arrange that file transfers proceed as close as possible to ring speed.

When Fileman.read is called, the initial request block has already been accepted by Command and passed on to Root. Among the parameters for the read operation is a port number in the client machine to which the data should be sent. Fileman immediately calls Storeman.reservedisc to gain exclusive access to the disc, so that other operations are prevented from moving the disc heads during the file transfer. Then, the first block in which the requested data lies is read from disc by a call of Objman.read, and Fileman begins transmitting the data to the client by calls of Ringman.send. This is particularly easy using the buffer window abstraction mentioned previously. Since blocks of objects are the same size as the largest basic block, Fileman simply needs to get a bfw for a block using Objman.read, window it in the case of the first and last blocks of the transfer and pass it, with the destination station and port numbers to Ringman.send. To maintain speed, Fileman makes explicitly asynchronous requests to Ringman.send, and while the current ring transfer is in progress calls Objman.read synchronously to prefetch the next block of the object. Thus the transfer is effectively double-buffered.

The task of Fileman.write is more complicated. Again its first action is to reserve the necessary disc bandwidth by a call of Storeman.reservedisc. Then Fileman.write obtains a free file server port number from Ringman.allocateport, and then issues reception requests for several maximum size basic blocks from the client machine by using act.module on Ringman.receive. As soon as this is done, Fileman.write calls Ringman.send to send a reply basic block to the client. This block contains the newly allocated port number, and its arrival is the signal for the client to begin transmitting the data to the file server on this port number at maximum speed. The file server will then see a sequence of basic blocks containing the data to be written arriving on this port. These will not be confused with file server requests, because they will be directed to the port number chosen by Fileman.write, not to the command port.

The task of accepting data at high speed is complicated by two factors. Firstly, write operations can transmit large amounts of data, and the client is free to transmit these data in basic blocks of

any size. Secondly, the Ringman interface is explicitly in terms of basic blocks. Each call of Ringman.receive supplies a bfw for a buffer to be filled with precisely one basic block of any size less than or equal to the window size. This means that the bfws returning to Fileman from Ringman.receive do not necessarily correspond to blocks of the object which can be written immediately to disc. In general, the data arriving from the ring must be copied into block-sized buffers before being written to the object using Objman.write. Thus there are two cycles of buffers controlled by Fileman.write. One set circulates between Fileman and Ringman, with the intention that at any time, one is being filled from the ring, one is having its checksum verified by Ringman, and a third is being dealt with by Fileman. The second cycle consists of a single buffer moving between Fileman and Storeman via Objman; this contains a block's worth of ring data copied from the ring buffers which is written to disc when full.

Frequently, however, arriving basic blocks do coincide precisely with object blocks to be written. Clients wishing high transfer rates will tend to use maximum-size (and therefore disc block-size) basic blocks, and often a transfer will start at the first word of a file. The CAP, for instance, swaps complete files out of its memory in this way. In this case, the copying of data can be avoided and the ring buffers written immediately to disc. The current implementation of Fileman.write, due to R.M. Needham and M.A. Johnson, is careful to make this optimisation whenever possible.

The difficulty of writing to a file at high speed is largely due to the primitive nature of the interface to the ring. It would be much more efficient if the ring interface were able to split an incoming basic block between two buffers. Then, Fileman would be able to submit a stream of buffers to Ringman without needing to concern itself with the sizes of the basic blocks, and these buffers could be written directly to disc. At the time of writing, a microprocessor-controlled implementation of such an interface is being tested in the Computer Laboratory [Gibbons81].

In summary, reading from a file involves a standard request and reply block exchange with an intermediate transfer of pure data basic blocks from server to client. Writing is slightly more unorthodox, involving two replies to the original request block. The first is Fileman's signal to transmit. The second arrives from Command after the data block stream has been accepted from the client, and is used to indicate whether the data arrived correctly.

When less than a basic block of data is to be read or written, Fileman.read and Fileman.write involve an unnecessary number of network exchanges. The data for a short read could be transmitted in the return code block instead of being sent in a different block to another port. Similarly, the data for a short write could be appended

to the initial request block. For this reason, Fileman has two simpler entry points Fileman.SSPread and Fileman.SSPwrite which are semantically identical to Fileman.Read and Fileman.write, but use the "single shot" protocol used by all other file server operations. An SSPread request from a client supplies a request block and receives a combined data and reply block. An SSPwrite request supplies the data to be written in the request block and receives a single return code block in reply. These functions use a simpler protocol, but usually involve the client copying data to and from a communications buffer. The CAP, for instance, can read a stream of pure data basic blocks into the required segment of real store, but would be obliged to remove the protocol information from an SSPread reply by copying.

5.10 Ucdecr

The use count decrementer maintains a queue of UIDs of objects whose counts of index references are to be decremented at some convenient time. It is called by Indexman when a UID is removed from an index, and calls only Root.decrementuc to decrease the object's use count and delete the object if the new value is zero.

The file server must be careful not to delete any objects as the result of an index transaction which fails. If a transaction containing a delete operation aborts, for instance, Ucdecr must not decrement the use count of the object which has been restored to the index. To prevent this possibility, Ucdecr must have some knowledge of which transactions have committed and which are still in progress.

Whenever Root obtains an interlock on an index, it calls Ucdecr.stop to suspend the decrementing of use counts. Until Ucdecr.start is called by Root at the end of the transaction, Ucdecr will record UIDs but will not attempt to call Root.decrementuc. The only parameter of Ucdecr.restart is an indication of whether the index transaction committed or aborted. If it aborted, Ucdecr scans its queue and removes all entries made since Ucdecr.stop was called. Entries may be removed needlessly by this simple mechanism, but the garbage collector will detect any garbage objects whose use counts are high.

As in the case of Lockman, there is a potential for deadlock between Ucdecr and Root; if an index UID is deleted from an index, for instance, then Ucdecr's call of Root.decrementuc will immediately cause Root to call Ucdecr.stop to signal the start of an index transaction. The problem is handled in the same way in this case, not by altering an otherwise adequate module structure, but by implementing Ucdecr as an event-driven task.

5.11 Objman

Objman implements the object abstraction of a random access sequence of fixed-size blocks. A block of an object is selected by supplying a bfw for its root block and the number of the block to be accessed.

To be able to use an object, Root must first convert the PUID for it into a buffer for the root block of the object tree. This is done by a call Objman.openobj, which is passed the UID of the object to be accessed and a vector in which the output bfw can be constructed. Objman uses the PUID to make the disc address of the object and also its required cylinder map entry; the block must be allocated and marked as a root block with the same PUID. Storeman.useblock is then called to load the block into the cache and return a bfw. If the block was already in the cache, Storeman will return its cylinder map entry as well, since these are stored with buffered blocks. If it was not in the cache, Objman calls Cmapman.blockstatus to return the cylinder map entry. This may cause a second disc transfer if the cylinder map is also not in the block cache. Objman can then compare the returned cylinder map entry to that which it derived from the PUID to decide on the validity of the PUID. Only if there is an exact match is the bfw returned to Root.

From this point on, all Objman operations invoked by Root, Indexman, or Fileman are passed this bfw as a handle for the object. Objman.read, for instance, accepts a bfw for the root block and a block number and returns a bfw for the data block of the object.

Objman.write accepts a bfw for the root block, a bfw for the data block, and a block number, but as opposed to Objman.read, behaves differently for normal and special objects. By consulting the object tree, Objman can determine whether or not a data block already occupies the block position. Only if the object is normal and there is an existing block is the write done in place; a call of Storeman.map associates the data buffer with the disc address and causes the disc write to occur. When the object is special and data blocks can never be written in place, or the object is normal, but the block position is empty, Objman must choose a new block to which to write the data. This is done by a call of Cmapman.allocate, which finds the nearest free block, changes its map entry in a buffered copy of the cylinder map, and returns the disc address. Objman now calls Intman.record to register the creation of a new intention due to the current transaction, and writes the new disc address into a buffered copy of the appropriate block of the object tree. If there was a previous occupant of this position, Cmapman.deallocate is called to

mark it intending-to-deallocate, and Intman.record is called again to register another intention. The data are finally written to the newly allocated block.

A single Objman.write operation can therefore create zero, one or two intentions invisibly to the caller, who cannot distinguish normal and special objects at the Objman interface. A sequence of Objman.writes will accumulate a number of intentions for the transaction. These correspond to buffered cylinder maps with modified entries and to buffered object maps containing new disc addresses. Both these sets of blocks have simply been marked dirty in the block cache but are not up to date on disc. During each call of Objman.write, only the data have been written immediately to disc. The task of ensuring that all this buffered structural information is correctly written to disc at the end of the transaction is performed by Intman on explicit calls from Root.

5.12 Cmapman

The cylinder map manager is responsible for interpreting the contents of cylinder maps and for ensuring that block allocations and deallocations pass through the correct intermediate states.

Cmapman.allocate is the entry point used by Objman to obtain a new block. The arguments supplied by Objman are a "home" disc address which is the block's optimal position, and the cylinder map entry which the block should have.

To find a cylinder map on which to search for a free block, Cmapman calls Blockman.choosecyl. Blockman maintains a bit map of the cylinders which are not full, and uses this to construct the disc address of the nearest cylinder map with a free block. By a call of Storeman.useblock, Cmapman can then obtain a bfw for the cylinder map. It is essential that this access be exclusive, since concurrent and apparently independent operations can require the same cylinder map to be modified if the objects involved have blocks on the same cylinder. For this reason, Storeman provides both exclusive and shared access to disc blocks, and Cmapman always requests exclusive access when a cylinder map is to be updated. Because Storeman provides the interlocking on cylinder maps, Cmapman is called as a procedure in all tasks which require it, all synchronisation between its instances being handled by the single Storeman task.

After obtaining a bfw for the cylinder map, Cmapman searches for a free block within the array of cylinder map entries. If the cylinder is the same as that on which the home disc block is allocated, a block is chosen which will minimise rotational latency during access. Since

Objman defines the home disc address for block N as the block at block N-1, this choice tends to favour the serial access of long file transfers. For objects with holes where block N-1 does not exist, the root block is used as the home address.

Once a block is chosen and its cylinder map entry changed, Cmapman calls Blockman.cylstatus to inform it of the number of free blocks now on this cylinder. Storeman.freebuffer is then called to release exclusive access to the buffered cylinder map. An argument of this call causes the cylinder map to be marked dirty in the block cache to ensure that it will eventually be written to disc.

Cmapman.deallocate is used to change a block from allocated to intending-to-deallocate. Again, Storeman.useblock is called requesting exclusive access to the block's cylinder map. The cylinder map entry is then marked intending-to-deallocate and the buffer is released and marked dirty in the block cache.

Cmapman.blockstatus is the entry point used to find the cylinder map entry of a block. It is used by Objman.openobj to verify the PUID of an object, and by Restart as it scans the cylinder maps looking for intentions. For this entry point Cmapman requests shared access to the block's cylinder map.

Cmapman.searchcyl is used by Intman to clear up intentions. The arguments supplied are the disc address of a cylinder map, the PUID of an object, and a procedure variable. Cmapman loops through the cylinder map searching for blocks in one of the intention states which belong to the PUID supplied. For each such block found, the procedure argument is called with the block's cylinder map entry and disc address as arguments. This procedure - always in practice supplied by Intman - is then free to mark the cylinder map entry allocated or deallocated as required by the transaction's commit bit. At the end of this loop, Cmapman notes whether any blocks have been freed on this cylinder, and calls Blockman.setcyl to inform it of the new block usage.

Cmapman's final entry point is Cmapman.intention which is called by Intman to turn an object's commit bit on and off. For added security, Intman supplies the full PUID, which Cmapman checks against the root block's cylinder map entry before allowing the commit bit to be modified.

5.13 Intman

Intman is responsible for filling and emptying the intentions vectors associated with a transaction. The natural representation for an intentions vector is a list of disc addresses of blocks on which

intentions have been created. Unfortunately, this is too costly to be used in practice. A transaction may create a large number of intentions, and a single write operation by the CAP microprogram dumping the contents of CAP store to a new file can create over 1000 intentions. To reduce the space required for intentions vectors, therefore, each entry contains only the disc address of the cylinder on which an intention was created. Since newly allocated blocks will tend to be close, a single intentions vector entry may represent many block intentions created by the transaction. To find the individual cylinder map entries for these intentions, Cmapman.searchcyl is used as described above.

Intman.record is the entry point used by Objman after creating an intention on a block. From the disc address supplied, Intman computes the cylinder number and inserts it in the intentions vector if it is not already there. It may happen, of course, that the intentions vector is full. If this happens, Intman calls Lockman.extend to add a new vector to the queue associated with the transaction.

Intman has only two other entry points, both of which are called by Root when a transaction ends. Intman.ensure does the minimum possible amount of work to allow a reply to be sent to the client. The problem here is that an interruption may occur immediately after the reply is sent, and it is essential that the actions taken by Restart correspond with the reply given to the client. A transaction must not be undone during recovery if the client had been informed that it committed.

When Intman.ensure is called, the actions taken depend on whether the request is to commit or to abort the transaction. Aborting takes no work, since the default state of the disc representation is to reverse all intentions. If the decision is to commit, however, then all intentions must first be written to disc. This is done by calling Storeman.write once for each intentions vector entry. Then a call of Storeman.flushobject requests Storeman to search its cache for any blocks whose buffered cylinder map entries identify them as belonging to the object, and to write any such blocks to disc. At this point, all intentions have been written to disc, but the object tree is "ahead" of the cylinder maps since the commit bit has not been set.

Intman.ensure now examines the intentions vector to see if the remainder of the commit operation can be optimised as described in section 4.5. This can be done when all intentions reside on the same cylinder as the root of the object tree. If this is the case, then Intman.ensure completes the commit operation immediately. Cmapman.searchcyl is called passing it the PUID of the object and also a procedure which will perform all intentions when it is called. At the end of the call, therefore, the buffered cylinder map reflects the correct new allocation state of the object. The cylinder map is then written to disc, and the intentions vectors are marked empty.

If intentions exist on cylinders other than the home cylinder, however, then the full commit sequence must be executed. Intman.ensure sets the object's commit bit by a call of Cmapman.intention, forces the cylinder map to be written and returns, leaving the remainder of the commit operation to Intman.cleanup.

Intman.cleanup will only have work to do if a transaction has aborted and there are intentions to be removed, or if a multi-cylinder transaction has committed. In the optimised case of a single-cylinder transaction, the intentions vectors will be in the same state as for a read-only transaction, showing that no intentions have been created.

Undoing an aborted transaction is more complicated than finishing a committed one because changes made by Objman to the object tree must be undone. Under most circumstances, these changes will not have been made on disc because the modified and dirty map blocks will have remained in the cache for the duration of the transaction. If the cache is being changed very frequently, however, then Storeman may need to write some of these dirty blocks to disc. Intman.cleanup must therefore be careful to reset the object tree to its original state before removing any intentions. It does so by a call of Cmapman.searchcyl for each cylinder map in the intentions vectors. Instead of passing a procedure to confirm or remove each individual intention, however, Intman passes a procedure which removes the effect of the intention from the object tree. If the procedure is called with a cylinder map entry for an intending-to-allocate block, it uses the tree position stored in the entry to remove that block's address from the object tree. Conversely, if a cylinder map entry for an intending-to-deallocate block is given to the procedure, it ensures that the indicated position in the object tree contains the disc address. Both these modifications to the object tree are made by the procedure calling Objman.coerceobject, an entry point provided especially for this purpose.

The calls of the procedure variable will not change any cylinder map entries, but may leave blocks of the object tree dirty in the cache. At the end of the pass over the intentions vector, therefore, Storeman.flushobject is called to write all these map blocks to disc, thus ensuring that the object tree is in its original state.

Once this has been done, the paths for committing and aborting rejoin in Intman.cleanup. The intentions vectors are scanned with another call of Cmapman.searchcyl for each cylinder map found. This time, however, the procedure argument changes the states of blocks in the cylinder maps according to the value of the transaction commit bit, either removing or performing all intentions. The cylinder maps are then written to disc and the intentions vectors are marked empty. Finally, if the transaction committed, the commit bit is reset by a call of Cmapman.intention and its cylinder map is written to disc.

5.14 Blockman

Blockman maintains the structure used to find free blocks near to some optimal position. This is a bit map showing which cylinder maps have at least one free block. In fact for each disc there are two bit maps, since blocks come in two sizes, small (242 words) and large (1024 words). Large blocks are used for leaf blocks of two- and three- level objects, and so always hold the data of a file or an index. Small blocks are used for all structural information; cylinder maps and non-leaf object map blocks. The size of large blocks was chosen to be that of the largest possible basic block, to simplify file transfers. The peculiar size of small blocks was chosen to minimise the wastage on each track of the discs.

Blockman has three entry points, newpack, choosecyl and setcyl. Blockman.newpack is called by Restart when Diskman informs it of the disc drives which are accessible as one of the first steps of the restart sequence. This entry point causes Blockman to allocate two new bit maps and to associate them with the number of the mounted disc pack.

Blockman.setcyl is used to inform Blockman of the allocation state of a cylinder. It has three parameters consisting of a disc address, and the number of small and large blocks in use on the cylinder. These numbers are used to set the bit map entries for the cylinder. This entry point is called initially by Restart as it scans the cylinder maps, and thereafter by Cmapman whenever blocks move to or from the deallocated state.

Blockman.choosecyl is called whenever Cmapman wishes to allocate a block. Its arguments are an optimal disc address for placing the new block, and an indication of whether a small or a large block is wanted. This allows Blockman to select an entry within a bit map, from which a search is made radially outwards until a cylinder with a free block is found. The output disc address can then be constructed. Note that the bit map cannot be immediately updated since Blockman cannot know whether the allocation of a block will exhaust the free blocks on the cylinder. It must therefore await a call of Blockman.setcyl.

Because Blockman controls a central data structure, it must run in a single instance in its own task.

5.15 Storeman

The Store Manager is another module which controls a central data structure. Storeman is responsible for providing buffers for its clients and for maintaining a record of which buffers hold current copies of disc blocks.

The most basic Storeman functions are concerned with allocating and freeing buffers. Storeman.usebuffer returns a bfw for a small or a large buffer; these correspond in size to small and large blocks. Storeman.freebuffer is used to release a buffer specified by its bfw.

To obtain the contents of a disc block, Storeman.useblock is called. The caller must supply a disc address, an indication of whether exclusive or shared access is required to the block and optionally a cylinder map entry to be recorded in the block cache*. Storeman.useblock returns a bfw for the block, which will have been read from disc unless it was already present in the block cache. Buffers obtained from Storeman.useblock, like those obtained from Storeman.usebuffer, are returned to the free pool by a call of Storeman.freebuffer. In this case, however, two additional parameters are significant. One is an action which Storeman is to perform before releasing the buffer; the choices are (1) to do nothing, (2) to mark the buffer dirty, and (3) to write the buffer to disc synchronously. The second parameter indicates whether Storeman should remember the mapping between the buffer and the block, or should consider the buffer to be empty. These parameters allow Objman to direct the efficient use of the block cache.

The procedure Storeman.map is used to make an association between a buffer and a disc block. As for Storeman.useblock, a disc address and a cylinder map entry are supplied. The caller also specifies an action as for Storeman.freebuffer. This may be one of: (1) read the block into the buffer, (2) write the buffer to the block, (3) mark the buffer dirty, or (4) do nothing. Storeman.map is the procedure used by Fileman (via Objman) to fill and empty its circulating buffers during long file reads and writes.

* As mentioned previously, if buffered disc blocks are tagged with their cylinder map entries, it becomes possible to find all buffered blocks belonging to a given object. This is necessary in two circumstances. When committing a transaction, all modified map blocks must be written to disc. When deleting an object, changing its size, or aborting a transaction, all knowledge about encached blocks of the object must be destroyed.

As mentioned previously, it is necessary for Intman to ensure that all modified object map blocks in an object are up to date on disc. The entry point which performs this service is Storeman.flushobject, whose function is to scan all buffers in the cache looking for those whose buffered cylinder map entries identify them as containing blocks of the object. Any buffered blocks found are either written to disc or deleted as requested by the caller.

The other major entry points of Storeman have to do with reserving and releasing the discs. Fileman, it will be recalled, obtains exclusive access to a disc drive before starting a file transfer to ensure that the transfer can proceed at ring speed. This is done by bracketing the file transfer with calls of Storeman.reservedisc and Storeman.releasedisc. Disc reservation is not an essential part of file transfers, but serves to improve performance by preventing other tasks from disturbing the position of the disc heads while the transfer is in progress.

It might be thought that functions concerned with the right to perform disc accesses should be included in the disc manager Discman, rather than Storeman. This was the approach taken in the initial implementation but it resulted in an unforeseen source of deadlock. During file writes, which occur under the disc interlock, Objman.write may call Cmapman.allocate to allocate a new block. Cmapman will request exclusive access to a cylinder map to do this in a call of Storeman.useblock. It may happen, however, that another task has requested that the cylinder map be written out, perhaps while cleaning up intentions. In this case, Storeman will have locked the buffer, since it must not be modified during a disc transfer, and will queue the new Cmapman request until the disc write has finished. This write request, however, will have been suspended by Discman because the task cleaning up intentions does not own the disc reservation. Thus, the owner of the disc reservation waits for a disc block, while the owner of the disc block cannot proceed because of the disc reservation. This particular sequence of events tended to cause a deadlock about once a day.

This problem was overcome in the first implementation by adding a path of communication between Storeman and Discman, so that Storeman could inform Discman whenever a caller was suspended waiting for access to a block. Discman could then check whether it had blocked a disc transfer for this address due to a disc reservation, and would release the transfer to avoid deadlock.

In a second implementation of Storeman and Discman, the responsibility for all scheduling was transferred to Storeman with consequent simplifications in the program structure. Storeman deals immediately with any request it can satisfy without accessing the disc, such as a request for a block already in the cache, or a call to

write a clean block. Before each disc access, however, Storeman checks that its caller is able to use the disc; either it must own the disc reservation or there must be no reservation in force. If the request cannot be allowed, the caller's request message is put on a disc queue. Only if the caller is permitted access to the disc is the requested buffer checked. If some other task has acquired exclusive access to it, or if a disc transfer is in progress on it, the request message is queued on the buffer. As a side effect of this queueing, however, Storeman checks whether the task whose request is now being suspended is the owner of the disc reservation. If it is, then the disc reservation mechanism is temporarily suspended, and all requests waiting on the disc queue are immediately restarted. When the request suspended on the buffer is later released, Storeman will again notice that this is the owner of the disc reservation, and will resume disc scheduling. The net effect is that the owner of the reservation has exclusive access to the disc until it must wait for some other task to make a disc access. Then the disc interlock is temporarily broken to allow the waiting task to release the disc block. Thus disc scheduling is efficiently performed most of the time, with occasional discontinuities due to unpredictable conflicts over cylinder maps.

The cache managed by Storeman is about 50 blocks in size, of which perhaps 40 are small (256 words) and 10 are large (1K words). The proportion of small to large buffers is arranged so that there are enough large buffers for file transfers, but most of the space is used for maintaining copies of structural information - the object and cylinder maps. To allow fast searching and modification of the cache, Storeman maintains a fairly complex internal structure. Every buffer is defined by a buffer descriptor (bfd) which contains (1) the base address and length of the buffer, (2) a disc address and a copy of its cylinder map entry if known, (3) a queue for Storeman requests which must wait for this buffer to be released, and (4) some status information. The status information consists of a use count of bfw's issued on this bfd, the dirty/clean bit and a flag to indicate that a disc transfer is in progress.

Each bfd can be threaded on zero, one, or two doubly-linked queues. The disc address link is use to link all bfd's on a single bucket of a hash table keyed on disc address, and the presence of a bfd on one of these queues means that the buffer holds the most recent copy of a disc block. The bfd will also be threaded on a free queue if its bfw use count is zero. Thus a bfd on no queue is in use but doesn't hold a disc block, and is probably being used for ring reception. A disc block in use will have a bfd on a disc address queue, and when it is released by Storeman.freebuffer, its bfd will also be threaded on the free queue.

This structure allows the most frequent operations to be very fast. Searching the cache for a particular disc block requires examination only of the two or three bfds linked on a particular hash bucket. Choosing a new free buffer is done by taking the first bfd off the queue of empty free buffers, or if there are none, off the queue of full free buffers. Freeing a buffer is simply a matter of queueing its bfd on the end of the queue of full or empty buffers.

5.16 Discman

The disc manager task has two main entry points Discman.read and Discman.write which read and write individual blocks. Each takes as arguments a bfw describing the store buffer and a disc address. Discman's principal functions are to translate a pack number in a file server disc address into a drive number acceptable to the disc device, and to retry disc transfers after errors. To do this, Discman manages a central table of identification blocks. Each disc pack has an identification block in a fixed physical address, and as part of its initialisation sequence, Discman attempts to read the identification block from all possible unit numbers. Since each identification block contains the disc pack number, at the end of this process Discman will know which packs are mounted on which drives.

The identification block also collects the "believe deletions" bit which is used in garbage collection as described in section 5.22. To allow the garbage collector interface Gci access to the identification block, Discman provides two entry points. Discman.readid returns a bfw on the identification block of a particular pack, and Discman.writeid causes it to be written to disc.

5.17 Ringman

The ring manager task organises the sending and receiving of basic blocks on the ring. It provides an interface between the ring transmitter and receiver devices, Ringtx and Ringrx, which operate in terms of basic blocks, and Ringman's clients which send and receive bfws of data.

Ringman.send is used to transmit a single basic block. Its arguments are a bfw defining the contents of the basic block, and a ring identifier consisting of the destination station and port number. Ringman first enlarges the window by two words at the front and one at the end of the buffer, (all buffers issued by Storeman are

windowed so as to leave these words free) and uses these words to construct the header, route and checksum packets of the basic block protocol. Computing the checksum is a relatively expensive operation, since it can require over 1000 additions but is essential for error control. The request is then entered in Ringman's transfer table, and a corresponding request message is sent to Ringtx. When Ringtx returns, Ringman replies to the caller after readjustment of the window.

Ringman.receive operates in much the same way. The supplied bfw is widened to allow reception of the three protocol words, the request is entered in the transfer table and a request to receive is issued to Ringrx using act.module. When Ringrx returns its request message Ringman verifies the checksum of the block and replies to the caller.

Under certain circumstances, a Ringtx or Ringrx request will never return to Ringman. Ringtx will try forever to send a basic block to a station which is switched off, and Ringrx will wait indefinitely to receive a basic block which is never sent. For this reason, both Ringman.send and Ringman.receive have a third timeout argument which defines how long the caller is willing to wait for a reply. Ringman measures time in the same way as Lockman, by sending a message to the the interval timer service of the Clock device. At each "tick", defined by the return of this clock message, Ringman scans its tables looking for transfers which have exceeded their time limits. If one is found, the corresponding device message is retrieved by Ringman, and a "timed out" return code is sent to the caller.

The timeout facility is not sufficient in all circumstances, however. During a file write, Fileman will be issuing repeated calls of Ringman.receive to accept the file data. Towards the end of the transfer Fileman will know how many words it expect to receive, but not in how many basic blocks they will arrive, since this is at the transmitter's discretion. Thus, when the last basic block arrives, Fileman will have a number of Ringman.receive requests outstanding which will never be satisfied. To allow these to time out would be inefficient, since the file operation would be effectively suspended during this period. Therefore, an entry point to Ringman to cancel an outstanding request is needed. Ringman.cancel takes the address of a Ringman request as its argument and performs the same actions as for a timeout; the device message corresponding to the request is retrieved from the device, and the cancelled request is returned to the caller.

5.18 Ringtx and Ringrx

Because the Cambridge ring provides a full duplex interface

allowing simultaneous reception and transmission, the simplest organisation of the driving software is as two independent devices. In the file server, these devices are the ring transmitter Ringtx and the ring receiver Ringrx.

Transmission on a local network such as the Cambridge ring is fundamentally simpler than reception. If a number of basic blocks are to be transmitted, then under normal circumstances it will be sufficient to transmit them one by one in order of submission* A simple queue of requests can be handled in first-in-first-out order. Reception must be handled differently, however, because the order of arrival of basic blocks on the ring is independent of the order of submission of requests to receive. A ring receiver must be programmed in an event-driven rather than a serial manner so that it can begin reception of any one of the basic blocks it has been told to accept, not merely the first one.

Ringtx, therefore, is a simple device programmed in the standard TRIPOS manner. For each request to transmit a basic block, it cycles the transmit side of the ring station through a sequence of states: set destination station number; send header and route packets giving up only on "ignored"; send remainder of block aborting on "ignored" or "unselected".

The ring receiver Ringrx must link Ringman's requests to receive on a chain of outstanding receptions. When a basic block arrives from some source, Ringrx accepts the header and port packets. These will define the length of the basic block, and to which port number it is directed. By interrogating the ring station, Ringrx can also determine from which station these packets arrived. Using this information, Ringrx searches the queue of outstanding requests to decide if the arriving basic block matches one of them. If so, the remainder of the basic block is accepted into the buffer provided; if not, Ringrx sets the select register to zero briefly to discourage the transmitter, and restarts the entire reception sequence. It is important that Ringrx search its internal queue of requests in order of arrival, since Fileman will attempt to maintain a queue of reception requests during a file write, and filling the request buffers in the wrong order would cause the file to be written out of sequence. Note that the simpler scheme of Ringrx accepting any basic block which arrives and passing it back to Ringman for the decision to accept or discard it cannot be used if it is desired to use the simple "unselected" flow control mechanism - the decision whether to accept

* In theory, a better job can be done by interleaving the transmission of ring packets of different basic blocks, especially if the receivers are of greatly differing speeds. In practice it is difficult to do this quickly enough to make it worthwhile.

or reject must be made before all of the basic block arrives.

5.19 Restart

Restart is the module which examines the discs for unfinished transactions and ensures that each object is in a consistent state. Loaded in the initial bootstrap, it is present as a single instance running in its own task. After performing the restart sequence in the first 30 seconds or so of a run of the file server, Restart deletes itself and its task to make more room available for Storeman's cache.

Restart has a single entry point, Restart.start, which is called by Discman after it finds which packs are mounted on the disc drives. Restart then immediately calls Blockman.newpack once for each pack mounted. This causes Blockman to allocate a bit map for the small and large blocks of each drive. Blockman will of course have no knowledge as yet of the allocation state of the cylinders, and it is another of Restart's functions to call Blockman.setcyl to initialise the bit maps.

Restart is programmed as an event-driven task, not to break dependency loops as in the case of Lockman and Ucdecr, but simply to overlap execution of the Restart code with disc accesses. After calling Blockman.newpack, Restart immediately uses act.module to issue Storeman.useblock requests for the first few cylinder maps of each disc, and then waits for the first of these to return.

As each cylinder map is accessed, Restart scans it looking for blocks in one of the intention states. If it finds none, it simply passes the number of used and free blocks on the cylinder to Blockman.setcyl so that the allocation state of the cylinder becomes known. If it finds one or more intentions, however, then it must remove them. To do this, it uses the facilities of Lockman and Intman in the normal way. For each intention found, Restart uses the disc address of the root block of the object to retrieve its cylinder map entry holding the commit bit. It reconstructs the object's PUID, and uses it to search an internal table. This table holds information about each inconsistent object found by Restart, and contains the commit bit and intentions vector. If the table search fails, Restart adds a new entry by calling Lockman.lockuid to obtain an intentions vector. In either case, Intman.record is called to add another intention to the vector.

This is done for each cylinder of each accessible pack. At the end of the disc scan, therefore, Restart will have accumulated a description of the transactions in progress at the time of the interruption. Restart now calls Intman.cleanup for each transaction

to deal with the intentions in the normal way. Each interlock is then released by a call of Lockman.unlockuid and the file server is now in a proper initial state, with no transactions in progress and the allocation state of each pack known. The Restart module then releases all store used by its code, data, stack and global vector, and normal service is resumed.

5.20 Gci

Gci is the garbage collector interface, a set of minimal functions which allow asynchronous garbage collection from a remote machine. The remote garbage collector is a program written by N.H. Garnett [Garnett80] and is not described in detail here.

The task of garbage collecting the file server's graph structure is similar to that found in the original implementation of the CAP operating system [Birrell78]. The essential requirements are that:

1) there be some means for the garbage collector to discover the universe of objects among which references are possible,
2) the collector be able to discover all significant references, and
3) the garbage collector be informed of the creation of new references during its execution.

The file server graph structure has a degree of freedom not found on the CAP's directory structure. In the CAP, once the collector had finished its marking phase, and had found an object for which there were no references and which was not active in the virtual memory, it knew that the object was inaccessible because there was no way for any program to obtain a capability for the object. All such objects could therefore be deleted at the collector's leisure. In the file server, this cannot be assumed; a client which has stored away the PUID of an object may choose to retain it in an index at any time, perhaps after the collector has decided that it is garbage. This will lead to unpredictable results for detached cyclic structures, since a client may retain a reference to such a structure after it has been partially deleted. The retention should clearly only be allowed if the structure is complete, and should be forbidden otherwise.

The approach taken to this problem in the file server is another application of the basic two-phase commit protocol, and is designed to make the deletion of all garbage objects atomic. Each object has a deleting bit stored in the cylinder map entry of its root block for the use of the collector. Gci, under direction from the external collector, will call Cmapman.deleting to set and unset this bit for a particular PUID. In addition there is a single per-pack believe deletions bit; one bit per pack is used because each disc pack is a

self-contained graph with its own root index. This bit is contained in the pack identification block and Gci will set and unset this bit by calling Discman.writeid on instruction from the collector. Gci will also delete an object for the collector. It does this by calling Root.decrementuc for the object, specifying that its use count is to be decreased by a very large number.

The general outline of a garbage collection is as follows. The initiative for collection is taken by Gci, perhaps as the result of an index use count decreasing to a number greater than zero. (This is a necessary, but not sufficient condition for the creation of a detached cyclic structure, as Birrell and Needham point out [Birrell78].) Gci then requests a resource manager on the network to start the external garbage collector program running in a free processor. There is of course a security problem here, since the file server must assume that its external collector is benign and not a malicious imitation. These security issues are beyond the scope of this thesis, and are examined in Needham78. In this particular case, Gci must trust the name lookup server [Wilkes80] to give it the network address of a trustworthy resource manager.

As the external collector starts up, an initial exchange with Gci on a file server port number reserved for this purpose informs the collector of the pack number to be collected, the PUID of the pack's root index and the value of the believe deletions bit. Gci retrieves all three data from the pack identification block using Discman.readid. Under normal conditions the believe deletions bit is off. The collector now discovers the universe to be examined by asking Gci to send it copies of each cylinder map on the pack. From these it can determine the PUID of every object, and also the current state of its deleting bit. If any deleting bits are found on, the collector will instruct Gci to remove them during this pass. Note that the exposure of cylinder maps is potentially a very serious breach of security. This service is only available to the machine known to Gci, and reliance is placed on the inability of third parties to read basic blocks destined for the collector.

After the collector has built a list of the PUIDs of all objects, it can scan all indices that are reachable from the root index. It does this by retrieving entries from indices in the normal way, exactly as any unprivileged client would. At the end of this scan it will have discovered the objects which are unreachable from the root index, and will call Gci to set the deleting bits for these objects. When this has been done the collector requests Gci to change the believe deletions bit by writing the pack identification block to disc. This makes all deletions effective simultaneously, as will be described shortly. The collector now requests Gci to destroy each of these objects in turn; Gci checks to make sure that the object's

deleting bit is set and that the believe deletions bit is also on before calling Root to destroy the object. Finally, the collector directs Gci to turn the believe deletions bit off for the pack.

Because the collector is asynchronous, cooperation is required from several modules of the file server. Whenever Objman.openobj is called to validate a PUID, it examines the deleting bit in its cylinder map entry. If this is off, then the access proceeds as usual. If it is on, however, then the pack's identification block must be checked by a call of Diskman.readid. If the believe deletions bit is off, then the access is allowed. If it is on, however, then "invalid UID" is returned to Root. Thus, all the garbage collector's deletions become effective simultaneously when the believe deletions bit is set.

The third condition for asynchronous garbage collection was that the collector must be informed of any new retentions while it is in operation. To do this, each time Indexman is about to preserve a PUID in an index, it tests a system-wide flag which Gci sets to indicate that a collection is in progress. If the flag is set, then Indexman calls Gci.recordintention passing the PUID. Gci then sends this PUID to the collector. None of these "retention messages" must be lost in the network, of course, and a particularly lightweight protocol due to R.M. Needham is used [Garnett80]. The garbage collector treats the arrival of such a message as it would the detection of an index entry; the object is marked as connected to the root index. If the object's deleting bit has been set, the collector must unset this by a call of Gci. If the object is an index, then it must now also be scanned in the normal way. The believe deletions bit cannot have been set at this point, or the index retention would have failed when the PUID to be retained was validated.

Turning the believe deletions bit is the only operation during which the file server and its garbage collector must be completely synchronous. What must be avoided is setting the bit just after access has been allowed to preserve a garbage object, but before the corresponding retention message has been dealt with by the collector, because a PUID for a deleted object would thus be retained. When the believe deletions bit is to be set, therefore, the collector includes in the request the number of the last retention message dealt with. On receipt of this request, Gci calls Command.suspend. This call only completes when all instances of Command are idle, so that no index retentions can possibly be in progress. Gci then compares the sequence number of the last retention message sent to the collector with that of the last message it processed. Only if the two match is the believe deletions bit set. In either case, Command.resume is immediately called to restart service, and the collector is informed of the result of the comparison. In case of failure, the collector retries the operation after processing the outstanding retention

messages.

Finally, it is important to note that garbage collection is proof against interruptions by either party, since the state of the collection is recorded entirely in cylinder maps and the identification block. It is thus always safe for either party to abandon a collection at any time, because the next run of the collector will be able to deduce the correct state by reading the cylinder maps. If the believe deletions bit is on, the collector must destroy all remaining objects marked for deletion. If it is off, the collector must remove all deletion intentions before starting the normal collection sequence.

Chapter 6

A Revised Transaction Mechanism

As indicated in chapter three, the file server was explicitly designed to perform atomic transactions on a single object at a time. This choice has been satisfactory in an environment where the main clients of the file server are general-purpose operating systems. As shown in section 3.6, in more general contexts, this would not be true, since there would be no way to maintain consistency in multiple-object data structures over interruptions. A data base client, for example, would be obliged to store all its information in a single special file, or would have to construct a higher level transaction mechanism following Paxton's example [Paxton79].

This chapter explores the changes which would be required to make a more general transaction mechanism. In a first step, the transaction mechanism is extended to allow many objects to be updated atomically. The modifications result not only in a more general mechanism, but also one that is more efficient than the current implementation; equivalent transactions will generate fewer disc transfers under the new scheme.

In a second step, the transaction mechanism is further generalised to allow several file servers to participate in a single atomic transaction. The method followed is that of the Juniper file server [Sturgis80], and requires only minor changes to the structure derived from the first step.

6.1 Changes To The Client Interface

In the current transaction mechanism, there is an identification between a transaction and an object open for writing; both are represented by a TUID. These concepts must be distinct at the client interface if multiple-object transactions are to be specified.

The simplest way to make this distinction would be to create a transaction abstraction represented by a transaction identifier, or TID. Operations start and commit transaction would be needed to perform the functions of open and close, respectively, and open and close would be restricted to the control of interlocks only. A client would call start transaction to obtain a TID, open a succession of objects under this TID, modify some of the objects, call commit transaction on the TID, and release all interlocks with close

operations. A more elegant variation of this protocol is used in the Juniper file server [Sturgis80], which uses the TID supplied in each read and write request to lock objects implicitly, rather than requiring explicit open and close operations.

These TIDs must have all the characteristics of unique identifiers. They must be difficult to forge and not reused. The implementation described below, however, requires the opposite properties of TIDs. They must be compact, since they will be stored in each cylinder map entry, and they must be generated in a predictable sequence. Therefore, we will adopt an interface which makes the transaction identity implicit rather than the interlock identities.

The open request is extended to contain a TUID, and a list of PUIDs. The PUIDs are all to be locked for a single transaction, which is specified implicitly by the TUID. If this TUID is zero, then a new transaction is to be started, with a new transaction identifier. If a valid TUID is supplied, however, then the request is to open these objects for the same transaction as the TUID supplied. In either case, a list of TUIDs, one for each PUID, is returned.

Using this request, an interactive transaction could be constructed which did not know the set of objects to be read and written. As each object was encountered, an open would be requested which contained the TUID of the last object opened, the first open containing zero in this position. This indirect specification would allow the file server to associate the same TID with all objects without exposing it to the client.

TUIDs would be used in the normal way to alter files and indices. A transaction would be committed by the close operation, whose form would be unchanged. The meaning of a close operation would be "commit the transaction to which this interlock belongs" and would have two effects. The transaction would be committed atomically, and all The TUIDs established under it would be destroyed. Thus a transaction might be performed as follows:

```
(tuid1, tuid2, tuid3) := open (0, puid1, puid2, puid3)
puid4 := retrieve (tuid1, 0)
tuid4 := open (tuid3, puid4)
  :              :
close (tuid1, TRUE)
```

6.2 Implementation of Multiple-Object Transactions

To perform multiple-object transactions, it is essential that one bit control whether the transaction commits or aborts. Thus, per-object commit bits can no longer be stored, and a more flexible way

must be devised for associating one commit bit with a transaction on some arbitrary collection of objects. A particularly simple way of doing this is to associate every change made by the transaction with the transaction identifier. The only requirement is that these identifiers be unique so that modifications due to different transactions are never confused.

The current method of recording changes made by a transaction in cylinder map entries can be retained. The only modification that need be made is to include a transaction number in each cylinder map entry as it is modified. The two state bits which select one of the four allocation states must be replaced by one state bit and a transaction number. The state of each block is thus "allocated by transaction N" or "deallocated by transaction M". Figure 6.1 shows the modifications necessary to the cylinder map format.

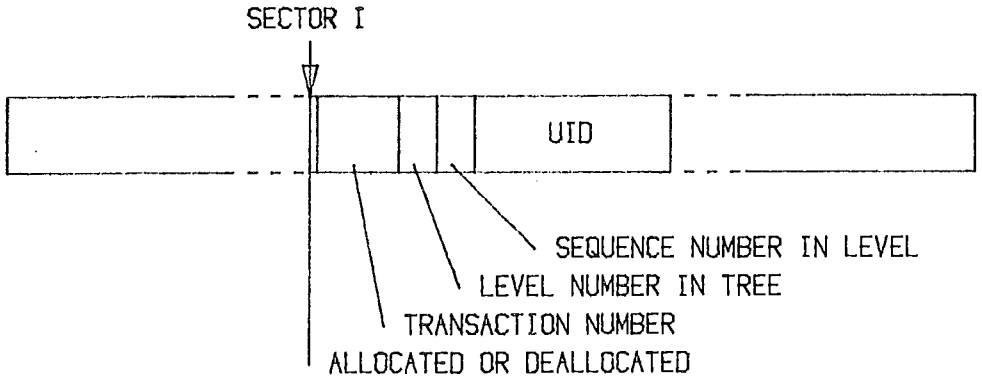

Fig 6.1 Modified Cylinder Map Format

The cylinder maps need not increase greatly in size under the new scheme. At 100 transactions per second, 30 bit transaction numbers would last about 4 months, and a method is outlined in section 6.8 for reusing these numbers once the supply is exhausted. Cylinder map entries might be laid out as follows:

allocation state	1 bit
deleting bit (root blocks only)	1 bit
transaction number	30 bits
level in object tree	2 bits
sequence number in level	14 bits*
UID	32 bits
	80 bits

On 80 Mbyte discs organised as in the current implementation, the

* As noted in section 4.3, it is only necessary to store half the UID in any cylinder map entry, rather than all 64 bits.

cylinder maps would represent a total storage cost of .5MB or 0.6%. To complete the mechanism, a central transaction table is needed which records the commit bit for each transaction. This gives the required indirection, since whether or not a change in block state is definite or tentative is defined by the central table. The allocation of a block by transaction N is conditional on its committing, and if the transaction fails, the block can revert to its deallocated state.

The transaction table can be managed simply by recording in it only the numbers of transactions which have not completed. A transaction number is recorded in the table before a transaction allocates or deallocates its first block. If the transaction commits, its number is simply removed from the table, thus committing all changes it has made. If the transaction fails, then all block state changes it has made must be reversed. Only then can the transaction's number be removed from the table. This strategy allows the table to be small since it need only contain the numbers of active transactions.

Each cylinder map entry is thus conditional on the presence or absence of its transaction number in the central table. The equivalent of Restart which removes "intentions" works as follows. For each cylinder map entry the central transaction table is searched for the map entry's transaction number. If it is not found, then this cylinder map entry was created by a transaction known to have committed. No action therefore need be taken. If the number is found in the transaction table then the state change must be undone. A block marked "allocated by transaction N" must be changed to "deallocated by transaction 0"; one marked "deallocated by transaction N" must be marked "allocated by transaction 0". Note that we are unable to reconstruct the actual transaction number which was previously in the cylinder map entry. This is of no importance since the previous transaction which changed this cylinder map entry is known to have committed. A conventional value of zero can be written in the cylinder map entry, on the understanding that no real transaction is ever given the number zero.

The performance advantage of the new scheme derives from the fact that cylinder map entries are changed directly to the values they should have if the transaction commits. Thus, committing a transaction simply involves ensuring that all modified cylinder maps are written to disc, and removing the transaction number from the transaction table. Aborting a transaction, however, will take the same amount of work as in the current implementation, since all changes made by the transaction to cylinder maps must be undone.

A certain amount of care is needed to ensure that the transaction table, which is now a critical data structure, is updated safely and efficiently. Clearly, there must be enough redundancy so that writing the transaction table to disc is always safe. One way to do this is

to maintain multiple copies of the table, and always arrange to write to an old copy rather than the most recent one. This idea has a particularly attractive implementation on disc units which allow sectors of different sizes on a given track. In this case, a small end-of-track sector can be created in a fixed place on each cylinder of each disc out of what would otherwise be wasted space. There would then be two distinguished blocks per cylinder: one cylinder map block and the transaction table block. When a transaction commits, the updated transaction table can be written wherever the disc heads happen to be at the time, unless this is the cylinder on which the last copy was written. For discs without variable block size formatting, a fixed number (at least two) of transaction table blocks could be allocated at intervals over the disc surface. This strategy is both efficient and safe, if the transaction table is always written with a sequence number so that the most recent copy can be detected. When restarting after an interruption for instance, each disc must be scanned to find the most recent copy of the table before the cylinder maps are examined.

6.3 Transactionman

The control of the transaction table must be entrusted to a new module, Transactionman. Since many instances of Root must synchronise on the creation of new transaction table entries, Transactionman must run as a single instance in its own task. It will control three pieces of information; the current contents of the transaction table, the disc address on which it was last written, and the next free transaction number. Figure 6.2 shows the contents of the transaction table.

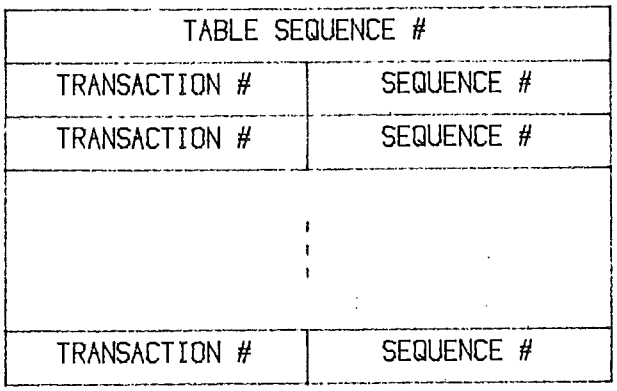

Fig 6.2 The Transaction Table

The transaction table will need to contain a sequence number which is incremented before each disc write so that Restart can find the most recent copy. Transactionman.create adds a new entry in an empty slot of the table, and writes in the new table entry the current table sequence number. The new transaction number is then returned to the caller.

Transactionman.flush is called by Cmapman as soon as the transaction attempts to allocate or deallocate its first block. Transactionman compares the current table sequence number with that recorded in the entry of the transaction. If they are equal, the table sequence number is incremented and Discman.writehere is called to write the transaction table to disc. If they are unequal, no action is taken since the transaction is already recorded on the most recent disc copy. An enquiry entry point, Transactionman.committed, is also needed for Cmapman's use. Given a transaction number, it returns a Boolean value indicating whether or not the number is present in the transaction table, which can be used by Cmapman to detect whether a transaction has really freed an apparently deallocated block.

Transactionman.delete is called to remove a transaction from the table. If its entry sequence number is the same as the current table sequence number, the entry is simply erased, since the number was never written to disc. Otherwise, the table sequence number is incremented, and the table is written to disc after erasure of the entry.

The marking of transaction table entries with the current table sequence number means that a short transaction which changes no block states will be added to and removed from the transaction table with no cost in disc transfers. This allows read and read-write transactions to be handled in a uniform manner at no extra cost.

6.4 Cmapman

Cmapman will be little changed under the new scheme. Cylinder map entries will change from recording four allocation states to two states and one number. Cmapman.intention will not be needed to set and unset the commit bit, however, since the equivalent function is assumed by Transactionman.

Internally, difficulties arise because it is no longer clear to Cmapman which blocks are definitively deallocated. In Cmapman.allocate, therefore, Transactionman.committed must be called to decide whether it is safe to reuse a deallocated block. In practice, a substantial saving can be made if Transactionman

guarantees to create transaction numbers in increasing numerical order, and maintains a system-wide variable containing the lowest uncommitted transaction number. During a first rapid pass over the cylinder map, Cmapman can use a very quick test for free blocks by comparing each entry's transaction number against the lowest uncommitted transaction number. Only if this scan produces no free blocks should a second pass be made which calls Transactionman.committed for each apparently deallocated block. This will find any blocks which have been deallocated recently.

Cmapman is never notified when a block which it has marked deallocated in fact becomes so. Because of this, all responsibility for keeping Blockman's allocation tables up to date must be shifted to Intman.

6.5 Intman

Given the central role of Intman in the current mechanism, it might be expected that substantial changes might be required for the new scheme. In fact, Intman's functions remain nearly unchanged.

Because transactions can abort, it is still necessary to maintain a list of block state changes made by the transaction so that these can be reversed if need be. As previously, recording individual disc addresses could allow a very large transaction to exhaust the available space, so cylinder numbers only are recorded, and the appropriate cylinder map can be scanned by Cmapman.searchcyl to find the changes made by a given transaction.

To allow Intman to keep Blockman informed of allocation changes on cylinders, Intman must maintain an allocation count A and a deallocation count D with each entry in the intentions vector as shown in figure 6.3.

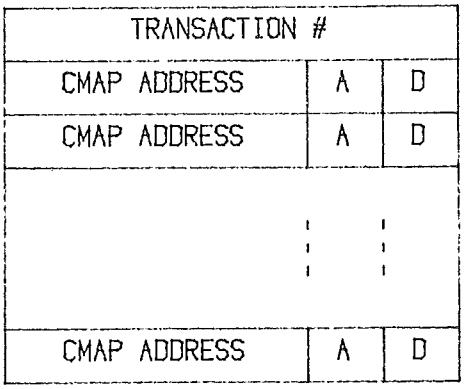

Fig 6.3 Modified Intentions Vector

Intman.record is now split into two entry points, Intman.allocate and Intman.deallocate which are called only by Cmapman.allocate and Cmapman.deallocate respectively. Intman.allocate searches the intentions vector for a matching cylinder number to that of the disc address supplied, and if one is found, increments its A count by one. If a match is not found, a new entry is made in the intentions vector with A=1 and D=0. In either case, Blockman.change is called to inform Blockman that there is one fewer free block on the cylinder. Intman.deallocate is very similar; the intentions vector is scanned, and a matched entry has its D count incremented by one. If no match is found, an entry is created with A=0 and D=1. Intman.deallocate does not call Blockman.change, however, since the block is not yet free.

Intman.ensure becomes trivial in the new scheme. If the transaction is committing, Storeman.writeda is called to ensure that each cylinder map in the intentions vector is up to date on disc. Transactionman.delete is then called to remove the transaction number from the transaction table. If the transaction is aborting, then no action need be taken.

Intman.cleanup also becomes simpler. If the transaction committed, then for each touched cylinder, Blockman will have been informed of the A allocations, but not of the D deallocations. This should now be done. If the transaction aborted, then as in the current mechanism, the object tree must be restored, and the cylinder map entries changed back to their earlier states. Finally, Blockman.change should be called indicating A deallocations on each cylinder involved, and Transactionman.delete should be called to erase the transaction number.

6.6 Blockman

The interface to Blockman must be expanded slightly. In addition to Blockman.setcyl which is already used by Restart to describe the initial allocation state of each cylinder, a new entry Blockman.change must be provided so that Intman can indicate relative changes to a cylinder. This new entry point precludes the use of a bit map to describe the allocation states of the cylinders of a disc, since Blockman must now be able to determine if the allocation of a number of blocks on a cylinder has left any free. Each bit, therefore, must be replaced by a count of free blocks.

6.7 Discman

To write the transaction table to disc, a new Discman function writehere is required. As for Discman.write, a bfw for the buffer to be written and a disc address are supplied by Transactionman, but the disc address is that at which the transaction table was last written. Discman constructs a new address by choosing some pack and the cylinder number which is currently under the read-write heads. It then writes the buffer to this address if it is not the same as the one supplied, otherwise to another address found by changing the cylinder number or using the current position on another pack. Finally, the input disc address is overwritten by the address actually used.

6.8 Restart

Restarting becomes more expensive in the new scheme, and consists of up to three passes over the discs instead of the one pass required under the current scheme.

Pass one reads all copies of the transaction table to find the most recent one, and the disc address at which it was written. This is most conveniently performed by Transactionman as part of its startup sequence, and at the end of it the last known value of the transaction table will be available in the normal way.

The second pass is as for the current scheme. Restart must check all cylinder map entries to detect changes made by failed transactions. These are recorded by calls of Intman.record in the

normal way, and at the end of the scan Restart calls Intman.cleanup with commit=FALSE to abort the failed transactions. At the end of this pass, Restart will be able to inform Transactionman of the next free transaction number. This is one greater than the largest transaction number found in any cylinder map entry.

A third pass may be required to replenish the supply of transaction numbers and transaction table sequence numbers, since these must be unique. This pass might be triggered when Transactionman detects that either sequence has exhausted half its range; in normal operation if Transactionman runs out of numbers, it can simply halt the file server. Note that at the start of this third pass, no transactions are incomplete, since these will have been dealt with in pass two. It is thus safe to update cylinder maps, since these are consistent with the object trees. The third pass, therefore, need only rewrite all cylinder maps setting all transaction numbers to zero. When this has been done, all copies of the transaction table must be overwritten by one with a sequence number of zero. The next free transaction number can now be set to 1. This third pass will only rarely be necessary depending on how many bits are used for transaction numbers and table sequence numbers, and the frequency of occurrence of transactions.

6.9 A Comparison of Costs

As well as being more general than the current mechanism, the modified transaction mechanism presented in this chapter will also be more economical in disc transfers. Consider an atomic transaction on a special object which alters a number of blocks. These blocks will be distributed over n cylinders, and another n' cylinders will be needed to find replacement blocks for them. These disc addresses will be held in m object maps of the object tree. As shown in section 4.5, a total of n+n' extra disc reads and 2(n+n'+1)+m extra disc writes are needed, of which 2n+2n'+m+1 transfers are synchronous, and n+n'+1 are asynchronous.

In the new scheme, the n+n' cylinder maps will still have to be read and written to change the states of the leaf blocks and their replacements. Thus n+n' reads and n+n' writes will be needed. Before any of these writes take place, however, the transaction table must be written to disc to mark the transaction incomplete, adding one to the number of writes. Then the m object maps must be written, and finally the transaction table must be written again to mark the transaction complete. Thus n+n' reads and n+n'+m+2 synchronous writes are needed, a total of 2n+2n'+m+2 synchronous transfers. Thus, one more synchronous transfer is needed that in the previous scheme, but the

n+n'+1 asynchronous transfers to clean up intentions are eliminated. Real time delay to the client is unlikely to increase since two of the transfers are of the "writehere" variety, and the elimination of the asynchronous cleaning up is likely to lead to a significant increase in overall efficiency.

6.10 Extension To Many Servers

The partitioning of the set of files and indices among a number of cooperating file servers may be done with only a few additional changes. The advantages of distributing the file service in this way are pointed out in Sturgis80; the amount of storage and the number of processors used can be adjusted to suit the requirements of a particular local network. There are two additional problems to be solved, however; how does a client direct requests to the server which controls a particular object, and how can the atomic transaction mechanism be extended to allow objects on a number of cooperating servers?

To allow clients to find the right server for a particular object, it is probably best if the address part of a UID is not extended to include a server number. This would restrict each disc pack to be under the control of a particular server, and would prevent a disc pack from being mounted on any server which had a free drive. Instead, it is better to maintain the current scheme in which UIDs contain disc pack numbers, and arrange that servers keep each other informed of the location of every mounted pack. A client could direct its first operation on a UID to any available server, which would use its knowledge of the packs currently available on other servers to forward the request. If the client noted the address of the server from which it eventually received a reply, it could use this as a hint for all further requests.

Extending atomic transactions to objects on many servers can be done easily following the pattern of the Juniper file server [Sturgis80]. The algorithm used is the two-phase commit protocol defined in section 3.2.1 with minor elaborations to cope with communication errors. It is derived from the two-phase commit protocol used in the Juniper file server, which is to be formally described in a forthcoming paper by B.W. Lampson and H.E. Sturgis. The basic change necessary is the partition of the range of transaction numbers between the different file servers. A transaction number would then be a pair (transaction number in server, server number).

A client would start a transaction by sending an open request with a list of PUIDs and a TUID of zero to any of the available file servers. This server would nominate itself as the coordinator of the transaction by allocating one of its own transaction numbers and writing this in the local transaction table. The coordinator would then scan the list of UIDs. Any objects which were in the coordinator server would be opened immediately for the transaction. The remainder would be divided into lists, one for each different server, and an open request giving the list of UIDs would be sent to each. This would of necessity be a different open operation from that used by clients, since the coordinator must include the transaction number which it has allocated in each list. The slave servers would set interlocks on the objects using the coordinator's transaction number rather than an internally generated one, and would arrange as usual that all interlocks were associated with a single intentions vector for the transaction. Thus, each server in the transaction would maintain a list of the local changes made by the transaction in the local intentions vector. The corresponding TUIDs would then be returned to the coordinator and thence to the client. Subsequent open requests would be directed to a server holding one of the objects in question, and would contain the TUID of an object already locked by this transaction. A server receiving such a request would obtain the correct transaction number from the server which created the TUID, using a privileged "transaction number of TUID" request, and would enter this number in its local transaction table*. Before replying to the client, it would inform the coordinator of its participation in the transaction if this were not the server which just supplied the transaction number.

When a close request arrived at a server, the transaction number of the TUID concerned would be looked up in the local copy of the intentions vector. If this showed the server to be a slave for the transaction, the server would simply generate an internal close request containing the transaction number and would send it to the coordinator. If it were the coordinator, however, then it would send an internal "get ready" request to all slaves for the purpose of

* Since transaction numbers from different servers may now appear in a transaction table, it becomes more difficult to maintain the "lowest uncommitted transaction number" useful to Cmapman. Lamport's technique for keeping distributed clocks in step can be used here [Lamport78]; whenever a transaction number form a different server arrives, the local next free transaction number would be increased to at least this value. Each server would also periodically broadcast its current value to ensure that the servers always stayed approximately in step.

synchronisation. On receipt of a get ready request, each slave would call an Intman entry point to flush all cylinder maps modified by the transaction, so that all changes made by the transaction were recorded on disc. Note that each slave would still have the transaction present in its local transaction table, so the transaction has not yet been committed by any of them.

When the coordinator received successful replies to all get ready requests, it would commit the transaction. Any failure of a get ready request must abort the transaction as a whole, since a slave may have been interrupted before all changes were recorded on disc. The coordinator would commit the transaction in the normal way, by removing its number from the local transaction table, and would send an internal close request to each slave directing it to do the same. If the transaction failed, the coordinator would send a close request to each slave with commit = FALSE, and would wait for a completion acknowledgement from each before deleting the transaction number from its local table.

This sequence must have the cooperation of the restart algorithm in all servers. When restarting, each server must check cylinder map entry transaction numbers against transaction table entries in the normal way; if no transaction table entry is found, the transaction is deemed to have committed. If a match is found, and the transaction number defines this server to be a slave for the transaction, then the server can take no unilateral action. It must enquire of the coordinator whether or not the transaction committed. On receipt of such a request, the coordinator can find the transaction's state by a call of Transactionman.committed, and the coordinator's reply can allow the slave to take appropriate action*. If the slave gets no reply from the coordinator, however, then it must wait. On no account must it abort or commit its part of the transaction without direction from the coordinator. The simplest strategy for the slave to follow in this case would be to re-establish interlocks on all objects involved in the transaction under newly generated TUIDs. All clients would thus be prevented from accessing these objects until a periodic retry of the coordinator enquiry succeeded in determining the fate of the transaction.

If on restart the server finds itself to be the coordinator for the transaction, because one of its own transaction numbers is still recorded in its local table, then the strategy is somewhat simpler. Because the transaction number is still in the table, it is safe for the coordinator to abort the transaction, whether or not any slaves

* Note that if the coordinator receives one of these enquiries before receiving a reply to a get ready request from the same server for the same transaction, it must abort the transaction at this point.

have received get ready requests. It may therefore unilaterally undo all local changes made by the transaction. However, it may not remove the transaction number from its table until it is certain that all slaves have also undone their parts of the transaction. It must send close requests with commit = FALSE to all slave servers, or, if this information was not recorded on disc and has now been lost because of the interruption, to all file servers. Only when it has received an acknowledgement from each server contacted can it delete the transaction number from its transaction table. Note that the delay while the coordinator is informing all slaves of the abort of the transaction does not prevent normal service from being resumed. Those objects involved locally in the aborted transaction can be unlocked and other transactions can be allowed to proceed normally.

As seen by a slave, therefore, a transaction has three distinct phases. The first phase lasts from the first open request it receives until the reply to the get ready request is sent. During this time, the slave is free to abort the transaction unilaterally in one of three ways:

1) It can send an explicit close request to the coordinator on detection of some internal error while processing one of the transaction's operations
2) It can refuse to get ready.
3) It can make the coordinator aware that it has crashed and restarted by enquiring about the state of the transaction.

In any of these cases, the coordinator must direct all slaves to abort their parts of the transaction. The second phase lasts from the transmission of the get ready acknowledgement until the slave is informed of the fate of the transaction, either on reception of the expected close request from the coordinator, or by enquiry. By entering the second phase, the slave abdicates the right to take unilateral action, and agrees to wait indefinitely until informed of the transaction's fate. The third phase begins with the arrival of the coordinator's decision. If the transaction committed, then the transaction number is simply removed from the local transaction table. If it aborted, then all local state changes (which will have been rediscovered by Restart if there has been an intervening interruption) must be reversed in the normal way before the transaction number is deleted and the close request acknowledged.

Chapter 7

Conclusion

In the preceding chapters the motivations behind the interface, algorithms, and program structure of a file server for a local network have been presented. At the time of writing, the file server has been in service for over a year, and the design choices can now be reviewed in the light of experience.

The fundamental goal of providing a general, shared, centralised file service has been achieved. The Cambridge file server is in constant use by two operating systems - a modified TRIPOS [Richards79] and the CAP operating system [Dellar80b] - both of which use it for all secondary storage. At any given time, four or five instances of TRIPOS may be accessing the shared filing system held on the server, and the CAP may be using it to swap segments in and out of its memory. There are also several microprocessor-based systems attached to the ring, and a number of servers running in these machines protect themselves against interruptions by archiving their state periodically on the file server.

The performance of the file server seems acceptable as reflected in the response times of the client operating systems. Measurements have shown that when the file server is otherwise idle, reading 256 words at random from a large three-level file takes about 50 milliseconds, and writing takes about 65 milliseconds, including all software and communications delays in client and server. These times are comparable to those given for the WFS file server [Swinehart79]. In a description of the conversion of the CAP operating system to use the file server for all secondary storage, Dellar reports that under light loads the new system is faster because of a reduction in the amount of store-resident code [Dellar80b]. As the swapping rate increases, however, performance is somewhat worse than in the previous system. This may be attributed partially to the additional software delays involved in sequencing file transfers, and partially to the fact that the Cambridge ring has a much lower bandwidth than a direct memory access channel to a local disc. The very lightweight file transfer protocol reduces this effect to some degree, since it allows transfers to proceed at about 80% of the maximum point-to-point bandwidth of the ring. A more complicated protocol would have reduced the transfer rate proportionally.

Several strategies could be used to increase the file server's performance. First among these would be the use of a more sophisticated ring interface. In the current implementation, Ringtx

requires two interrupts to transfer a basic block, and Ringrx requires four. A more serious overhead than this high interrupt rate is the software calculation of the checksum of a basic block, which takes about a quarter of the block transmission or reception time. Thus, during a file transfer there is contention for the server's memory from the disc controller, from the ring interface, and from the processor computing checksums and handling frequent interrupts. Much of this processor activity could be removed by using a specialised ring interface to handle the basic block protocol. Such a microprocessor-controlled interface is now being tested in the Computer Laboratory [Gibbons81], and its use in the file server will push the file transfer rate to within a few percentage points of the maximum point-to-point ring transfer rate. The use of this interface may reduce the processor load to such an extent that it becomes feasible to attempt several file transfers simultaneously by interleaving the transmission and reception of blocks of different transfers. Only transfers which used different disc drives could be scheduled concurrently, of course, or any potential benefits would be lost in the extra movements of the disc heads. Transfers overlapped in this way would individually proceed at a slower rate, but might allow a higher utilisation of the centralised file server hardware.

A second area where performance improvements are possible is the allocation of blocks to files. In the current implementation, writing to a file for the first time is slowed by the need to allocate blocks on the fly. This is due to the software delays involved in choosing a cylinder map and then searching the block cache each time a block is required, and also to occasional disc delays as a cylinder map becomes full and the next candidate is not in the cache. To a large extent, the software delays will be masked when the microprocessor-controlled ring interface reduces the load on the processor. These can be further reduced by a strategy of allocating several blocks in a single call of Cmapman, rather than just one block. Beyond this, the possible improvements seem to lie in the area of improving the physical contiguity of files, either by preallocating blocks to files before any attempt is made to write to them, or by allocating contiguous groups of blocks rather than single blocks. These policies would speed up writing by allowing some of the allocation work to be done ahead of time, and by improving the serial access characteristics of files. Trade-offs of space for time such as these are the perennial concern of the filing system designer, but in the current circumstances where the average object size is about 1500 bytes, both policies would seem to be wasteful of space. The file organisation described in this thesis of variable-depth trees to which blocks are allocated on demand is biased towards efficiency in space rather than access time, and will tend to lead to scattering as the disc becomes

full.

The effects of the reliability mechanisms built into the file server are rather harder to judge. Certainly, the requirement to be able to recover from an interruption which destroys a disc block has resulted in unavoidable extra disc transfers. This cost is involved whenever structural information is changed, and is most noticeable when writing to to a new file or creating an object. Object creation, requiring two disc transfers, is in fact cheaper than the subsequent index retention, which will require either three of five transfers, depending on whether the optimised commit sequence can be used. Dellar describes an ingenious way of overcoming this cost in the CAP operating system by maintaining a pool of new objects ready for use [Dellar80a]. The costs of creating new objects and of writing to a new file are essentially indications that allocating and deallocating blocks are relatively expensive operations. This expense is greater than it need be since the transaction mechanism does not take advantage of the fact that the vast majority of transactions are successful. The revised mechanism presented in chapter six is biased towards the frequent outcome, and is thus more efficient as well as being more general.

In practice, the transaction mechanism is not heavily used by the client operating systems. The CAP, for instance, never opens objects explicitly, but always uses PUIDs to swap complete copies of files in and out of its store. If a file write fails because of a communications failure or a file server crash, the CAP can simply restart it because all file server operations are repeatable. Under these circumstances, it is not particularly useful to have a file revert to its original state on recovery. The only circumstance in which the CAP uses the atomic transaction facilities of the file server is in directory updates; the data part of every directory is stored in a special file, so that a CAP interruption while a directory is being swapped out never leaves an inconsistent state. In practice, this has meant that the lengthy scan of the filing system which the CAP used to perform on each restart has been eliminated in the file server based system. The other main client, the TRIPOS operating system, uses the transaction mechanism both to maintain consistency of its directories over interruptions, and to synchronise the access of its different instances to the shared filing system. These patterns of use are largely an indication that operating systems usually write one object at a time, and that the consistency requirements between different objects are rather loose. Dellar points out, however, that a multiple-object transaction mechanism would have been useful in updating CAP directories since a directory is implemented as an index and a special file which must be kept in step [Dellar80b]. Other applications such as data bases, which cannot use the simple "read old

file create new file" pattern typical of operating systems, would require a general atomic transaction mechanism like that presented in chapter six.

The actual representation of the redundant information on disc has a number of attractive properties. Error recovery is normally very rapid, and is not affected by the complexity of the index graph structure or the number of objects in existence. Under normal circumstances, the file server will deal with interrupted transactions in about 30 seconds. The program which scans the graph structure in the event of a cylinder map corruption, however, requires about 30 minutes. The rapidity of this restart sequence was invaluable during program development, but is perhaps less essential when the program is in working order.

Another property of the representation is that it tends to maintain a high degree of locality. When a transaction is to be committed, the object and cylinder maps which must be written are close together on disc, and often all on the same cylinder. This organisation is advantageous given the access characteristics of moving-head discs, and tends to reduce the cost of performing atomic transactions.

The amount of redundancy added to the disc representation is actually rather small. In essence, the change has been made by replacing a block allocation bit map by a list of cylinder maps having 64 bits per entry. In terms of the total disc space available, these cylinder maps represent a negligible cost of 0.5%. Unfortunately, this minimal redundancy leads to a number of possibilities for "second-order" errors which do not occur with the marginally more expensive pair-redundant scheme of Sturgis et al., described in section 3.3. The mechanism relies on the fact that a cylinder map is only written when all object maps are known to be correct, and conversely object maps are only written when all cylinder maps are known to be correct. Thus, for example, transactions on different objects sharing blocks on a common cylinder must not commit at the same time. Otherwise, an interruption which destroys a cylinder map when object maps of another transaction have only been partially written will be unrecoverable. Similarly, if the root block of a normal object is lost when it is overwritten, the attributes of the object will be lost; neither the object's uninitialised data value nor its maximum size will be known after the object tree is rebuilt. These objections do not make the mechanism unworkable, but require more careful attention that the simpler pair-redundant strategy.

Much of this thesis has been concerned with strategies for overcoming the damage caused by failed disc writes. This is a rare form of interruption - it has occurred only twice in a year of operation - but has been the cause of all the redundancy introduced into the disc representation and of the subtleties in the transaction

mechanism. The need to maintain the redundancy of every block of structural information at an acceptable cost has introduced complexity into the algorithms, and therefore potential sources of error. The conclusion which might be drawn from this work is that discs which can lose information during a transfer with appreciable probability are unsuitable for building reliable storage. An interesting research problem would be to investigate the types of hardware assistance which might simplify the design of reliable file servers.

The failure of a disc transfer is not in itself a disastrous event. As in the case of file server operations, disc writes are repeatable, and successive attempts may eventually write correct data. Disaster occurs when not only the contents of the disc block but also the store copy from which it was being written are lost. Under these circumstances it is necessary to reconstruct the block contents using only other disc blocks; this leads to the level of disc redundancy used in the file server, and to a doubling in the number of disc transfers. Between two disc writes to the same block of structural information, another disc block must be written to maintain redundancy.

This situation can be improved by arranging not to lose the store copy of a disc block over an interruption. This could be done by making part of the processor's address space non-volatile, perhaps by using battery-powered semiconductor memory which would not lose its contents in a power failure. Keeping all disc buffers in it - and assuming that no interruption altered the contents of non-volatile store - would eliminate the need for redundant recording of information on disc. The stable storage abstraction of section 3.3 could be implemented much more efficiently by arranging to rewrite all disc buffers on restarting after an interruption. A little care is required to avoid writing partially modified disc blocks in this way, or writing some, but not all of a set of blocks. Perhaps the simplest way to arrange this is by storing a table of "ready-to-write" bits in the non-volatile memory, and by having each buffer contain a pointer to one of the table entries. A set of buffered disc blocks could then be marked for writing by changing the single ready-to-write bit to which they all referred. On restarting, each buffer would be scanned, and written to disc again if it was marked ready to write.

This simple scheme has two major advantages. Because it eliminates the need for redundant disc information, the number of disc transfers would be greatly reduced. In a file server implemented in this way, it would halve the number of disc writes of structural information. Secondly, a disc block can be considered written by the processor as soon as its ready-to-write bit is turned on, since it will eventually be transferred to disc whether or not interruptions occur. The non-volatile store can thus serve as a buffer in the true sense of the

word by largely decoupling the activity of the processor and the disc. There is also the possibility of optimal disc scheduling, which does not arise if disc writes must be ordered carefully to ensure redundancy.

As just described, the non-volatile memory attached to a processor could function as a cache for large volumes of information on disc. It could also be used as the only storage medium for small amounts of essential information. The transaction table of chapter six is an obvious candidate. Instead of storing a copy on each disc cylinder, two copies could be kept permanently in non-volatile store. The processor would update the table by reading one copy and writing the other, so that in the event of an interruption, there would always be a consistent version of the table to use in recovery.

As local networks become more widespread, file servers such as the one described in this thesis will continue to be built. At least in the short term, it will be most economical to share the use of expensive mechanical storage devices among inexpensive processors. The arrival of cheap electronic or magnetic bulk memories is likely to change this situation, by encouraging distribution of the long-term storage of objects. However, the use of increasing numbers of processors in distributed computations will tend to lead to complex failure modes as individual machines crash or messages are lost. The problems investigated here of protecting a file server and its clients from the unpredictable effects of concurrency and interruptions will continue to be one of the main obstacles to the construction of distributed systems.

References

[Avizienis78] Avizienis A.
"Fault-tolerance: The Survival Attribute of Distributed Systems".
Proc. IEEE 66, pp 1109-1125, October 1978.

[Bernstein80] Bernstein P.A., Shipman D.W., Rothnie J.B.
"Concurrency Control in a System for Distributed Databases (SDD-1)".
ACM Transactions on Database Systems 5 1, pp 18-51, March 1980.

[Birrell78] Birrell A.D., Needham R.M.
"An Asynchronous Garbage Collector for the CAP Filing System".
Operating Systems Review 12 2, pp 31-33, April 1978.

[Birrell80] Birrell A.D., Needham R.M.
"A Universal File Server".
IEEE Transactions on Software Engineering SE - 6 5, pp 450-453.

[Davies79] Davies C.T.
"Data Processing Integrity".
Computing Systems Reliability ed. T. Anderson & B. Randell, Cambridge University Press 1979.

[Dellar80a] Dellar C.N.R.
"The Distribution of Operating System Functions".
Ph.D Dissertation, Cambridge University, September 1980.

[Dellar80b] Dellar C.N.R.
"Removing Backing Store Administration from the CAP Operating System".
Operating Systems Review 14 4, pp 41-49, October 1980.

[Dennis66] Dennis J.B., van Horne E.C.
"Programming Semantics for Multi-programmed Computers".
CACM 9 3, pp 143-155.

[Deutsch76] Deutsch L.P., Bobrow D.G.
"An Efficient, Incremental Automatic Garbage Collector".
CACM 19 9, pp 522-526, September 1976.

[Dijkstra78] Dijkstra E.W., Lamport L., Martin A.J., Scholten C.S., Steffens E.F.M.
"On-the-Fly Garbage Collection: An Exercise in Cooperation".
CACM 21 11, pp 966-974, November 1978.

[Dion80] Dion J.
"The Cambridge File Server".
Operating Systems Review 14 4, pp 26-35, October 1980.

[England72] England D.
"Architectural Features of System 250".
Infotech State of the Art Report 14, 1972.

[Eswaran76] Eswaran K.P., Gray J.N., Lorie R.A., Traiger I.L.
"The Notions of Consistency and Predicate Locks in a Database System".
CACM 19 11, pp 624-633, November 1976.

[Fraser69] Fraser A.G.
"Integrity of a Mass Storage Filing System".
Computer Journal 12 p 1, 1969.

[Garnett80] Garnett N.H.
"An Asynchronous Garbage Collector for the Cambridge File Server".
Operating Systems Review 14 4, pp 36-40, October 1980.

[Gibbons81] Gibbons J.J.
"Interfaces to the Cambridge Ring".
Ph.D. dissertation, Cambridge University, 1981.

[Gifford79] Gifford D.
"Weighted Voting for Replicated Data".
Proc 7th Symposium on Operating System Principles, pp. 150-162, December 1979.

[Giordano76] Giordano N.J., Schwartz M.S.
"Data Base Recovery at CMIC".
Proc ACM SIGMOD Conference, 1976.

[Gray78] Gray J.N.
"Notes on Data Base Operating Systems".
Advanced Course on Operating Systems, Lecture Notes in Computer Science 60, pp 393-481, Springer-Verlag 1978.

[Hopper78] Hopper A.
"Local Area Computer Communications Networks".
Ph.D dissertation, Cambridge University, April 1978.

[Janson76] Janson P.A.
"Using Type Extension to Organize Virtual Memory Mechanisms".
Ph.D. dissertation, MIT technical report MIT/LCS/TR-167, 1976.

[Jones78] Jones A.K.
"The Object Model: A Tool for Structuring Software".
Advanced Course on Operating Systems. Lecture Notes in Computer Science 60, pp 7-16, Springer-Verlag, 1978.

[Lamport78] Lamport L.
"Time, Clocks, and the Ordering of Events in a Distributed System".
CACM 21 7, pp 558-565, July 1978.

[Lampson74a] Lampson B.W.
"Protection".
Operating Systems Review 8 1, pp 18-24 January 1974.

[Lampson74b] Lampson B.W.
"Redundancy and Robustness in Memory Protection".
Information Processing '74, IFIP Congress 1974.

[Lampson79] Lampson B.W., Sproull R.F.
"An Open Operating System for a Single-User Machine".
Proc. 7th Symposium on Operating System Principles, pp 98-105, December 1979.

[Lauer78] Lauer H.C., Needham R.M.
"On the Duality of Operating System Structures".
Proc Second International Symposium on Operating Systems, IRIA October 1978. Reprinted in Operating Systems Review 13 2, pp 13-19, April 1979.

[Lindsay77] Lindsay C.H.
"Informal Introduction to Algol68".
North-Holland, Amsterdam 1977.

[Liskov74] Liskov B., Zilles S.
"Programming With Abstract Data Types".
Proc. Symposium on Very High Level Languages. SIGPLAN Notices 9 4, pp 50-59. April 1974.

[Lorie77] Lorie R.L.
"Physical Integrity in a Large Segmented Database".
ACM Transactions on Database Systems 2 1, 1977.

[Melliar-Smith75] Melliar-Smith P.
"A project to investigate data-base reliability".
Technical report, Computing Laboratory, University of Newcastle-Upon-Tyne, 1975.

[Menasce80] Menasce D.A., Popek G.J., Muntz R.R.
"A Locking Protocol for Resource Coordination in Distributed Databases".
ACM Transactions on Database Systems 5 , 2, pp 103-138, June 1980.

[Metcalfe75] Metcalfe R.M., Boggs D.R.
"Ethernet: Distributed Packet Switching for Local Computer Networks".
Xerox Palo Alto Research Center technical report CSL-75-7, November 1975.

[Mitchell79] Mitchell J.G., Maybury W., Sweet R.
"Mesa Language Manual",
version 5.0 Xerox Palo Alto Research Center technical report CSL-79-3, April 1979.

[Needham77] Needham R.M., Birrell A.D.
"The CAP Filing System".
Proc. 6th Symposium on Operating System Principles, Operating Systems Review 11 5, pp 11-16, 1977.

[Needham78] Needham R.M., Schroeder M.D.
"Using Encryption for Authentication in Large Networks of Computers".
Xerox Palo Alto Research Center technical report CSL-78-4, September 1978.

[Organick72] Organick E.I.
"The Multics System: an examination of its structure".
MIT Press 1972.

[Ousterhout80] Ousterhout J.K., Scelza D.A., Sindhu P.S.
"Medusa: An Experiment in Distributed Operating System Structure".
CACM 23 2, pp 92-104, February 1980.

[Parnas72] Parnas D.L.
"A Technique for Software Module Specification with Examples".
CACM 15 5, pp 330-336, May 1972.

[Paxton79] Paxton W.H.
"A Client-Based Transaction System to Maintain Data Integrity".
Proc. 7th Symposium on Operating System Principles, December 1979.

[Randell78a] Randell B.
"Reliable Computing Systems".
Advanced Course on Operating Systems, Lecture Notes in Computer Science 60, pp 282-392, Springer-Verlag 1978.

[Randell78b] Randell B., Lee P.A., Treleaven P.C.
"Reliability Issues in Computing System Design".
Computing Surveys 10 2, pp 123-166, June 1978.

[Randell79] Randell B.
"System Reliability and Structuring".
Computing Systems Reliability ed. T. Anderson & B. Randell, Cambridge University Press 1979.

[Rappaport75] Rappaport R.L.
"File Structure Design to Facilitate On-Line Instantaneous Updating".
Proc ACM SIGMOD Conference 1979, pp 1-14.

[Redell80] Redell D.D., Dalal Y.K., Horsley T.R., Lauer H.C., McJones P.R., Murray H.G., Purcell S.C.
"Pilot: An Operating System for a Personal Computer".
CACM 23 2, February 1980.

[Reed78] Reed D.P.
"Naming and Synchronization in a Decentralized Computer System".
Ph.D. dissertation, MIT technical report MIT/LCS/TR-205.

[Reed79] Reed D.P.
"Implementing Atomic Actions on Decentralised Data".
Preprints for Proc. 7th Symposium on Operating System Principles, pp 66-74, December 1979.

[Richards69] Richards M.
"BCPL: A Tool for Compiler Writing and System Programming".
AFIPS SJCC Proceedings 35, pp 557-566, 1969.

[Richards79] Richards M., Aylward A.R., Bond P., Evans R.D., Knight B.J.
"TRIPOS - A Portable Operating System for Mini-Computers".
Software Practice and Experience 9 7, pp 513-520 July 1979.

[Shoch80] Shoch J.F., Hupp J.A.
"Measured Performance of an Ethernet Local Network".
CACM 23 12, pp 711-720, December 1980.

[Stern74] Stern J.A.
"Backup and Recovery of On-Line Information in a Computer Utility".
M.I.T technical report MAC TR-116 January 1974.

[Stroustrup79] Stroustrup B.
"Communication and Control in Distributed Computer Systems".
Ph.D. dissertation, Cambridge University, 1979.

[Sturgis74] Sturgis H.E.
"A Postmortem for a Time Sharing System".
Xerox Palo Alto Research Centre report CSL 74-1, January 1974.

[Sturgis80] Sturgis H., Mitchell J.G., Israel J.
"Issues in the Design and Use of a Distributed File System".
ACM Operating Systems Review 14 3, pp 55-69, July 1980.

[Swinehart79] Swinehart D., McDaniel G., Boggs D.
"WFS: A Simple Shared File System for a Distributed Environment".
Proc. 7th Symposium on Operating System Principles, December 1979.

[Thomas79] Thomas R.H.
"A Majority Concensus Approach to Concurrency Control".
ACM Transactions on Database Systems 4 2, pp 180-209, June 1979.

[Verhofstad78] Verhofstad J.S.M.
"Recovery Techniques for Database Systems".
Computing Surveys 10 2, pp 167-196 1978.

[Wilkes75] Wilkes M.V.
"Time Sharing Computer Systems".
3rd edition, American Elsevier, 1975.

[Wilkes79a] Wilkes M.V., Wheeler D.J.
"The Cambridge Digital Communications Ring".
Proc. Local Area Communications Network Symposium, Boston, May 1979. National Bureau of Standards Special Publication.

[Wilkes79b] Wilkes M.V., Needham R.M.
"The Cambridge CAP Computer and its Operating System".
Operating and Programming System Series, Elsevier North Holland, 1979.

[Wilkes80] Wilkes M.V., Needham R.M.
"The Cambridge Model Distributed System".
Operating Systems Review 14 4, pp 21-29, January 1980.

[Wulf74] Wulf W. et. al.

"HYDRA: The Kernel of a Multiprocessor Operating System".
CACM 17 6, pp 337-345, 1974.