A Capability-Based Persistent Distributed Shared Memory

A thesis submitted to the

University of Newcastle, NSW

for the degree of

Doctor of Philosophy in Computer Science

by

Frans Alexander Henskens

MAY, 1991

Candidate's Certificate

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

(Frans Alexander Henskens)

Acknowledgements

I wish to offer my sincere thanks to Associate Professor John Rosenberg, my friend, colleague, and supervisor during this research project. Not only have I received the best possible supervision from Professor Rosenberg, but I have benefited enormously from his insight, his wide ranging knowledge, and his constant encouragement.

Thanks are also due to other friends and colleagues: Professor J. Leslie Keedy for founding the MONADS project, for his belief in me, and for his encouragement and expertise; Mr David Koch for the numerous discussions and for his hardware expertise; Mr Peter Brössler for the late night discussions while I was struggling with network protocols and the MONADS architecture; Dr Michael Hannaford for his encouragement and willing ear when my spirits were low.

Thank you to the Australian Postgraduate Research Awards Scheme and the University of Newcastle for providing me with the opportunity to perform this research whilst simultaneously providing me with the means to support my family.

Lastly I must thank my wife, Hélène, and my sons Frans, Samuel, and Willem, for being my family, and providing me with the motivation to complete this work.

TABLE OF CONTENTS

SYNOPSIS

This thesis presents research into the application of the Distributed Shared Memory paradigm to distribution of an existing persistent store based on a large uniform virtual memory. Control of access to this store is provided by support at the architectural level for capability-based addressing.

Distribution of the store increases the number of potentially faulty system components, making the store more susceptible to failure. It also extends the potential domain of effect of the failure of a single machine to any subset of the set of machines connected to the network. For this reason the related topic of stability of the store is also investigated.

The distributed system described in this thesis hides such distribution from the user, providing a stable and secure persistent environment. Features of the distributed store include naming and location transparency, system-wide data coherency, and fine-grained control over access to data and programs. Moreover, programs written for non-networked use execute without change in the distributed environment.

The thesis is organised as nine chapters.

Chapter 1 introduces the reader to the thesis, providing an overview of the MONADS project on which the work is based, and presenting the rationale for and the aims of the research.

Chapter 2 provides the reader with background knowledge of networking in general, and presents several approaches to the problem of providing distributed access to resources. The concept of Distributed Shared Memory (DSM), which is the approach to distribution used in this work, is then introduced.

Chapter 3 introduces the concept of persistence. The MONADS architecture, as mentioned earlier in this chapter, had the potential to satisfy all the requirements of a persistent store. In chapter 3 the persistence paradigm is presented, and existing single-user and distributed persistent systems are described.

Chapter 4 describes the MONADS architecture. Understanding of this architecture is necessary because the subsequent chapters present modifications and extensions to it.

Chapter 5 presents extensions to the single-node MONADS architecture which implement a DSM approach to distribution. Access to resources is achieved in a manner which makes their physical location totally transparent to the user. The architecture presented in this chapter assumes that programs and data are never moved from the disk on which they were created, and that a disk is never moved from the node on which it was formatted and partitioned.

Chapter 6 addresses the restricted nature of the model presented in chapter 5. In chapter 6, schemes which allow disks to be mounted on any active machine, and which allow data and programs to migrate between disks and machines are presented. These schemes are significant because they allow such movement of resources without compromising the location transparency achieved in chapter 5.

Chapter 7 addresses the issue of stability. The chapter starts with some background discussion, and then presents a scheme for achieving stability of the single-node MONADS store. Simply adding this stability scheme to the MONADS architecture, together with LEIBNIZ, would provide a system satisfying the requirements of a persistent programming environment. An extended scheme which provides a stable distributed system is then presented.

Chapter 8 describes an implementation of the distributed architecture using an experimental system consisting of three MONADS-PC computers.

Chapter 9 concludes the thesis by summarising the achievements of this research and indicating directions for future research.

Chapter 1 INTRODUCTION

1.0 Background to this Research

The research presented in this thesis was performed as part of the MONADS project. This project was instigated by then Dr. J. L. Keedy at Monash University in 1976. Work on the project is currently in progress at two locations; at Newcastle, Australia under the leadership of Associate Professor J. Rosenberg, and at Bremen, Germany under the leadership of Professor J. L. Keedy.

The main aim of the project is the development of improved software engineering techniques. The need for such new techniques became apparent to Dr. Keedy during his employment as a member of the VME operating system design team with ICL. Two major deficiencies of existing software systems were targeted. These were

(1) the high cost of software development and maintenance, and

(2) weaknesses in the area of security of data.

The MONADS group developed techniques based on the decomposition of software systems into collections of *information-hiding modules* as first proposed by Parnas [87, 88]. Access to these modules, and thus to all programs and data in the system, is controlled by *capabilities* [39, 47]. After a moderately successful attempt to implement these techniques using modified commercially available hardware [51, 122], the group developed a series of purpose-built designs [61, 65] culminating in the MONADS-PC architecture [100]. This architecture provides explicit support for both modules and capability-based control of access to them. Significantly, the architecture also removes the distinction between short-term and long-term data by completely eliminating the concept of a separate filestore, thus providing a good foundation for the construction of a *persistent programming environment* [9, 11]. The high level programming language LEIBNIZ [44] exploits the features offered by the MONADS-PC architecture to make substantial progress towards the provision of such an environment.

1.1 Rationale

The research leading to this thesis identified two deficiencies of the MONADS architecture. These were

(1) the lack of support for *distribution*, and

(2) issues relating to the *reliability* of the store.

The first deficiency meant that every MONADS-PC computer operated in total isolation, and that it was not possible for users on different machines to share resources such as programs and data, or even to exchange electronic mail messages. A solution to this deficiency had been alluded to in [3], but the solution was very general, and many specific problems had not been addressed.

The second deficiency meant that uncontrolled system shutdowns left the virtual store in an inconsistent state, with serious implications for both data integrity and control of access to data. Moreover, it was not possible to satisfactorily recover from such a failure.

To the reader, this second deficiency may appear to be unrelated to the first, and may thus be perceived to be a totally separate research topic. In fact this is not the case because connecting discrete computers to form a distributed system

(1) increases the possibility of failure because of the increased number of potentially faulty system components, and

(2) extends the potential domain of effect of the failure of one of the machines to any subset of the set of machines connected to form the system.

The environment provided by the LEIBNIZ language and the MONADS architecture provided no support for distribution, and was rather fragile. It therefore did not completely satisfy the requirements of a distributed persistent programming environment.

1.2 Aims

The first major aim of this research was to achieve distribution of the MONADS architecture by connecting an arbitrary number of discrete MONADS computers to a local area network. The created system would enable users of the connected computers or *nodes* to share resources such as programs and data, and devices such as printers. Several important properties were required of the distributed system. These related to the user's perception of the services offered by a connected node compared to those offered by a discrete machine, and were as follows

(1) The distribution of the system would provide users with improved functionality without compromising the functionality available from a single, non-connected machine.

(2) A resource would be identified by name, and that name would define the resource only and not its current location. As a consequence the resource could be moved to another node and still be accessed using its original name.

(3) All users at all nodes would have a coherent view of shared data.

(4) The owner of a resource would have control over access to the resource on a network-wide basis.

(5) Programs developed for use on a non-networked machine would execute without change over the network.

The MONADS architecture already provided a large, system-wide virtual memory space with explicit support for information-hiding modules protected by capabilities. The natural method for achieving distribution of the architecture was to extend this memory space to encompass the connected nodes, thus creating a *distributed shared memory* (DSM). A number of other researchers had previously created DSMs [36, 74]. The techniques developed by these researchers did not support the creation of *large* shared memories, meaning that their usefulness was restricted to applications such as data sharing between parallel processes. To enable the sharing of *all* resources, it was necessary to develop *scaleable* techniques.

As work on the creation of the DSM progressed, and the implications of failure of a node became apparent, a second major aim emerged. This second aim was to produce a *fault-tolerant* virtual store. This store would recover from hardware or system software failure by reverting to some previous consistent state in much the same way as database systems may *roll-back* after a failed commit operation [7]. A store with this property is said to be *stable*. The features required of the techniques developed to implement stability were

(1) flexibility to allow logically related groups of data to be rendered stable separately from the other data in the store,

(2) the ability to initiate a stabilise operation from both system and user software, thus providing a basis for the construction of higher-level transaction based systems,

(3) efficient usage of disk space and processor cycles, and

(4) extendability to a network of connected machines.

In this thesis we show that both of these major aims have been achieved.

1.3 Organisation of the Thesis

The thesis is organised as nine chapters.

Chapter 2 provides the reader with background knowledge of networking in general, and presents several approaches to the problem of providing distributed access to resources. The concept of Distributed Shared Memory (DSM), which is the approach to distribution used in this work, is then introduced.

Chapter 3 introduces the concept of persistence. The MONADS architecture, as mentioned earlier in this chapter, had the potential to satisfy all the requirements of a persistent store. In chapter 3 the persistence paradigm is presented, and existing single-user and distributed persistent systems are described.

Chapter 4 describes the MONADS architecture. Understanding of this architecture is necessary because the subsequent chapters present modifications and extensions to it.

Chapter 5 presents extensions to the single-node MONADS architecture which implement a DSM approach to distribution. Access to resources is achieved in a manner which makes their physical location totally transparent to the user. The architecture presented in this chapter assumes that programs and data are never moved from the disk on which they were created, and that a disk is never moved from the node on which it was formatted and partitioned.

Chapter 6 addresses the restricted nature of the model presented in chapter 5. In chapter 6, schemes which allow disks to be mounted on any active machine, and which allow data and programs to migrate between disks and machines are presented. These schemes are significant because they allow such movement of resources without compromising the location transparency achieved in chapter 5.

Chapter 7 addresses the issue of stability. The chapter starts with some background discussion, and then presents a scheme for achieving stability of the single-node MONADS store. Simply adding this stability scheme to the MONADS architecture, together with LEIBNIZ, would provide a system satisfying the requirements of a persistent programming environment. An extended scheme which provides a stable distributed system is then presented.

Chapter 8 describes an implementation of the distributed architecture using an experimental system consisting of three MONADS-PC computers.

Chapter 9 concludes the thesis by summarising the achievements of this research and indicating directions for future research.

Chapter 2 NETWORKS AND DISTRIBUTION PROTOCOLS

2.0 Introduction

The physical interconnection of discrete computers to form Local Area Networks (LANs) is well understood. Processors are typically connected to a contention bus such as Ethernet [40, 79] or by point to point links to form a ring which is managed using protocols such as Token Ring [48], Slotted Ring [91], or Cambridge Ring [85]. These LAN systems provide a means of reliably transferring messages between nodes at a speed similar to the transfer of data from a disk.

Higher level protocols build on the message passing facilities provided by LAN technology to provide services to the user. We consider that desirable attributes of the resultant network include:

- interconnection transparency: users need have no knowledge of the fact that they are using a network of processors rather than a uni-processor,
- location transparency: the name of a resource (e.g. a file or device) does not define the node on which it currently resides,
- naming transparency: the same resource name used on different nodes will identify the same resource,
- coherency: all nodes have the same view of shared data,
- security: the owner of data has control over access to the data on a network-wide basis,
- fault tolerance: the ability to fall back to and continue from a previous consistent state after a catastrophe such as a node crash.

In this chapter we first describe the ISO/OSI Reference Model [58], which defines a layered basis for protocol development. Protocols conforming with this model are currently emerging [52], and the adoption of the model as a global standard is not expected until the mid

1990's. We then describe existing network architectures based on the message passing, remote procedure call, and distributed shared memory approaches.

2.1 The OSI Reference Model

The Open Systems Interconnection (OSI) Reference Model [58] provides a layered view of network architecture. This approach was taken ".. to provide a common basis for the coordination of standards development for the purpose of systems interconnection, while allowing existing standards to be placed into perspective .." ([58] page 5). The model provides a functional and conceptual framework for the further development of layer standards while not specifying the actual implementation of the layers.

The OSI Model provides seven layers, numbered 1 for the lowest layer up to 7 for the highest. Each layer except for layer 7 provides services to the layer above it, shielding the higher layer from details of how those services are implemented. Similarly, each layer except for layer 1 uses services offered by the layer below it. Layer 7 is the means by which application processes access the services offered by the ISO environment, and layer 1 provides access for the ISO environment to the physical communication medium.

As well as using and providing services, each layer entity communicates with its equivalent layer entity at other nodes in the network according to a layer protocol. Layers at the same level on different nodes are called *peer* layers, and the communication between them is virtual, achieved using the services offered by the lower layers. Before layer n entities on nodes A and B can communicate, they must be connected, which means that communication must exist between the entities implementing layer $n-1$ on nodes A and B, which in turn means that communication must exist between layer $n-2$ entities, and so on down to layer 1, where physical connection between the nodes exists. This protocol means that there is a considerable overhead in connection establishment messages for each layer prior to the communication of user data.

When a user application message is transported from source node to destination node by an OSI network, it may pass through several intermediate nodes en route. Layers 1, 2, and 3 form the *Communication Subnet*, which means that they provide the means for data to travel from a node to the immediately adjacent node. Layers 4, 5, 6, and 7 are called *end-to-end* layers because communication between peer layers at these levels is between the implementing entities at the source and destination nodes, and the layers are not involved in intermediate nodes.

The layers, and a summary of their purposes as stated in the Reference Model document are

(1) *Physical Layer*: provides the mechanical, electrical, functional, and procedural means for the transmission of bits across physical connections.

(2) *Data Link Layer*: takes the raw transmission facility of the physical layer and transforms it into an error-free facility. Data is broken up into *data frames*, which are transmitted sequentially. Functions performed by the layer include sequence control for frames, error detection and recovery, reporting of unrecoverable errors, and flow control.

(3) *Network Layer*: provides the transparent transfer of data between the end-to-end layer above, even when the connection spans several subnetworks that offer dissimilar services. Functions performed by the layer include routing and relaying, network connection multiplexing onto data link connections, error detection and recovery, sequencing, flow control into the subnet, expedited data transfer (if specified), and management issues such as statistics and accounting.

(4) *Transport Layer*: is the lowest end-to-end layer, and shields the higher layers from knowledge of the underlying network. It can multiplex a number of transport connections onto a single network connection, or alternately split a transport connection over a number of network connections to improve throughput. Functions performed include end-to-end sequence control on

individual connections, end-to-end error detection and recovery, end-to-end segmenting, blocking and concatenation, end-to-end flow control on individual connections, and expedited data-unit transfer.

(5) *Session Layer*: provides the facilities necessary for management of the dialogue between peer Presentation Layer entities. It can transparently recover from loss of connection, provide transaction management (useful for distributed database systems), control interaction so that a full-duplex link appears as half-duplex or simplex, and maintain synchronisation points in a dialogue so that reversion to an agreed point and restarting from that point can occur.

(6) *Presentation Layer*: is concerned with the syntax of the data. It can potentially implement three different syntaxes for one transmission, the originating syntax, the transfer syntax, and the receiving syntax. The layer can perform transformations such as ASCII to EBCDIC if required by the hardware of the communicating nodes.

(7) *Application Layer*: provides the interface by which an application process on one host can communicate with an application process on another host (e.g. file transfer, job transfer, database access). Services provided are grouped into general and application specific, though an application may use services from each each group.

Layers one and two of the OSI model, the *Physical* and *Data Link* Layers, specify a basic message passing facility in which errors are detected and possibly rectified. In this thesis we are interested in building *on top* of such basic message passing, and so we will not discuss issues concerning basic message passing any further. We simply assume that a facility providing a Data Link Layer interface is available, and that it is either working correctly, or not working at all.

2.2 Existing Network Protocols

The interconnection of computers to form networks adds significantly to the utility provided by a group of discrete computers. Using a network it is possible for resources such as disks and printers to be shared by multiple machines. Users appreciate utilities such as electronic mail, and the ability to access remote computers by remote login. Application programs running on networked computers can access and manipulate data stored on other machines. Implementing such facilities requires that processes running on separate computers communicate with each other via the network. The term for the communication between processes is *Interprocess Communication* (IPC). IPC between processes executing on the same processor is trivial. IPC between processes executing on different processors may be of several forms.

Tightly coupled multiprocessors, as commonly used for parallel processing, achieve IPC through the use of shared physical memory connected to a common bus which allows a process running on one processor to deposit data in a memory location, and a process running on another to subsequently access the data by a simple memory read.

Physically distributed processors are not connected to a common bus, and thus cannot share physical memory. IPC can occur between processes running on physically distributed processors using a number of different protocols, including message passing, Remote Procedure Call (RPC) [116, 124] and Distributed Shared Memory [112].

Message passing involves the depositing of data into a message block by the sending process, the making of the message available to the receiving process, and the subsequent reading of the data from the message block by the receiving process.

RPC allows a process to invoke a procedure running on a remote processor, and to transmit and receive data back in the form of parameters to the procedure call. Implementation of RPC requires the existence of an underlying message passing system.

Distributed Shared Memory (DSM) is another paradigm that allows IPC. DSM provides the abstraction of a shared physical memory to processes

running on processors that are not tightly coupled. It has the advantage that processes can share data structures like arrays and linked lists as well as the flat data that can be logically shared using message passing and RPC.

2.2.1 Message Passing Systems

Message passing systems effectively extend the underlying communications mechanism by allowing shared information to be passed by value. There are a number of message passing systems in general use. These include IBM's SNA [78], OSI based MAP and TOP [52], and TCP/IP [30]. In this section we concentrate on TCP/IP, which appears to be a defacto standard on workstation-based local area networks.

The Transmission Control Protocol/Internet Protocol (TCP/IP) [30] defines a set of protocols which allow the interoperability of computer equipment connected to a LAN. The LANs may then be further connected by gateways to form a virtual wide area network (WAN) known as an *internet*. Users of the internet are offered a range of services including electronic mail, file transfer, and remote login.

The protocols make no assumptions about the underlying network hardware or type, meaning that it is possible for computers connected to different network types to communicate as long as the networks themselves are connected by a gateway. Each computer connected to an internet is assigned a unique name, called an *IP address* defining both the network to which the computer is connected, and the computer itself within that network. This address is not the same as the physical node address that allows the network hardware to identify the computers connected to each individual LAN. In this sense a TCP/IP network is virtual.

IP breaks messages up into small units called packets or datagrams. IP communication is *connectionless*, meaning that each packet contains the full source and destination addresses, and is routed separately. An applications programmer using a connectionless service like IP to achieve communication must handle issues such as flow control, acknowledgements, and error control in the application code, because

packets can be lost, duplicated, or delivered out of order. Provision of a general purpose protocol that allows application programs to transmit large quantities of data, called *streams*, without concern for each individual packet used in the transmission is provided by TCP.

TCP is a higher-level stream delivery protocol built on top of IP. It is a full connection oriented protocol providing a full duplex *virtual circuit* between the communicating parties. This means that, once a connection has been established between the source and destination entities, the application programs can transfer data over the connection while the protocol monitors the connection for errors and attempts to rectify them. If an unrecoverable error such as hardware failure occurs, the communicating applications are informed by the protocol that the connection has been lost. The advantage of connection oriented services such as TCP over connectionless services is that they can provide a reliable data transfer facility to the application program. A disadvantage of the building of TCP on top of IP (a connectionless facility) is that a TCP connection is virtual, with no guaranteed bandwidth, meaning that network congestion can cause fluctuations in the speed of data transfer provided by the connection. Even though the full protocol suite extends far beyond the IP and TCP protocols, it is called TCP/IP because of the importance of the stream delivery service provided by them.

Since we are concerned with LANs in this thesis, we will consider TCP/IP only as used within a single LAN. One of our requirements of a good network system is location transparency, and the first step in achieving this is the provision of a virtual name for a node rather than the physical name that is used by the network hardware to pass data from one node to another. The TCP/IP virtual name for a node is its IP address, and nodes with attached disk store their IP address on that disk. The method used by a computer connected to a TCP/IP LAN to determine the physical address of another machine, and its own IP address if it is diskless, is of particular interest to us in this thesis because of our location transparency requirement.

Dynamic mapping between virtual TCP/IP addresses and physical network addresses, and vice versa, is achieved using the Address

Resolution Protocol (ARP) and Reverse Address Resolution Protocol (RARP).

ARP allows mapping from internet addresses to physical addresses. Consider node N_1 which wants to communicate with node N_2, but which only knows N_2's internet address I_2. N_1 broadcasts a special packet requesting that the node with internet address I_2 supply its physical address P_2. All nodes, including N_2, receive the request, but only N_2 responds (other nodes use the message to update their address mapping table with N_1's data if necessary). On receipt of the response, N_1 stores the internet to physical address mapping, and can now communicate with N_2.

RARP allows a node to find out its own internet address (particularly useful for diskless nodes), and relies on the existence of at least one RARP server. RARP servers maintain a table mapping physical to internet addresses for all computers connected to the LAN. If node N_1 wishes to know its internet address, it broadcasts a RARP packet containing its own physical address P_1. The RARP server(s) respond by transmitting a packet containing the required internet address I_1. The existence of multiple servers can be advantageous in the case where a server is busy when the RARP request is sent, but can result in contention problems in Carrier Sense Multiple Access with Collision Detect (CSMA/CD) networks when several servers attempt to reply simultaneously.

TCP/IP is a widely used and popular suite of protocols. When considered with respect to our desirable network attributes, however, it does not perform well for the following reasons:

- users are very aware of the fact that they are connected to a network because local commands are different to remote ones. For example, mail to a local user requires provision of the user's name only, whilst mail to a remote user requires the provision of location information. Of course this can be hidden by application specific mechanisms such as *aliases* that sit above TCP/IP,

- access to a resource requires knowledge of the location of the resource because all access is achieved by use of the internet address, which defines location. Aliases, and higher level protocols such as Network File System (NFS) [111], however, can be used to hide the differences between local and remote resources by providing a flat name space,

- location forms part of the internet name for a resource. Thus a resource name defines the resource for all nodes other than the host node, where the internet address is not needed,

- there is no provision for shared data in TCP/IP other than sharing by use of remote login, such sharing being controlled by the host operating system,

- control over access to data is limited to the protection mechanisms provided by the operating system running on each node, and

- there is no provision for fault tolerance.

2.2.2 Remote Procedure Call Systems

The tacit assumption in the setting up of connection oriented communications is that both parties initiate the transmission of data, and the full duplex nature of the link in fact provides the facility for data to travel in both directions simultaneously. In many situations this assumption is incorrect. A file server, for instance, transmits data only in response to a request from another node. The node in this situation is called a *client*, whilst the file server is a *server*. The *request-response* nature of the communication between a server and a client is similar to the situation in which a program calls a procedure, and is returned a result. Researchers such as Birrell [17] have adopted this view, and developed mechanisms by which request-reponse communication can be achieved using procedure calls in application programs. This type of communication is known as Remote Procedure Call (RPC).

Implementations of RPC require an underlying virtual network to transfer the procedure call message between nodes, and many use the virtual network provided by TCP/IP, for example Sun RPC [110], Apollo NCA/RPC [42], and Mayflower RPC [13]. The application programmer view of RPC varies between implementations. Some provide an interface which makes RPCs look the same as local procedure calls, e.g. Cedar [17], whilst others intentionally provide different syntax for local and remote calls, e.g. Mayflower [13]. Neither type of implementation is able to handle parameters of the complexity possible with local procedure calls. This is because complex data structures are built using pointers, and pointers are meaningless when taken out of the context in which they are created. A review of RPC implementations is contained in [116].

As described in [115], RPC may be implemented by linking one or more library procedures called *stubs* into the client's address space. Stubs can initiate messages to servers, and receive replies. An application program can make a procedure call to a local stub, which then collects the parameters to the call into a message. The stub then calls on the local transport protocol entity to transmit the message to the server machine. The message is passed to the appropriate server process stub, which unpacks the parameters, and calls the server procedure with a local procedure call. The server procedure is thus unaware that it is, in fact, being activated remotely. The return to the RPC is achieved in similar fashion, with the effect that a client-server communication takes place.

RPC implementations can be classified as either *blocking* or *non-blocking*. Blocking RPC suspends the calling process while the RPC is being processed, and so is a true simulation of local procedure call. Non-blocking RPC allows the calling process to continue immediately without waiting for the returned result, thus approximating a message passing system that explicitly invokes a remote procedure rather than relying on a remote process receiving and replying to a message. Most implementations of RPC provide for optional blocking or non-blocking operation, with no compulsory result being required for the non-blocking form.

Transport protocols can be optimised for use with RPC. The V Message Transport Protocol (VMTP) [26], for instance, has been designed to perform best for the request/response communication used in RPC. Rather than establishing and releasing connections as in OSI, the VMTP protocol provides connection establishment with the delivery of a request message. The connection status is dynamically updated with the receipt of subsequent requests. Receipt of a message is acknowledged by the response, so that a typical RPC communication consists of only two messages, a request and a response. Another optimisation used by V [27] and Amoeba [84] is improvement to buffering. Invocation of RPC results in the calling program setting up buffer space from which request data is directly transmitted and into which the response is directly written, meaning that creation of intermediate message buffers and copying to and from these is not necessary.

RPC itself has been used in the implementation of distributed operating systems such as Amoeba [84], distributed application development environments such as Mayflower [13], and operating system utilities such as Sun's Network File System (NFS) [111].

2.2.2.1 Distributed Operating Systems and RPC

Distributed operating systems attempt to provide the abstraction of a centralised time sharing system to users. This abstraction is achieved using multiple processors connected by a network. Details such as the location of files, the interconnection and communication mechanisms used, and the processor being used are hidden from the user by the system. Examples of distributed operating systems are Locus [92], V [27] and Amoeba 4.0 [84]. We examine V and Amoeba 4.0 in more detail as they are the most recently released systems, and tackle a number of problems that are similar to those encountered in the work covered by this thesis.

2.2.2.1.1 V

The V project [27] is a continuing research project which aims to provide the resource and information sharing facilities of a centralised

system by controlling a cluster of high-performance workstations and a high-speed network with a distributed operating system. The V distributed operating system consists of a kernel that runs on each of the connected workstations, and services that are implemented at the process level. The kernel provides network-transparent *address spaces* in which processes run, *lightweight processes* that can share an address space, and *IPC* which is achieved by blocking RPC. The kernel thus provides the means for connection between applications and service modules without itself implementing most services. Services that are implemented in the kernel include those that manage processes, memory, communication, and devices.

Processes, process groups, and communication endpoints are identified by unique numbers called *entity identifiers.* These are 64 bit binary values that are host-address independent, meaning that a mapping is needed between entity identifiers and host addresses. This mapping is maintained by the kernel using a mapping table along with multicast messages to query other kernels about mappings that are not currently in the table, or to update the table after, for instance, process migration. To ensure that identifiers are unique on a network-wide basis, the node kernels cooperate in allocation of identifiers to ensure not only that a proposed identifier is currently unused, but also that an identifier has not been recently used. This second proviso is necessary because of the finite size of entity identifiers, which means that reuse of the identifiers eventually becomes necessary. It is designed to minimise the possibility that use of an entity identifier could result in access to the incorrect process. This could occur if the original process had terminated and the identifier had been reused while an entry mapping the entity identifier to the old process still existed in a kernel mapping table.

IPC is request-response based. *Client* processes use the kernel *send* primitive to request a service from a *server* process. Servers may implement either the message or RPC modes of operation. The mode of operation of the server process is not apparent to the client process because it blocks pending a response from the server. If the server implements the message mode, it uses the *receive* kernel primitive to receive the next request message. It invokes a procedure to process the request, and sends a response back to the client process. If the

requested server process is busy, the request message is queued up, thus providing serialisation of requests. In RPC mode the server executes as a procedure invoked by the client process. The use of the RPC mode allows the concurrent handling of server requests, with its associated performance benefits. Send kernel calls are trapped by the local IPC module if the server is local, and handled by the network IPC module otherwise.

To achieve IPC, the client uses the send kernel primitive to make a call with data (parameters) in a client-supplied buffer, and the response data is delivered by the kernel into this buffer. The network issues of error handling and flow control are catered for by the use of the response message both as acknowledgement and authorisation to send the next request. Messages are of a fixed length of 32 bytes, but a *data segment* of up to 32 kilobytes in length may be attached. The kernel interface, buffering, and network packet transmission and reception are optimised to handling the 32 byte fixed length messages because more than 50% of the V network traffic is short messages. The V Message Transport Protocol (VMTP) [26] is used for network IPC. This protocol performs well under request-response usage because there is no explicit connection establishment or release phase as exists, for instance, in OSI. VMTP connection occurs with the receipt of a client request, and is updated with each subsequent request. Short messages are incorporated into the message header. Each process descriptor contains a template VMTP header, with some fields initialised on process creation, reducing the time taken to prepare a send packet. A short message is loaded into application registers, copied into processor registers and written into the process descriptor header template, which is then queued for transmission. It is interesting to note that V network IPC time measurements indicate that it is faster to import an 8KB block of data already in the physical memory of a remote node than it is to load the same block from local disk [27].

VMTP supports multicast, the sending of a message to a group of destinations. Multicast is used, for example, for dissemination of load information as part of distributed scheduling, and for synchronisation of the V time servers. Processes may be collected into groups, for example the group of file servers or the group of processes executing a certain parallel program. Group identifiers belong to the same name

space as process identifiers. A group send can contain a qualifying process identifier or process group identifier. This can be used to address from a group of servers, the particular server(s) acting for the identified process(es). An example of the use of this facility is process scheduling, which is distributed with one process scheduler per node. A suspend operation on a process P is achieved by sending a message to the group of process schedulers with P's process identifier provided as a qualifier. The kernel routes the message to the host node for P (using the entity identifier to host addresses mapping table), meaning that it is delivered to the process manager responsible for P, and not to the other process schedulers. A client knowing the identity of a group of servers can thus request an action on a process without knowing the identity of the specific server responsible for that process.

The V kernel maintains the physical memory as a cache of pages from open files. The address space in which a process runs is organised as set of ranges of addresses called *regions*. Each region is bound to a portion of an open file and provides the process with a window onto that portion of the file. The kernel manages the binding of regions to open files, the caching of blocks from the open file, and the maintenance of consistency. Consistency is achieved using locking at the file server together with a block ownership protocol. When an address space is initialised prior to program execution, an address space descriptor is allocated and the program file is bound to the address space. The process references a virtual address in the address space causing a page fault to occur, because binding between the address space and a cached copy of the addressed portion of the program file has not been set up. The kernel maps the referenced virtual address to a block in the program file, and if the block is in the kernel page cache it binds the block to the process' address space. If the block is not in the kernel page cache, it is requested from the server that is managing the program file and mapped into the process address space when it arrives. Thus there is no need for a kernel program loading mechanism. Program access to permanent data is achieved in a similar fashion after the newly opened file is bound to a region of the address space.

The term *object* in V means a resource, for example a process, an address space, a communications port, or an open file. All servers are

effectively object managers and implement names for the set of objects they manage, meaning that an object specified by name can be handled by the server without reference to a name server. This relies on the client process being able to determine which server handles the object. When an object manager creates a directory of object names, it allocates a globally unique directory name that is used as a prefix to the names of all objects in the directory, and adds itself to the *name handling group* of processes. The manager for any object can be found by multicasting the character-string object name to the name handling group. When a program is initiated, a table of name prefix to object manager mappings is initialised, and this table is maintained while the program runs. Table entries are not updated immediately by some coherency protocol whenever a described mapping changes. The approach taken is that out-of-date data is detected when the data is used. When an operation is invoked on an object manager using a stale name mapping, the operation will fail, resulting in the deletion of the incorrect table entry and its replacement after a multicast query obtains the correct mapping information.

Once a character-string name has been mapped to an object, for example on a file open operation, an *object identifier* is used in subsequent references to the object. Object identifiers are formed by the concatenation of the object *manager-id* with the *local-object-id.* The manager-id is an IPC identifier specifying the manager that implements the object, whilst the local-object-id specifies the object relative to the manager. When a manager crashes, its manager-id is invalidated. A new id is allocated to an object manager on restart of the manager, or on reboot of the system. Thus object identifiers can only be allocated to objects such as open files or address spaces that have a shorter lives than their managers, and character-string names are used for long-life objects such as files.

If an object manager is replicated or distributed across nodes, a client uses the manager group identifier to access the server implementing the required object, and then the specific server identification for subsequent accesses. If the object migrates or the server entity crashes, the client receives an error message on the next access, and must then revert to the server group identity to allow rebinding to the new implementing server entity. Operations such as replicated file writes

use the group address to perform updates on every copy, ensuring that every replica is updated by checking off response messages against its list of individual servers.

In terms of our stated desirable qualities, V performs quite well. A V network provides an abstraction of single-machine operation in which the user need not know about the distributed nature of the resources he is using. Resource naming is location transparent in that the name prefix defines the server or server group that implements the resource, not the physical location of the resource. Naming transparency is also provided because resource names are unique at the server level, meaning that once the server identity prefix is added the name is network unique. A rudimentary coherency scheme ensures a consistent view of shared data. Naming lacks uniformity because of the difference between long term object names and short term object identifiers. Security and fault tolerance are not strong features either, although work is in progress on a distributed atomic transaction protocol which should assist fault tolerance.

2.2.2.1.2 Amoeba 4.0

Amoeba is a distributed operating system consisting of a distributed kernel and suite of services accessed through the use of RPC. The hardware it controls is categorised as consisting of four components

- *workstations* which execute processes such as window managers and editors that require constant user interaction,
- *pool processors* which provide most of the processing power, typically consisting of many single board computers each having private memory and a network interface,
- *specialised servers* which run dedicated processes with special resource demands (e.g. file servers run best on machines with disks), and
- *gateways* to other Amoeba systems that are accessible only over wide area networks (WANs). The gateways shield local nodes from knowledge of WAN protocols.

Amoeba is object based, with an object being defined as "a piece of data on which well-defined operations can be performed by authorised users" [84]. Objects are accessed and protected by use of unique *object capabilities*. Capabilities are kept secret by randomly choosing them from a 128 bit wide sparse name space, and are protected from illegal modification by the inclusion of redundancy and a checksum field. Each object is accessed by a *service port* whose identity forms part of the object's capability. Objects are managed by *servers*. A server is implemented as a lightweight process that shares the address space of the object. Multiple server processes can jointly manage a group of similar objects providing a *service*. Client processes use RPC to request servers to carry out operations on objects.

A server process indicates its willingness to accept requests to a particular service port using a system call called a *get_request.* This informs the kernel that the server is prepared to accept messages addressed to the port. Get_request calls are used by the kernel to maintain an internal table mapping service ports to servers. Client processes perform RPC using the *do_operation* system call which contains the capability for the object and an operation code (which defines the selected operation) as a parameter, and server processes reply to the RPC using the *send_reply* system call. Amoeba RPC is blocking, meaning that client processes are suspended until a reply is received.

When a client calls do_operation, the local kernel extracts the service port identity from the presented object capability (the capability is verified by the server) and checks its internal table in an attempt to find a server implementing the service. It is possible for a kernel to receive a do_operation for a service port for which it knows no server location, or for a port whose server has died or migrated since data for that port was inserted. In this situation a message is broadcast by the kernel seeking another kernel with an outstanding get_request for the port. If a reply is received, the port/network address pair is stored for future reference and the RPC can continue.

To shield users from knowledge of the binary capabilities used to access objects, a central *directory service* is provided. Directory entries consist of name/capability pairs. The basic operations possible on

directory objects are lookup, enter, and delete. Arbitrarily complex graph structures can be built because directories are themselves objects, meaning that capabilities for directories can be included in other directories. Users must, however, possess a capability for the root directory of such a structure in order to traverse it. The directory service replicates its internal tables on multiple disks so that single node failure will not prevent it from operating.

The Amoeba file service, the *bullet service*, is unusual because files are *immutable*, meaning that a file cannot be modified once it has been created. Files are also *contiguous* both on disk and in bullet server cache, so the only management information needed for a file is its start location, size, and ownership information. The only file services supported by the bullet service are read, create, and delete. Creating a file involves writing all the data in one operation, and receiving a capability for the file. Typically, a file name would then be assigned and a directory entry made.

Data consistency for sets of objects (atomicity) is possible through the provision of an atomic update facility in the directory service. This facility allows atomic changes to be made to the mappings between sets of names and sets of capabilities. Thus if an atomic update of a set of objects is required, the modified versions are written to new files, and the new capabilities for the files are entered as a single operation into the directory.

There are two possible states for Amoeba processes, *running* or *stunned.* A stunned process exists, but does not execute instructions. Stunning is used for program debugging, and for process migration. When a process is stunned, the kernel passes its state in a process descriptor to a *handler*, which is a service that handles anomalies such as process exits and exceptions for the process. The capability for a handler is included in the process descriptor for every process.

Process migration is achieved by stunning the process, after which the handler passes the process descriptor to the new host. The new host copies the memory contents relevant to the process, then restarts it. The new capability for the process is passed to the handler, which kills the process at the old host. Attempts to communicate with a migrating

process receive 'process stunned' replies until the process is killed, and 'process not here' replies after the process is killed. The new process must then be located as described above before communication can resume.

Processes exist in a segmented virtual address space. Segments can be mapped into and out of this address space. Segment mapping operations can be accompanied by a capability. A process can unmap a segment, which remains in memory (though not in the process address space), and pass the capability for the segment to another process which can map the segment into its address space. If the processes are running on different machines, the segment must be copied between their memories.

The use of service ports as a means of accessing the server(s) that implement a service introduces a potential security problem. Service port identities are large binary strings (48 bits), and knowledge of a port identity is sufficient to allow access to the port. A server will only perform a requested service if a valid and appropriate capability is presented with the request. It is possible with such a scheme for a malicious process to pretend to be a server by issuing a get_request on a server port. The process could then wait for calls on the service, and illegally collect the object capabilities that accompany such calls. The solution used in Amoeba is to place an interface box or *F-box* between each processor module and the network. This device, which can be implemented in software (e.g. in a trusted operating system), or in hardware, transforms all messages entering and leaving the processor.

Under the F-box scheme, each service port is represented by two ports, a privately known get-port *G*, and a put-port *P* known to server clients. The F-box implements a publicly known one way function *F*, where the relationship between *P* and *G* is $P = F(G)$. To indicate its readiness to accept client requests, a server uses the get_request system call, including its get-port identity *G*. The F-box computes the corresponding put-port identity *P*, which is made available to clients. Clients use *P* when making requests on the server. The client's F-box does not transform *P* in the outgoing do_operation message. A process wishing to masquerade as a server does not know the private get-port identity for the real server, and so cannot provide the correct *G*

parameter for the get_request system call. The one-way property of *F* prevents the bogus server from calculating a value for *G* that will result in its local F-box presenting the correct *P* identity to clients. Replies from the server to the client are similarly protected, with the client using a get-port *G'* and including *P' = F(G')* in the do_operation message.

Amoeba 4.0 is successful in providing an abstraction of a single powerful central processor using a pool of lower powered processors. Resource location transparency is provided through the use of location independent service ports as a means of access to service implementations. Access to objects is achieved by object capabilities. which provide naming transparency and protection. The combination of service ports and capabilities potentially allows a malicious process to masquerade as a server and illegally obtain object capabilities. To prevent this an encryption scheme has been developed for the server part of object capabilities. Consistency of information and fault tolerance are Amoeba's main weaknesses. The atomic update facility provided by the directory service provides consistency at the (immutable) file level but is unsuitable for transaction-oriented applications, whilst fault tolerance at the data level has not been tackled yet. Although the directory replication scheme reduces the risk of reliability problems with the centralised directory service, it is a potential source of bottlenecks.

2.2.3 Distributed Shared Memory Systems

Whilst message passing and RPC provide shared access to data by *value*, shared memory systems allow processes to access data in a shared memory space by *reference*. This means that knowledge of the *address* of data in the shared memory space is sufficient to allow the process to access the data. Processes in such an environment are able to share arbitrarily complex data structures such as linked lists that rely on the following of pointers, without the need to flatten and rebuild the structures as required by message passing and RPC.

If the processors are loosely coupled, there is no physically shared memory, so the underlying communications system is used to provide

an abstraction of a shared memory space. To understand how this abstraction is achieved, it is helpful to look at how *virtual memory management* can use a relatively small physical memory together with disk storage to provide the abstraction of a large physical memory.

Virtual memory (VM) managers partition the virtual address space into smaller blocks. Some implementations use fixed length blocks called *pages*, and others use variable length blocks called *segments*. The available physical memory is similarly divided. The processor generates *virtual addresses*, and a mapping mechanism is used to locate in physical memory the VM block containing that address. The physical memory location corresponding to the virtual address is then found using its offset relative to the beginning of the block. The VM block containing the accessed virtual address may not be in the physical memory at the time of access, creating a condition called a *fault*. To enable resolution of faults, another mapping is maintained between VM blocks and their location on disk, so that, if a block is not in physical memory when an address in the block is accessed, the VM manager can copy it from disk.

Distributed Shared Memory (DSM) extends the VM concept across the physical memories of multiple computers that are connected by a network. These computers are called *nodes*. When a processor accesses an address in DSM, and the required data is not in the physical memory of the node, a fault occurs. Resolution of the fault may involve importing the block from another node into local physical memory using the network. The fact that the blocks of a DSM VM space are distributed across multiple nodes creates problems not experienced by single physical memory VM implementations. The major problem is the maintenance of a consistent view of the DSM across nodes. This problem is similar to the problem of *cache coherence* in multiprocessors, as discussed in [71, 75, 108], and the solutions adopted, which are discussed below, are similar to those used for multiprocessors. Other problems include the location of DSM blocks, the protection of data from illegal access, and coping with unexpected node or network failures. DSM does have significant benefits which makes solving these problems worthwhile.

DSM allows the sharing of data between physically distributed machines without the need for application programmers to write network specific code. This is possible because the shared memory is implemented *underneath* application programs, meaning that programs access remote data in exactly the same way as they access local data.

There are four basic algorithms for providing distributed shared memory [109].

(1) The Central-Server Algorithm provides a single page server which maintains the only copy of shared data, and which either provides data on request or stores modifications to the data. Because a central server has to process requests from all nodes, bottlenecks can occur. These can be reduced by use of several servers allocated a range of addresses each, with other nodes having knowledge of the mapping from address to server node. This scheme is called Distributed Centralised Server, and is unimplemented to our knowledge.

(2) The Migration Algorithm is a single reader/single writer scheme in which data is always migrated to the reading or writing site. For efficiency reasons it is usual to migrate the data in blocks, so that programs exhibiting high locality of reference have reduced communication costs. If several processors require simultaneous access to the same block, thrashing can occur. As discussed in [74], remote blocks may be located by broadcast, but strategies such as the use of a number of managing servers can be used to improve efficiency. As a result of the ease of conversion of an implementation of this algorithm to the superior read-replication algorithm, no practical applications of the migration algorithm are known.

(3) The Read-Replication Algorithm allows multiple readers of a data block, reducing the communications overhead for frequently read blocks. The algorithm is multiple reader/single writer similar to the cache coherence protocols described in [36, 71, 75] and is implemented in IVY [74], Memnet [36], and Choices DVM [59].

(4) The Full Replication Algorithm allows multiple readers and multiple writers for a block of data, employing sequencing of reads and writes to maintain consistency. Sequencing is achieved by use of global time stamps on writes and sequencing reads relative to local writes as described in [6] for cache coherence. Such a scheme has not yet been implemented.

In the following sections, we describe several implementations of Distributed Shared Memory.

2.2.3.1 Abramson and Keedy

The first distributed shared memory scheme, proposed in [3] and on which this work is based applies to a paged virtual memory. The blocks of data transferred between nodes are the same size as a virtual memory page, allowing page transfer to be handled as part of the virtual memory management. Each page has a static owner whose identity can be derived from the address of the page, meaning that finding the location of a page is always efficient. The owner of a page is responsible for maintaining a multiple reader/single writer protocol. The scheme is essentially a combination of Distributed Centralised Server and Read-Replication because, while a single node is responsible for the provision of a particular page of data, multiple read-only copies of the page can exist simultaneously in the main memories of computers connected to the network. This scheme will be more fully described in chapter 5.

2.2.3.2 IVY

IVY [74], which is aimed at improving the performance of parallel processing by use of DSM, provides each of the parallel processes with an address space which is partially private and partially shared. For the shared address space, it implements a dynamic page distribution scheme in which each page has an owner but ownership is moved from node to node as the page migrates. The owner of a page is responsible for maintaining a list of nodes having a read-only copy of the page called the *copy set* for the page. If a node wants to write to the page it must become the owner of the page. The owner at the time of the write request transmits the page, and the copy set to the requesting node

(unless the owner itself is the node requesting write access), and the receiving node becomes the new owner of the page. The new owner must then transmit invalidation messages to each member of the copy set, thus eventually becoming the only node with a copy of the page and able to write to the page. If another node wishes to read from the page, the owner changes its access to read-only, transmits a copy of the page and updates the copy set.

Each node maintains a complete copy of the page table of the distributed virtual memory containing, amongst other information, the probable owner, *probowner*, of each page. A node changes the *probowner* information for a page in its copy of the DSM page table if

- it receives an invalidation message for the page, in which case the probowner becomes the source of the message, or
- it relinquishes ownership of the page, in which case the new owner is the new probowner, or
- it receives a copy of the page (to resolve a read fault), in which case the provider of the page is the probowner, or
- it forwards a write page fault request, in which case the source of the request is the probowner because the source of the request is expected to become the owner in a short period of time.

Over time the ownership of a page can vary considerably, and it can become necessary for a page request message to pass through a chain of nodes before a page is found. To make location of pages more efficient, ownership information is regularly broadcast so that nodes can update the probowner field of their page table.

A major disadvantage of IVY is the fact that every node maintains a full page table of the shared memory, together with the status information for each page, in the local memory of the node. As the shared memory space grows, so does the associated page table at each node, meaning that the IVY system is not scaleable to large shared virtual memories. IVY provides naming and location transparency for data in the DSM because such data is accessed by unique virtual address, and shared

memory page tables contain probable owner information. There is no described notion of protection of access to data in the DSM, or of fault tolerance.

2.2.3.3 MemNet

MemNet [36] was an attempt to overcome the input/output (I/O) view of a network, which uses kernel calls to initiate network traffic. MemNet instead provides *memory extension* through the connection of each node, using a special-purpose MemNet device, to a high speed token ring. The MemNet device takes the processing of communication protocols away from the CPU, connects to the local bus, and provides access to remote data at local bus speed.

Advantages of using a ring topology are listed as

- the access time for each ring interface is limited to an absolute upper bound, meaning that the time taken to send a message and receive a reply is bounded
- the token ring provides an explicit broadcast, with each ring interface seeing each packet
- all interfaces see the same ordering of ring transactions, making data consistency easy to implement
- ring accesses are overlapped, providing a pipelining effect because multiple transactions can be on the ring simultaneously

The MemNet device provides the host node with access to a paged shared address space. The pages, or *chunks*, are small at 32 bytes each, and are allowed to migrate between MemNet devices as required. Internally, each MemNet device consists of an interface to the host node's system bus, an interface to the token ring, and some local memory that is divided into a cache and a reserved area. The sum of the reserved areas of all MemNet devices must be sufficient to store all the shared memory chunks since the system does not support disk paging. A chunk must be in the cache section of the MemNet memory

of the device attached to a node before the node can access an address in the chunk.

A reference to an address in the shared address space is sent to the MemNet device, which determines if the reference can be satisfied locally. If not, a message is sent on the ring. The device that is able to satisfy the request does so by modifying the request message to include the required chunk, meaning that a chunk fault can always be resolved in one circuit of the ring. Since the response is predictable and very fast, the MemNet device does not differentiate between accesses to addresses held in the local cache and those requiring message transmission. It simply blocks the local processor, rather than the running process, in each case.

The coherence scheme used by MemNet is single-writer/multiple-reader. Each MemNet device maintains a chunk table with an entry for every chunk in the shared memory. The attributes or *tags* maintained for each chunk include

- *valid*, is there a valid copy of the chunk in the local cache?
- *exclusive*, has this node exclusive access to the chunk?
- *reserved*, is the reserved space for this chunk at this node?
- *clean*, is this chunk unmodified, meaning that it can be flushed without updating the reserved area copy of the chunk?
- *cached location*, where is the chunk in the local cache, if present?

Operations on MemNet shared memory are *read*, *write*, and an atomic *read-modify-write* (RMW). Before data contained in a chunk can be read, the attributes for the chunk must have the valid tag set. If valid is not set, the device sends off a request for a valid copy of the chunk. If the provider of the chunk has exclusive access to it, the exclusive tag must be unset before the chunk is transmitted.

A write operation can continue on a chunk if the valid and the exclusive tags for the chunk are set. If the valid tag is set and the exclusive tag is

not set, then an invalidation message must be broadcast so that other devices can invalidate their copy of the chunk. If neither the valid nor the exclusive tags are set, a message is broadcast requesting a valid and exclusive copy of the chunk. The first receiver invalidates its copy and includes a copy of the chunk in the message. The other devices with valid copies invalidate their copies as the message passes them.

The start of a RMW sequence results in the valid and exclusive tags for the chunk being checked. If they are not both set, actions as described above are initiated. When they are both set, the chunk address of the subject word is stored in a special RMW register. Any request arriving from the network for that chunk is delayed by transmission of a wait packet until the RMW sequence is complete or until it is terminated because of a delay in the write operation, after which the requested chunk is transmitted.

Consistency of the tags is crucial to shared data coherence. Such consistency is ensured by the *Network/Local Tags* protocol, which is followed by each device. The protocol relies on the fact that ring messages are sequential, and so are received by each device in the same order. The local tags for a chunk are read every time a packet referring to that chunk arrives, and the tags, with the packet opcode and a local state bit (that indicates whether the packet was transmitted by this node) determine the actions of the device and the next state of the tags.

Eventually it will become necessary to purge a chunk from the local cache to make room for a new chunk. When a chunk is purged, it is transmitted on the ring and stored in the reserved memory of the first device that has the reserved bit set for the chunk. That device may itself have no free space in its cache memory, but the reserved space is guaranteed to be available for the chunk. MemNet allows multiple devices to have reserved space for the same chunk, though the impact, if any, of this feature on the performance of the system is as yet unknown. The chunk replacement scheme used is modified random replacement. Each device monitors the chunks passing its ring interface, constantly updating the clean tags for each chunk. Chunks with the clean tag set are the first ones to be purged on cache flush because there is guaranteed to be a copy of the chunk in the cache of a

device that has reserved space for the chunk, meaning that the purge does not involve a message to update the reserved location.

As with IVY, the MemNet shared memory is not efficiently scaleable to large shared virtual memories because a page (chunk) table for the entire shared address space, together with the associated status information (tags), is maintained at every node in the network. The nature of the ring network used also places a restriction on the scaleability of the system. Location transparency is provided by the ring-based chunk request mechanism, and naming transparency is provided by the unique virtual addresses used to access data. There is no protection of access to data, and no provision for fault tolerance.

2.3 Conclusion

Many protocols that enable the exchange of data between physically distributed computers have been implemented. These protocols are typically based on message passing, remote procedure call (RPC), or distributed shared memory (DSM). Remote procedure call and distributed shared memory systems rely on an underlying message passing system, abstracting over details of this underlying system.

The abstraction provided by RPC is of a request-response relationship between processes executing on possibly distributed processors. This means that a *client* process makes requests of a some other *server* process in a similar fashion to procedure calls within a single program. Using RPC, for instance, a client process executing on one computer may request provision of a block of data from a disk server process executing on a second computer. The RPC abstraction thus allows the sharing of data *by value* using the familiar procedure call mechanism. A consequence of this is that dynamic data structures such as lists must be flattened prior to sharing because the inclusion of pointers is only meaningful if the data is located in exactly the same place in the client's virtual memory space. In a sense this is similar to the flattening of data structures for storage on disk.

DSM allows the sharing of data *by reference*, meaning that a process may directly access any data, local or remote, by address. An immediate advantage of this is that dynamic data structures may be shared without

the need for flattening and reconstruction. The DSM abstraction is typically provided using similar techniques to those used in virtual memory management, with data being provided in chunks analogous to virtual memory pages. A problem common to the small number of DSM systems previously implemented is that they require the maintenance of a full *page table* at each node in the network. This means that provision of a wide shared virtual memory space requires the dedication of a large amount of local main memory for the storage of this table. As a result the existing implementations are not scaleable to large virtual memories. Other problems associated with existing DSM implementations include the lack of control over access to data, and the lack of techniques to cope with node or system failure, with its associated impact on the integrity of the store and the data structures necessary to maintain it.

Chapter 3 PERSISTENT SYSTEMS

3.0 Introduction

Persistent systems support mechanisms which allow programs to create and manipulate arbitrary data structures which outlive the execution of the program which created them [9]. This has many advantages from both a software engineering and an efficiency viewpoint. In particular it removes the necessity for the programmer to *flatten* data structures in order to store them permanently. Such code for the conversion of data between an internal and external format has been claimed to typically constitute approximately thirty percent of most application systems [9]. In this sense a persistent system provides an alternative to a conventional file system for the storage of permanent data. This alternative is far more flexible in that both the *data* and its *interrelationships* can be stored in its original form. In order to achieve this a uniform storage abstraction is required. Such an abstraction is often called a *persistent store.* A persistent store supports mechanisms for the storage and retrieval of objects and their interrelationships in a uniform manner regardless of their lifetime. The MONADS DSM supports such a store.

3.1 Persistence

Persistence is a measure of the length of time for which data exists and is useable [9]. In traditional systems the persistence of short term data such as local variables and heap items depends on the programming language being used, whilst the persistence of long term data that exists between executions and versions of a program depends on a file system or database management system. Persistent programming systems, on the other hand, provide in a uniform manner for all periods of data longevity. This means that in a persistent programming system the techniques for handling data are independent of the period for which the data persists [8, 81]. In a persistent system, for instance, dynamic structures such as lists and trees may continue to exist *in their original form* despite the fact that the program that created them has ceased execution. This is in contrast to traditional systems, which

maintain data in short term form for manipulation in virtual memory, and change the data to long term form for storage in a separate filestore.

With such traditional systems data must be transferred from the filestore into virtual memory for use by programs, and transferred back to the filestore if persistence beyond the life of the program is required. Mechanisms such as pointers rely on the position of the data in virtual memory, and traditional systems cannot guarantee that data will reside in the same position every time it is transferred into virtual memory from the filestore. For this reason data structures such as lists and trees must be converted to a format suitable for storage in the filestore prior to being transferred to it. The data structures must later be reconstructed as they are transferred from the filestore into virtual memory.

Three principles underpin the persistence abstraction [8, 9, 81]. These principles are

(1) *Persistence Independence.* This means that the length of time that data persists is totally independent of the way in which the data is manipulated. Programmers cannot control the transfer of data between long and short term store. Such transfers are performed by the system itself.

(2) *Data Type* . This means that there are no types of data object for which special cases apply, and all types of data object are allowed the full range of persistence.

(3) *Management Orthogonality.* This means that the mechanism for providing and identifying persistent data objects is orthogonal to the type system, computational model, and control structures.

Systems complying with all three principles are said to display *orthogonal persistence.* Most attempts to produce such systems have used traditional operating systems such as Unix [95] or Mach [4] as a foundation for a persistent store called a *persistent object manager* [21, 35, 121]. These stores in turn support persistent programming languages such as PS-algol [1, 11], Galileo [5], and Napier88 [83].

A small number of persistent systems comprise purpose built architectures/operating systems [33, 102] with associated persistent programming languages such as Leibniz [44, 68], and Pascal/M [98]. Recent work [99] suggests that the use of such purpose built architectures as a basis for persistent systems may provide greater flexibility and performance than is achieveable with systems based on traditional operating systems.

Other persistent programming language designers have attempted to achieve persistence without using an underlying persistent object manager. Such attempts have failed to comply with one or more of the three principles. Some have resulted in languages well suited for manipulating relational databases [77, 105] but providing persistence only for first order relations, in contravention of the Data Type Orthogonality principle. Others [94, 106] associate persistence with type, in contravention of the Management Orthogonality principle. Using these languages it is possible to produce data structures for which the transient copy in memory differs from the copy in the persistent store because part of the structure is of the 'incorrect' order or type. Subsequent transferring of such data structures from the persistent store to local memory would result at best in loss of data, and possibly in the existence of dangling references.

The design of persistent programming languages is beyond the scope of this thesis and is therefore not discussed further. The construction of persistent object managers is of interest because these managers provide an alternative to the persistent store provided by the MONADS DSM.

3.2 Persistent Object Managers

Persistent object managers provide persistent programming languages (PPLs) with a persistent store in which to manipulate data objects. According to [23, 31], properties required of such a store include

(1) *Uniformity*. A single mechanism is used for storage of all types of data.

(2) *Unbounded size.* The physical properties of the storage media are hidden, and the store provides the abstraction of having infinite size.

(3) *Stability.* Failures are hidden, resulting in a store that appears to be error free.

Two distinct approaches to the construction of such stores have emerged

(1) the construction *from scratch* of dedicated architectures such as the MONADS architecture [67], and

(2) the building of a layered architecture *on top of* an existing hardware platform and operating system (for example [9, 21, 22, 23, 117]).

The MONADS distributed persistent store is described in detail in chapter 4, distribution of the store in chapters 5 and 6, and stability of the store in chapter 7. In the remainder of this chapter we discuss the second approach to the creation of persistent stores, and the implications of this approach on distribution of the store.

3.3 Persistence Using Existing Hardware and Operating Systems

The principle underlying the implementation of most persistent stores is *persistence by reachability.* According to this principle, the fact that an object *A* refers to another object *B* means that *B* will persist as long as *A* does [50]. This implies that an object exists in the persistent store if it is referred to by another object in the persistent store. A further implication is that there is a persistent object whose existence is *not* dependent on its being referenced by another persistent object. This object is called the *root of persistence* for the store. The root of persistence is maintained at a well known location in the store, and points to a graph structure containing all the persistent objects in the store. This means that all persistent objects are reachable from the root of persistence.

Data objects in the store are created by programs. Such data may be

(1) transient data (for example local variables) which does not persist beyond the life of the program, or

(2) persistent data.

Examples of stores to support persistence include CMS [9], POMS [29], CPOMS [22], and Thatte's Persistent Memory [117]. The latest generation of such stores, which supports the Napier88 language, has been designed in both single user [23] and distributed [72] forms. This store conceptually provides an infinitely large *heap* in which all persistent objects reside. A consequence of this design is that all objects, regardless of type, are stored in exactly the same way, freeing the object manager from any need to know about the intended use of an object. This means that the store exhibits the *uniformity* property.

In the Napier system programs are compiled into an intermediate code called *Persistent Abstract Machine* (PAM) code [31]. Users or *clients* of the system invoke the PAM interpreter to execute programs. Each such invocation causes the creation of a *local heap* pointed to by a *process header* object which represents the invocation. Since the process header object is reachable from the root of persistence, the process may be restarted after a system failure [72].

The local heap is used by the PAM for the storage of both transient objects[1], and newly created objects that will eventually become persistent. Its implementation and function differs between the single user and distributed versions of the store. In each case the use of a local heap is beneficial because it allows garbage collection to be performed more efficiently by localising transient objects.

3.3.1 The Single User Napier Store

The single user Napier store [23] is constructed as a set of layers above the Unix operating system [95]. The store is physically a single Unix file containing the heap of persistent objects. When the PAM is asked to execute a program, a process, with its local heap, is created as described above. The only objects directly addressable by this process

[1] These include temporary data structures and stack frames.

are those in the local heap. If the process wishes to access an object in the persistent store, the PAM must firstly copy the object into the local heap. This system requires the use of mechanisms to ensure that at most one copy of a persistent object exists in the local heap at any time.

Persistent objects are named using *Persistent IDentifiers* (PIDs) which uniquely identify them. Data within an object is identified by its offset relative to the start of the object. Translation between PIDs and store addresses is performed by a software address translator, with the resultant negative effect on system performance. Such translation benefits from hardware implementation, as demonstrated by Cockshott [28] and others [14]. Every time a PID is used, the system must

(1) determine whether a copy of the object is already exists in the local heap, and if so at what address, and

(2) if a copy of the object does not exist in the local heap, the location of the object in the persistent store must be calculated.

When a process accesses a persistent object for the first time, the whole object is copied into the local heap, and all subsequent accesses use corresponding local heap addresses.This means that if the data contained in the object is modified, the object must be copied back to the persistent store before the changes become persistent.

The second property required of a persistent store is *unbounded size.* Since no store can in fact have this property, an illusion of unbounded size is provided. In order to maintain this illusion using a store of finite physical size, redundant objects must be removed from the store, thus freeing the physical space occupied by the objects. This process is called *garbage collection.* Garbage collection is the act of removing objects that are no longer needed by the user(s) of the store. Such objects include

(1) transient objects such as stack frames and other temporary data structures created by a program and no longer needed by it, and

(2) objects which had been placed in the persistent store, are no longer needed by the owner, and are therefore no longer referenced.

Since all objects are created in the local heap, and most of these are transient, garbage collection is most commonly performed on the local heap. Less frequent garbage collection of the persistent store removes formerly persistent objects which are no longer required by the user and have thus been removed from the persistence graph.

It is possible for the local heap to become full of copies of persistent objects and transient objects which are still required, and so are not yet garbage. In this situation the least recently used objects are copied from the local heap to the persistent store. Some of these objects may in fact be transient, in which case they will eventually become garbage, and will be removed by a global garbage collect operation. This negates the optimisation achieved through the garbage collection of local heaps, so it is important that the local heap is sufficiently large to minimise the possibility of overflow into the persistent store.

The third property required of a persistent store is *stability*. Since a persistent system hides the physical attributes of data, details of the storage of the data, including *failure* of the storage, must also be hidden from the user. A store that provides the illusion of being failure free is said to be stable. Failure of the store may result in

(1) the *destruction* of data caused by, for instance, the disintegration of a disk platter or a head crash, or

(2) the *corruption* of data, due to an unexpected system shutdown which causes the data to become inconsistent.

Physical failure of the storage media is best handled by a dumping or backup strategy, and is not discussed further here.

To ensure that the store remains *consistent*, modifications to the store are achieved as a series of *atomic* operations called *checkpoints*. This means that a modification to the store is either successfully completed, or it reverts to the previous checkpoint state if a system failure occurs.

Checkpointing of the single user Napier store is achieved at the page level, and is achieved by making a copy of pages before they are modified in the stable store. The set of such copies is called a *before look* [21, 23]. The before look for the store is contained in a special part of the store called the *shadow area*. When a process accesses an

object for the first time, a copy of the object is made in the process local heap. If the process attempts to modify the object, and the persistent page containing the part of the object that is to be modified has not been previously modified since the last checkpoint, space is allocated for the page in the shadow area[2], and this is recorded in the stable store red-tape. The before look is not made at this stage because the persistent page has not actually been modified.

Before such a modified object can be copied back to the stable store[3], the shadow copy of the modified pages must be complete. This before look copy and copying of the object from the local heap to the stable store may occur when the heap is garbage collected, as described above. Such garbage collection is necessary because of

(1) lack of space in the local heap, or

(2) as the initial stage of a checkpoint operation.

An update of the store to the next stable state is not permitted until the entire before look is complete and all modified pages of persistent objects have been copied to the stable store, meaning that if a system failure occurs while the store is being modified, the previous stable state of the store may be reconstructed from the page copies in the shadow area. The size of the before look, and thus the shadow area, depends on

(1) the number of pages modified since the last checkpoint, and

(2) whether the modifications were to newly created objects[4] or to pre-existing objects.

---

[2] Thus ensuring that there is sufficient disk space for the copy of the object.

[3] The modified version of the object is copied back to the original position of the object in the store, thus overwriting the previous version.

[4] In which case there would be no copy of the object in the before look.

3.3.1.1 Evaluation

The single user Napier scheme successfully implements orthogonal persistence. The costs involved in the implementation are considerable however, and affect both the performance of the system and its utilisation of the available storage media.

Performance is affected by the need for translation, in software, from PIDs to local heap addresses. The copying of objects into the local heap area, the achievement of stability by copying persistent pages into the shadow area prior to modification, and the later copying of modified objects back to the stable store, has an impact

(1) on performance in terms of processor time consumed by the copy operation, and

(2) on disk space utilisation.

The effect of copying on disk space utilisation is particularly severe because the scheme statically divides the available disk space into store and shadow regions, and does not have the ability to adjust these divisions according to the dynamics of system use, meaning that more space than usually necessary must be allocated to the shadow region.

Lastly, the fact that there is no overall uniform memory mechanism means that the system is difficult to understand, and thus extend. This lack of uniformity is typified by the fact that the unit of transfer from the persistent store to the local heap is a complete object, whilst the unit of transfer from the local heap to the shadow area, and also back to the persistent store, is the virtual page.

3.3.2 The Distributed Napier Store

The distributed Napier store [72, 121] is constructed above the Mach [4] operating system. Mach provides the implementors with features such as

(1) the ability to provide processes with alternate page fault handlers called *external pagers*,

(2) distributed *inter-process communication* (IPC) which allows location independent procedure calls, and

(3) Multiple threads of execution in a single process address space.

The distributed stable store comprises a single four gigabyte paged address space called the *persistent address space.* Like the single user store described above, this store implements a heap of persistent objects, however these objects are addressed using their *virtual addresses* rather than PIDs. This addressing technique means that advantage can be taken of address translation hardware, with fault conditions being handled by the external pager associated with each client. A significant improvement over the single user Napier store has been achieved by the separation of the functionality of the single user system's local heap between a *local page cache*, and a *local heap.* Persistent objects are paged into the local page cache as required, and *newly created* objects only are stored in the local heap. Local heap pages are allocated from the persistent address space, meaning that addressing the local heap is consistent with addressing the persistent store. The local heap pages are logically special, however, because objects stored in them cannot be shared with other clients. Before an object created by a client can be accessed by other clients, it must be copied from the local heap into persistent pages, thus becoming a (potentially) persistent object.

Coherency of the store is achieved at the page level using a multiple reader/single writer protocol similar to that described in section 5.3. Distribution is achieved using a combination of the the Central-server and Read-Replication models described in section 2.2.3. The server function is performed by the *Stable Store Server*, which is centralised and consists of

(1) The *Server Request Handler*, which receives requests from clients in the form of IPC messages and services these requests by passing them to the appropriate components of the server. In handling these requests the Server Request Handler also implements the coherency protocol through its internal *Coherency Manager.*

(2) The *Stable Store Manager*, which reads pages from, and writes pages to the stable medium (the physical store).

(3) The *Stable Heap Manager*, which provides the abstraction of a store consisting of a heap of persistent objects. This manager is called, for instance, when a new persistent object is created.

Each client executes against a subset of the stable store pages which are contained in a local *page cache*. This cache is maintained by the client's *external pager*, which communicates with the Stable Store Server using the *Client Request Handler*. Part of the local page cache contains the client's *local heap*, which consists of a contiguous set of previously unused pages from the persistent address space, and in which the client places newly created objects. The local heap is locked into main memory, meaning that pages from it may not be discarded by the system. Since the local heap contains local objects, its pages are not shared with other clients. The strategy which allows the sharing of a persistent page which contains pointer(s) into a local heap is discussed below.

The local page cache effectively provides each client with a window onto a four gigabyte persistent store. Not all of this store is available for use by clients, however, because

(1) Part of the client address space is occupied by the PAM interpreter and the data structures required by the coherence protocol.

(2) Only half of the apparently available store is available at any time. This is because of the generation-based garbage collection technique [72, 120] used to garbage collect the store and to provide remote access to persistent pages containing pointers into local heaps. The technique potentially requires one or more *copy out* pages for every locally modified persistent page[5].

Coherency of the persistent store is controlled by the Coherency Manager section of the Server Request Handler. The coherency

---

[5] The designers plan to upgrade the garbage collection algorithm to collect from one 4 gigabyte address space to another, thus doubling the size of the persistent store.

protocol moves pages between states according to a finite state automaton. The server request handler maintains the *Export Table*, hashed on page number, to enable coherency control. This table contains an entry for each persistent page currently in a client page cache. Each entry indicates the current state of the page, and two lists. These lists are

(1) The *V-list*, which contains an entry for every client currently holding a copy of the page. A page is always exported with read-only access, and may be provided by any member of the V-list under instruction from the Server Request Handler. When a client requests write access to a page, the other members of the V-list for the page are instructed by the Coherency Manager to invalidate their copies.

(2) The *D-list*, which contains an entry for every client who has seen the page since it was last stabilised. This list is used to enable the maintenance of *associations*.

An *association* is a set of clients who have become dependent on each other because they have seen the same modified data. Each association has an accompanying list of persistent pages which contain the commonly accessed modified data. Associations change whenever a client obtains a copy of a page that has been modified relative to the stable copy of the page. When a client obtains a copy of a modified page

(1) the association to which the client belongs is *merged* with the association of other clients dependent on the page, and

(2) if necessary, the page is added to the list of pages modified by members of the association since the last stabilise.

When any member of an association requests a stabilise operation,

(1) all members of the association must garbage collect their local heap, and

(2) all pages modified by members of the association must be copied to the stable store as an atomic operation, thus ensuring that the store remains consistent.

A modified persistent page held in a client's page cache may contain a pointer into the client's local heap[6]. When another client requests a copy of such a page, or when the client performs a stabilise operation, the local heap object must be moved from the local heap pages into persistent store pages. Because the local heap page(s) containing the object may also contain non-persistent objects, it is not satisfactory to simply make such page(s) persistent[7]. The solution adopted is to

(1) Record all persistent pages in the local cache which contain pointers into the local heap in a *remembered set.*

(2) Maintain a set of persistent pages called *copy-out* pages with each client. The maintenance of a sufficiently large pool of copy-out pages is controlled by the coherency algorithm, which ensures that the server is requested to allocate more of them whenever the pool size falls below a minimum level.

Whenever a page in the remembered set is copied from the local page cache, the local heap objects pointed to from the page are copied into one or more copy-out pages, and the pointers changed to reflect the new location of the objects. Other pages in the remembered set are also scanned and pointers are changed as necessary. Pointers from the local heap to the moved object(s) are modified using a different technique.

By read and write protecting the original local heap page, the local PAM can detect any subsequent attempt to access the old copy of an object moved to a copy-out page, and correct the pointer used so that it points to the copy out version. The next garbage collection operation, which necessarily traverses the local heap, detects and corrects any remaining pointers to the old copy of the object. This means that after garbage collection the protection on the local heap page may be removed. Use of this technique for modifying local heap pointers avoids

---

[6] A pointer from a persistent object to a local heap object renders the local heap object persistent.

[7] Doing so potentially increases the number of transient objects in the persistent store, necessitating more frequent (and costly) garbage collection of the store. The main reason for the maintenance of local heaps is to avoid frequent garbage collection of the persistent store.

the expense of traversing the entire heap whenever a page in the remembered set is exported.

The copy-out pages are also used when the local heap becomes full, and garbage collection is unsuccessful in creating sufficient space for continued operation. The least recently used objects are copied from the local heap into copy-out pages. During this operation some transient objects may be effectively moved into the persistent store, meaning that when they cease to be of use and become garbage, they will not be found during garbage collection of the local heap. The major benefit of the use of local heaps is the ability to localise garbage collection, so it is important that the space allocated to local heaps is sufficiently large to minimise spill-over into the persistent store.

Stability is achieved using an *after look* strategy, meaning that a copy of modifications to an object is made after the modification. In accordance with this strategy, modified persistent pages which are removed from a local page cache as part of page discard or a stabilise operation are written to previously unused *shadow pages* on disk in accordance with the technique proposed by Lorie [76]. This means that, temporarily, both the stable and modified versions of the page are stored on disk. The final stage of a stabilise operation updates the disk page mapping information using Challis' algorithm [25], thus creating a new stable state for the store. Prior to this final stage the disk mapping information points to the preserved previous stable state, thus permitting reversion to that state in the case of a client or system failure.

3.3.2.1 Evaluation

Like the single user version, the distributed Napier system effectively implements orthogonal persistence. The scheme represents a significant improvement on the single user design in terms of the improved utility of multi-user and distributed access to the store. Other features of the distributed design which improve on deficiencies of the single user version include

(1) the use of virtual addresses rather than PIDs for naming objects, resulting in the ability to exploit address translation hardware, thus improving address translation performance,

(2) the achievement of coherency at the virtual page level and the use of an *after look* approach to stability, thus eliminating the pre-modification copying of objects, and

(3) the provision of the abstraction of a heap of persistent objects to the user whilst maintaining a consistent paged view of memory.

The use of the central server rather than distributed server model for page distribution increases the possibility of network bottlenecks in the form of page requests, transmissions, and returns, at the server. Whilst it is sensible to couple the page server and coherency management functions, the location of the coherency manager with its associated write request, invalidate, invalidate acknowledge, and change access messages within the stable store server further the bottleneck problem. In addition, the centralised design results in a system which is totally reliant on the availability of the server. If the server fails, all of the persistent store becomes inaccessible to clients. In a distributed server system, on the other hand, the failure of a server renders only part of the store inaccessible. The design of a distributed server Napier system involves significant extensions to the failure and garbage collection strategies, and is the subject of future research.

Finally, the implementors have reported several problems related to the construction of their store above an existing architecture and operating system [121]. These include lack of operating system support for marking of locally cached pages as unmodified but "precious" (i.e. not to be discarded), deficiencies in exception handling, and a general lack of control over page discard. The solutions to these problems introduce inefficiencies, for example the necessity to non-destructively write to every page brought into the local page cache. These inefficiencies would not occur if a purpose-built architecture was being used.

3.4 Conclusion

Persistent systems may be created by either building above existing hardware and operating systems, or by manufacturing purpose build architectures. Whilst purpose built persistent systems are rare, a number of examples of software implemented persistent systems have been developed. Single user and distributed versions of the Napier88 persistent system exist, and successfully exhibit orthogonal persistence.

These systems, however, are restricted to some extent by the architectures above which they are constructed. The size of the persistent store that they provide is limited to the address width of the host hardware. In the case of the single user system, performance is detrimentally affected by the lack of hardware support for PID to local heap address translation. Development of the distributed Napier system has been complicated by deficiencies in some operating system mechanisms.

The centralised server architecture of the distributed Napier system means that failure of the server prevents all clients from accessing the persistent store. In addition to this, the server forms a potential source of bottlenecks in a system with numerous clients. Finally no mechanism for control of access to data is provided by the store itself. Rather the system relies on the integrity of compilers, trusting that they will never generate illegal addresses.

Chapter 4 VIRTUAL MEMORY AND THE MONADS ARCHITECTURE

4.0 Introduction

The MONADS store provides a paged virtual memory which can be efficiently accessed by very large addresses. Protection of the data in the store, which is persistent, is provided by capabilities. The MONADS Distributed Shared Memory (DSM) described in this thesis distributes this large virtual address space across a number of networked nodes. Blocks of the virtual memory are transmitted between nodes as part of the virtual memory management algorithm. In this chapter we first discuss the conventional approach to virtual memory management. We then describe the MONADS virtual memory and the role of capabilities in the control of access to data stored in it.

4.1 Virtual Memory

Computer memory may be classified as *primary* or *main*, and *secondary*. Primary memory is directly addressable by the processor, and typically comprises random access memory (RAM) and read only memory (ROM). Secondary memory is not directly addressable by the processor, and typically comprises disks and tapes.

Before the introduction of virtual memory, the code and data addressed by a running process was restricted to a size small enough to totally fit into primary memory. The primary memory itself could be partitioned to allow a number of concurrently running processes, with their data, to exist in memory simultaneously. Such an arrangement meant that main memory size placed restrictions on both the number of processes that could run concurrently on the computer, and on the size of processes and directly addressable data. Techniques such as swapping [16], overlays [49], and virtual memory [70] were developed to overcome these restrictions. The first two of these techniques are quite primitive and restrictive, and often result in an inefficient system. Virtual memory, on the other hand, provides uniform handling of program code and data, and its use is transparent to the programmer, meaning that programs do not need special code to take advantage of

the the "extra" memory. It has established itself as a powerful and flexible memory management mechanism.

Virtual memory provides the abstraction of a larger directly addressable memory than is actually present in the form of main memory. It decouples the logical addresses used by programs from main memory addresses, and includes part of the secondary memory in the addressable memory space. In practice the processor is only able to directly access main memory, so virtual memory management involves hardware and software techniques that automatically transfer code and data between main and secondary memory.

Most modern machines use virtual memory based on fixed size blocks called *pages*. These implementations usually encompass the machine's main memory and a dedicated portion of the space on attached disk(s). The virtual memory disk area is called *the paging area*. (confusingly called swap in Unix parlance).

Virtual memory was further extended in MULTICS [86] and the IBM System/38 [15] to allow all files to exist in the virtual memory, thus completely eliminating the need for a separate filestore and creating a *flat store*. This required that every byte of every file have a unique virtual address. The virtual memory width of 36 bits used by MULTICS, for instance, proved to be inadequate as the amount of file data grew. The implementation of wider virtual memories using conventional techniques was impractical because of the size of the data structures needed for their management [113].

The use of virtual memory for program code and data has been extended in a similar, although somewhat more limited fashion, in some operating systems to allow for *memory mapped files* [73], thus allowing files to be linked to and addressed via a process address space. This technique temporarily associates a range of virtual addresses with the data in a file, meaning that such file data can be addressed directly without the need for explicit file handling instructions in program code.

The MONADS system implements new virtual memory management techniques [2] that are used to manage a 60 bit wide virtual memory on

the experimental MONADS-PC [100], and will be used to manage a 128 bit wide virtual memory on the proposed MONADS-MM [103]. The MONADS system allows all data, whether on disk or in main memory, to co-exist in virtual memory, and to be directly accessed using large virtual addresses, providing a flat store of practical size.

In the next section we describe how a conventional virtual memory operates and is managed. This is essential to the understanding of the MONADS memory management scheme.

4.2 Conventional Virtual Memory Management

Paged virtual memory was first introduced on the Atlas computer [70], and is the most common form of virtual memory management. The virtual memory space is partitioned into fixed size blocks called *pages*, and the main memory is partitioned into the same size blocks called *page frames*. A mapping is maintained between virtual memory pages and their location in main and secondary memory. Whenever the processor reads from or writes to a virtual address, the address must be mapped onto a location in main memory so that the access can occur. If such a mapping is not possible, then the virtual memory page containing the address does not exist in the main memory. This condition is called a *page fault*.

When a page fault occurs, the operating system performs the following tasks.

(1) The process that generated the faulting address is suspended until the fault is resolved.

(2) The required virtual memory page is moved from the secondary store into the main store. This is called *page retrieval.*

(3) When the page has been brought into the main memory, the mapping data is updated.

(4) Since access to addresses in the page can now occur, the suspended process is reactivated and the failing instruction repeated.

Eventually the main memory will fill up with retrieved pages. To provide room for further pages, it is necessary to remove others from main memory. This is called *page discard* or *page replacement.* Algorithms for page discard are well understood, and some common algorithms are described in [107]. The processes of page retrieval and page discard are collectively called *paging.*

Multiprogrammed computers support multiple processes running concurrently. These processes are allocated slices of processor time by software called the *process scheduler.* Each process in a typical multiprogrammed computer runs in its own virtual memory space, which contains the process stack, code, and temporary data structures. The virtual addresses used by each process are the same, but the data seen at any particular virtual address may be different for each process. Such architectures require separate mapping data for each process. Switching between mapping data occurs every time the process scheduler suspends a process and activates another. In this way a particular virtual address will map to a different place in main memory (or may not exist in main memory) for different processes.

The first machine to implement a paged virtual memory was the Atlas [70]. The Atlas maintained mapping data in an *associative memory.* Associative memory provides a *key field* for each data record stored in it. A key may be presented to the associative memory and it is (logically) compared in parallel with all keys in the store, so that the data corresponding to a key value can be found very quickly. Atlas used an associative store entry for each page frame of main memory, keyed on virtual page number. A page fault occurred if no entry existed for the required virtual page. This process is demonstrated in figure 4.1.

The complexity of the hardware required to allow parallel checking of key fields means that large associative stores are prohibitively expensive to implement. The Atlas physical memory was quite small, and only one set of mapping data was required because the computer was not designed for multiprogramming, so mapping based on an associative store was appropriate. As main memory sizes grew, however, it became impractical to build large enough associative stores, and a different technique based on *page tables* was adopted.

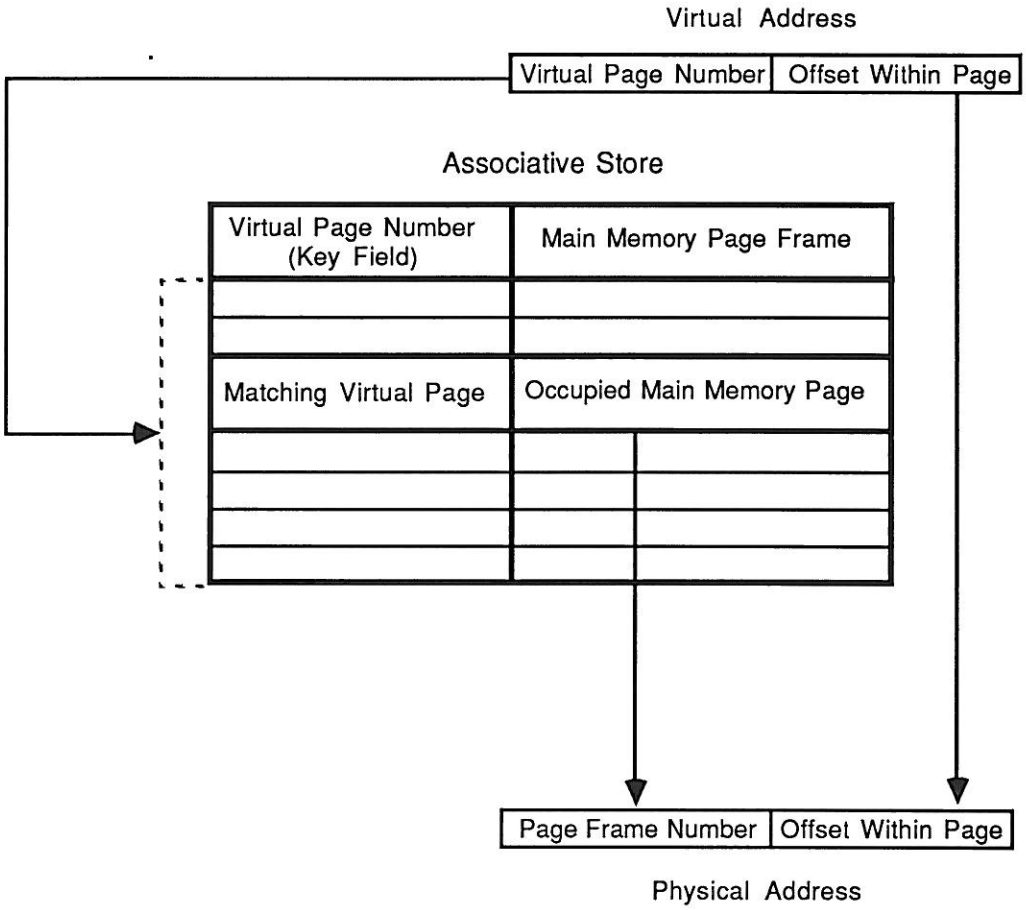


Figure 4.1. Use of Associative Memory in Address Translation.

A page table is a linear list, indexed by virtual page number. Each entry describes the current status and location of the corresponding virtual page. The length of a page table is thus proportional to the number of virtual pages. To support multiprogramming and to provide protection between processes it is usual to have a page table per process with a page table switch occurring as part of a process switch. This is implemented by having a page table register pointing at the current page table[1].

A typical page table is shown in figure 4.2. The location field of each page table record contains the main memory page frame number if the virtual page is in main memory, and the disk address of the virtual page if it is not currently in main memory. The present bit indicates

[1] There is usually also a page table length register so that page tables need not be the maximum length.

whether the page currently exists in main memory. The status field contains access information such as read, write, or execute access rights, and information used by the page discard algorithm such as how recently the page has been used and whether it has been modified. When a process generates a virtual memory address, the virtual page number within the address is used as an index into the page table. If the present bit of the corresponding entry indicates that the page is in main memory, the offset of the address within the virtual page is appended to the main memory page frame number obtained from the page table. The resultant main memory address is then accessed (subject to the access rights indicated in the status field). If the virtual page is not in main memory, a page fault occurs. In this case the disk address information is used to retrieve the page from disk, and after the page table data has been updated the faulting instruction is repeated.

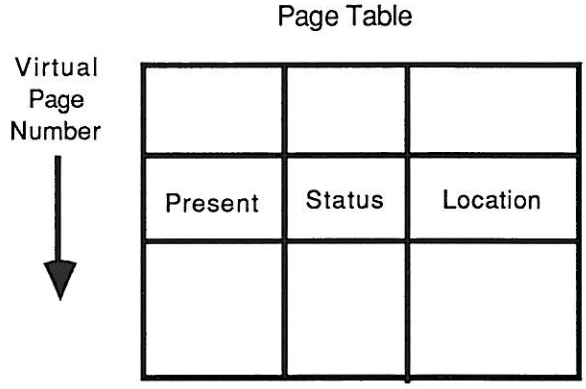


Figure 4.2. The Structure of a Typical Page Table Entry.

Disk addresses of pages currently in main memory are typically kept in a separate table called the *frame table* which is indexed on main memory page number. When page discard of a modified main memory page frame occurs, the frame table is used to determine where the frame contents are stored on disk. Unmodified page frames may, of course, be discarded without the need for an update of the corresponding disk page.

In principle it is necessary to consult the page table on each memory access to map between the virtual memory address generated by the running process and the corresponding main memory address. The

extra memory access involved would be detrimental to system performance, and to decrease this overhead, a high speed translation lookaside buffer is commonly used. This buffer contains the most recently accessed address translation entries, and relies on the principle of *locality of reference*, which states that a process references only a small fraction of its pages during any phase of execution. The set of pages necessary in main memory to enable efficient execution is called the *working set* for the process [37, 38]. Translation lookaside buffers are usually implemented as *set associative caches* [108].

For small virtual memories the page table, which contains an entry for every virtual memory page, is typically small enough to be kept in main memory. Large virtual memories require large page tables. For example, the VAX-11 [73], which has a virtual memory size of up to 2^{32} bytes[2] and a page size of 2^9 bytes, requires a page table with 2^{23} entries for a maximum size address space, each of 2^2 bytes. The maximum size page table is thus 2^{25} bytes or 32 megabytes long. Considering that such a page table is potentially needed for every process, it is not feasible to hold the page tables in main memory.

The solution is to place page tables in virtual memory. A result of this is that parts of the page table itself can be paged out, meaning that page faults can occur reading the page table. In order to ensure that such page faults can be resolved, it is necessary to lock some information about the page table into main memory. This can be achieved by recursively building page tables of the pages used to hold the next lower level page table until the result is small enough to lock into main memory. The VAX-11, for example, requires at most 2^{16} pages for its page table, as described above. In that case the second level page table would be 2^{18} bytes or 256 kilobytes in size, which is small enough to be locked down into main memory. Computers with larger virtual memories may need to increase the page size and/or use more levels of page table to achieve locked-down page tables of similar size.

The use of a multi-level page table means that a virtual memory access can involve an extra memory access, with an associated risk of a page

---

[2] In fact the VAX-11 has three virtual address spaces of up to 2^{30} bytes. We have simplified the description for the sake of clarity.

fault, for each extra level of page table. On the VAX-11, for instance, every virtual memory access must be eventually mapped onto a main memory address. If the page table entry corresponding to an address is not entered in the translation lookaside buffer, then the page table must be consulted. If the entry corresponding to the relevant page table page is not entered in the translation lookaside buffer, then the secondary page table must be used to find the main memory location of the page table page. If the secondary page table page is not in main memory, a page fault occurs, and the page must be brought in off disk. Paging in part of the page table may in turn necessitate the paging out of some other page. This involves updating the page table to reflect the removed page's new status, and this update may itself result in a page fault, and so on.

The above examination of an example of conventional virtual memory management demonstrates the complications introduced by the multi-level page tables required by large virtual memories. The fundamental problem is that the length of the page tables is proportional to the size of the *virtual* memory. We now examine an alternate approach in which the length of the table mapping virtual addresses to main memory addresses is proportional to the size of *main* memory.

4.3 Management of Large Virtual Memories

Two mappings are required in order to implement virtual memory. The first is a mapping from virtual addresses to main memory addresses for pages currently in main memory, and the second is a mapping from virtual addresses to disk addresses for pages not in main memory. The conventional approach to virtual memory uses the same data structures and mechanisms, based on page tables, for both of these mappings. Abramson [2] and others [4, 24] have proposed that different mechanisms and structures be applied to the relatively static disk address information, which is only needed in page fault resolution, and to the volatile main memory address information which is needed for every memory reference. The way to achieve this is to decouple the virtual address to main memory address mapping from the virtual address to disk address mapping. The Mach operating system [4], for

example, implements such decoupled mappings for the purpose of achieving portability between architectures.

Abramson's scheme, which was implemented in the MONADS-PC [100], uses special purpose hardware for translating the virtual addresses generated by programs. Unlike conventional lookaside buffers, which can only hold a subset of the address translation entries, the special purpose translator maintains virtual page to main memory mappings for *all* main memory page frames in high speed memory [2]. Thus page tables are not needed for the translation of virtual addresses to main memory addresses, and a translation miss only occurs if the required page is not in main memory.

When a page is discarded, the address translation entry for the appropriate main memory page frame is simply marked as invalid. This scheme avoids the problem of nested page faults experienced by conventional schemes because it is not necessary to modify a page table on page discard.

When a page fault occurs, a separate mechanism is invoked to find the disk location of the required virtual page. In keeping with the relatively slow access speeds of disks and the infrequency of page faults relative to memory accesses, this mechanism need not be so finely tuned for maximum performance, and can, for instance, be implemented without the use of expensive high speed memory. The scheme used in the MONADS system is based on disk page tables which are similar to the page tables described in the previous section, but which are only used for the virtual address to disk address mappings. This is described in section 4.3.2.

To improve the efficiency of page discard, the page fault manager maintains an additional data structure called the *Main Memory Table* (MMT). This table, which is locked into main memory, contains an entry for every main memory page frame. Each entry indicates either the disk address of the virtual memory page currently occupying the frame, or that the frame is unoccupied. The format of the table is shown in figure 4.3. When a modified page is removed from main memory, it is written to the disk page indicated in the MMT. This avoids reading the disk page table, which could result in a further page

fault. The MMT is also used to maintain a free list of main memory page frames. The lock count field is used to lock pages into main memory, thus preventing their discard.

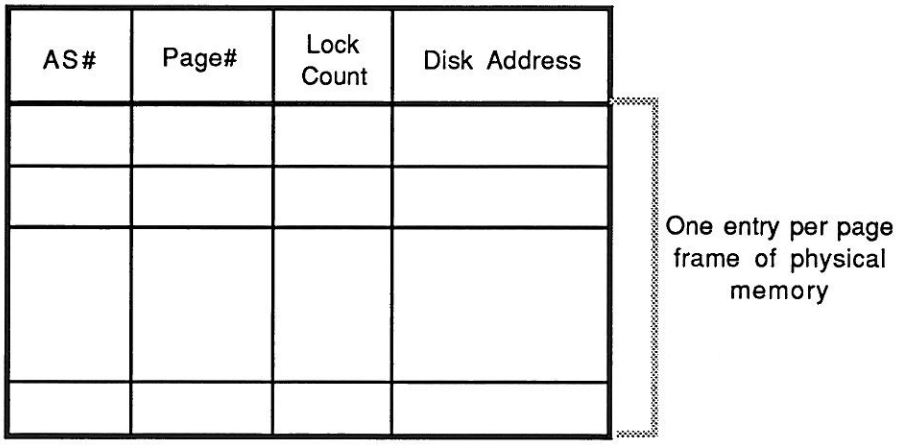


Figure 4.3. The Structure of the Main Memory Table.

4.3.1 Address Translation Hardware

Abramson's address translation hardware implements a hash table with embedded overflow for storing virtual address to main memory address mappings. Similar schemes have been adopted on the IBM System/38 which used a single hash table in main memory assisted by a translation lookaside buffer [15], and the MU6-G which used multiple hash tables queried in parallel [43]. To improve performance, Abramson's scheme maintains the hash table entirely in dedicated high speed memory.

Each cell of the hash table has the structure shown in figure 4.4. The key field identifies the virtual page represented by the cell, whilst the main memory page frame number indicates where in main memory the virtual page is located. Each cell also has a *foreign bit*, an *end of chain bit*, *modified bit*, *access rights bits*, and a *link field*. The purposes of these is explained in the following paragraphs.

When presented with a virtual address, the hardware hashes the virtual page number to the address of a cell in the table, and compares the virtual page number with the key. If they match, the page frame is retrieved and the access can proceed.

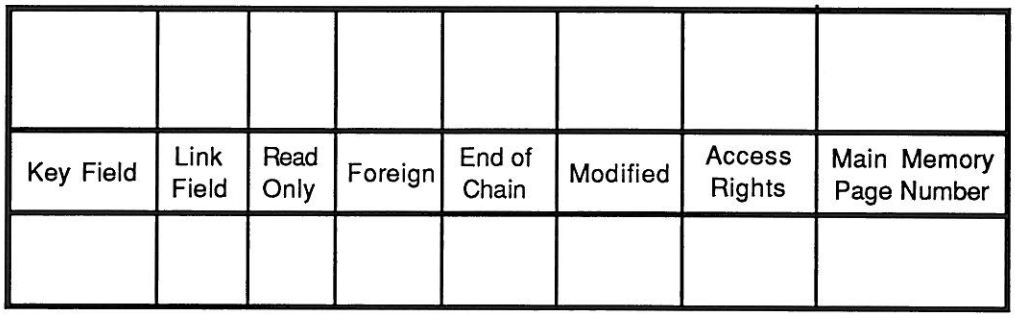


Figure 4.4. The Structure of a Typical Hash Table Cell.

It is possible for more than one virtual page number to hash to the same cell. This situation is called a *clash.* When a clash occurs, unoccupied cells are used to form a chain of entries with the same hash value. The link field in each cell in the chain points to the next cell in the chain. The last cell in a chain is indicated by the end of chain bit. Each cell in a chain, except for the cell at the head, has its foreign bit set to indicate that its virtual page number does not hash to the occupied cell.

The address translation process may be summarised as follows. The presented virtual address page number is hashed, and if the resultant cell contains a valid entry, the key field value is compared with the address. If these do not match, the chain is followed until either a match is found, in which case the translation is complete, or end of chain is reached, in which case the virtual page is not in main memory. If the virtual page is not in main memory a page fault is generated.

The read-only bit prevents writing to a page so marked. If an attempt is made to reference a page in contravention of the read-only bit, a *write fault* interrupt is generated and the reference is aborted. If an address in a page is successfully written to, the corresponding modified bit is set by the hardware.

This scheme leads to the possibility that a virtual page number V_1 may hash to a cell C containing an entry, for some virtual page V_2, which is part of a chain but not the head of the chain. This is indicated by the foreign bit, which, when set, indicates that the cell contains an entry for a page whose address hashes to the cell at the head of the chain, and so no entry could possibly exist in the chain for page V_1. When this

situation occurs, a page fault is generated for page V_1, and the entry for page V_2 is moved to an unoccupied cell to free the original cell C for an entry for page V_1. Since the entry for V_2 is part of a chain, the link pointing to the moved entry must be changed. The last cell in each chain has its link field pointing to the head to facilitate this operation. When page V_1 is brought into memory it is mapped using cell C. The address translation process is shown pictorially in figure 4.5.

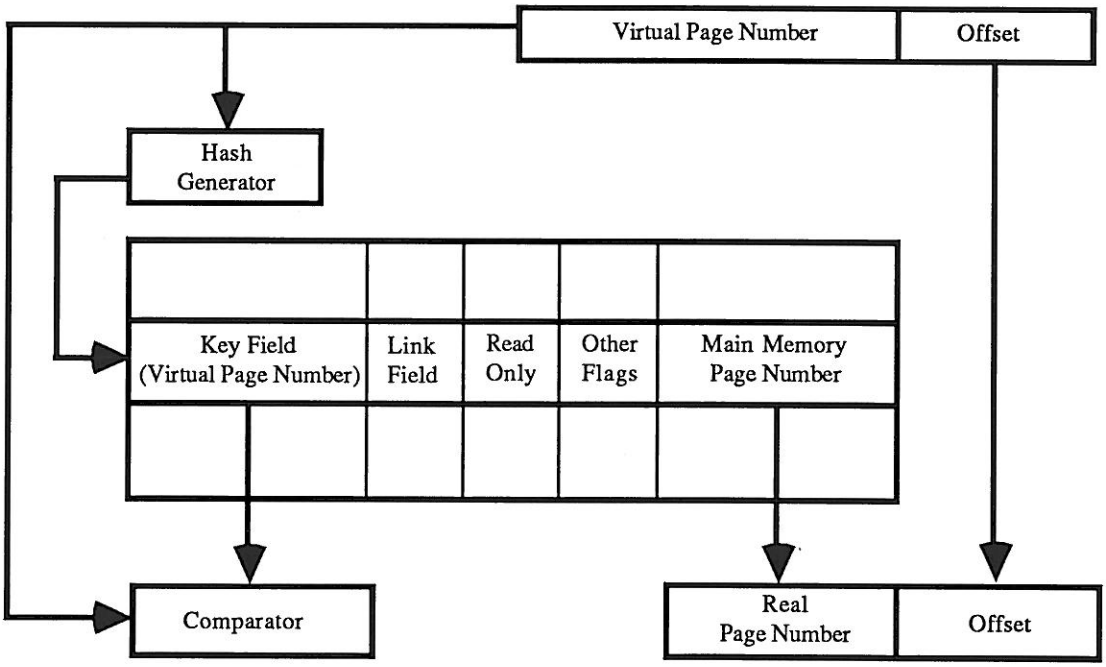


Figure 4.5. The Address Translation Process.

As described above, clashes on a cell require following of a chain on every access. This represents an overhead which must be minimised in order to achieve acceptable performance. Such performance can be achieved if the hash table is sparsely occupied, meaning that the probability of clashes and the lengths of chains are kept small. It has been shown [80] that if the hashing function provides an even distribution of hash keys, the expected number of probes to retrieve an item from a hash table is given by

$$E = 1 + (\alpha / 2)$$

where α is the *loading factor*, the ratio of occupied cells to the total table size. Abramson's implementation in the MONADS-PC uses a hash

table that has four times as many cells as there are pages of main memory. Thus the loading factor is 0.25, and the expected number of probes per address translation is 1.125. Notice that the number of cells in the hash table increases *linearly* with the size of main memory. Since the number of virtual pages is doubled by widening addresses one bit, the hash table width increases *logarithmically* with the size of virtual memory. This is in contrast to the page table approach, in which page tables increase linearly with the size of virtual memory.

We have shown that the described address translation scheme is efficient provided the hash table is of sufficient size, and that it is scaleable to large virtual memories. The experimental MONADS-PC system, for instance, supports 60 bit virtual addresses, and the new MONADS-MM system [103] supports 128 bit addresses. In the MONADS system all processes operate within the one large virtual address space. A single address translation table is thus sufficient for all processes running on a MONADS computer. This is in contrast with conventional virtual memory implementations that reuse virtual addresses for each process (excluding lightweight processes), and thus maintain a separate page table per process.

The MONADS-PC address translation unit (ATU) is implemented using the techniques described above. It provides mapping between virtual and main memory addresses, and generates a page fault interrupt if the page containing the required address is not in main memory. The following section describes how MONADS secondary storage is structured, and how page faults are resolved.

4.3.2 Secondary Storage Management

In the MONADS-PC system the virtual memory encompasses all disk and main memory. Thus in MONADS, virtual addresses can refer to any byte on any disk connected to the computer. In order to resolve a page fault, the disk location of the page must be determined. The first step is determining which of the attached disks contains the page.

Modern disk drives can have very large capacities. It is often convenient to split up the capacity of such a drive into smaller units called *partitions* which are logically equivalent to separate disks. A

partition, of course, can encompass an entire drive if required. The logical disks formed by partitioning a drive attached to a MONADS computer are known as *volumes*[3]. When a volume is created, it is allocated a unique *volume number*.

The volume number can be used to define the range of virtual addresses that is stored on the volume. To achieve this we use the volume number as the high order bits of all such addresses. MONADS virtual addresses are wide enough to allow a volume number field that does not limit the number (ie there are enough volume number bits) or size (ie there are enough lower order bits) of disks. This avoids complex mappings to maintain the association between virtual addresses and volume numbers[4]. The MONADS kernel maintains a *Local Mount Table* of currently mounted volumes. Each entry in this table consists of a volume number field and a disk drive number field.

Each volume, then, has its own range of virtual addresses. This range is further divided into areas corresponding to the logical entities such as processes, files and programs that exist on the disk. These areas are called *address spaces*, and are identified by address space numbers. Address spaces are further divided into fixed size pages, identified by page numbers. A MONADS virtual address, then, consists of four parts, as shown in figure 4.6. On the prototype MONADS-PC system, addresses are 60 bits wide, with 6 bit volume numbers, 28 bit within volume address space numbers, 16 bit within address space page numbers and 12 bit offsets. The MONADS-MM system [103], with 128 bit addresses, allows for 32 bit volume and address space numbers.

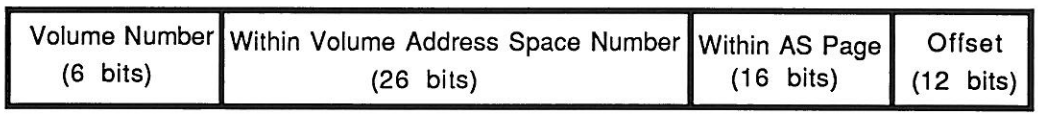


Figure 4.6. The Structure of a MONADS-PC Virtual Address.

Each address space has its own page table which maps from virtual addresses to disk addresses for that address space. This page table is

---

[3] MONADS-PC volumes are of maximum size 256 megabytes.

[4] The moving of objects between volumes will be discussed in chapter 6.

stored within the address space which it describes. Because the address translation hardware does not need to access this table, its format need not be permanently fixed, and there may be several different formats. The format for any particular address space must, of course, be known by the virtual memory software to enable it to control paging. For example, in the case of a large database, the pages may be stored in contiguous disk blocks. In such a case the page table need only consist of a starting disk address along with the number of pages, since the disk address of any page may be computed using the start address and the page number. This flexibility is not possible with conventional systems.

Address spaces for which disk block allocation is completely dynamic require a separate entry for each page. The size of such a page table depends on the number of pages in the address space and on the page size. The largest MONADS-PC address space is 2^{28} bytes (256 megabytes) and the page size is 2^{12} bytes (4 kilobytes). The page table for the largest possible address space requires 2^{16} entries (one for each page), and this is placed in virtual memory. Entries in the page table contain disk location information only, because status data is only required for active pages and can be maintained in the ATU. This means that each entry need only consist of a 16 bit relative disk block number since the maximum size of a volume is 256 megabytes. For the largest possible address space, with a 16 bit (2 byte) entry for each of 2^{16} pages, the page table is 2^{17} bytes or 128 kilobytes.

The address space page table, which we call the *primary* page table, therefore potentially occupies 32 pages of the address space. Since it is of unknown size, the primary page table is located at the high-order address end of the address space, growing downwards towards the data.

To enable the pages of the primary page table to be found on disk, a *secondary* page table is required. It is stored in the first page of the address space. The system maintains a table of the address spaces stored on a volume, together with the disk location of the first page of each address space, in a well-known location on the volume. These *volume directories* are described later in this section. The secondary

page table, like the primary page table it describes, exists in virtual memory and consists of 32 entries of 16 bits each, a total of 64 bytes.

It is recognised that many address spaces contain only a small amount of data. To reduce the number of disk accesses needed to read such small address spaces, we store the first 256 entries of the primary page table in the first page of the address space. This means that it is not necessary to read *two* pages (page zero for the secondary page table, and the top page for the primary page table) to access the first 256 pages of an address space. Thus, for address spaces of up to 256 pages (or one megabyte), the complete primary page table is stored in the first page.

The first page of an address space thus contains the 32 entries of the secondary page table and 256 entries of the primary page table, a total of 576 bytes. This leaves space for approximately 3.5 kilobytes of data in the first page of an address space, meaning that for address spaces smaller that 3.5 kilobytes, a page fault is resolved in the course of accessing the address space paging information. The structure of an address space is illustrated in figure 4.7.

The consistent structure of the address spaces described above allows the page fault handler to calculate the location of any primary page table entry. When a page fault occurs, the page fault handler attempts to read the page table entry for the page. The virtual address of this page table entry may be calculated since the virtual address of the start of the page table is known. This may in turn cause another page fault if the required page of the primary page table is not in memory. If this second page fault does occur, the secondary page table entry for the primary page table page must be read to determine the disk location of the primary page table page. Again this address is easily generated since the virtual address of the secondary page table within page zero of the address space is known. This read could cause a third page fault on page zero of the address space. To resolve this page fault, the volume directory must be accessed. This volume directory is stored in address space zero of every volume.

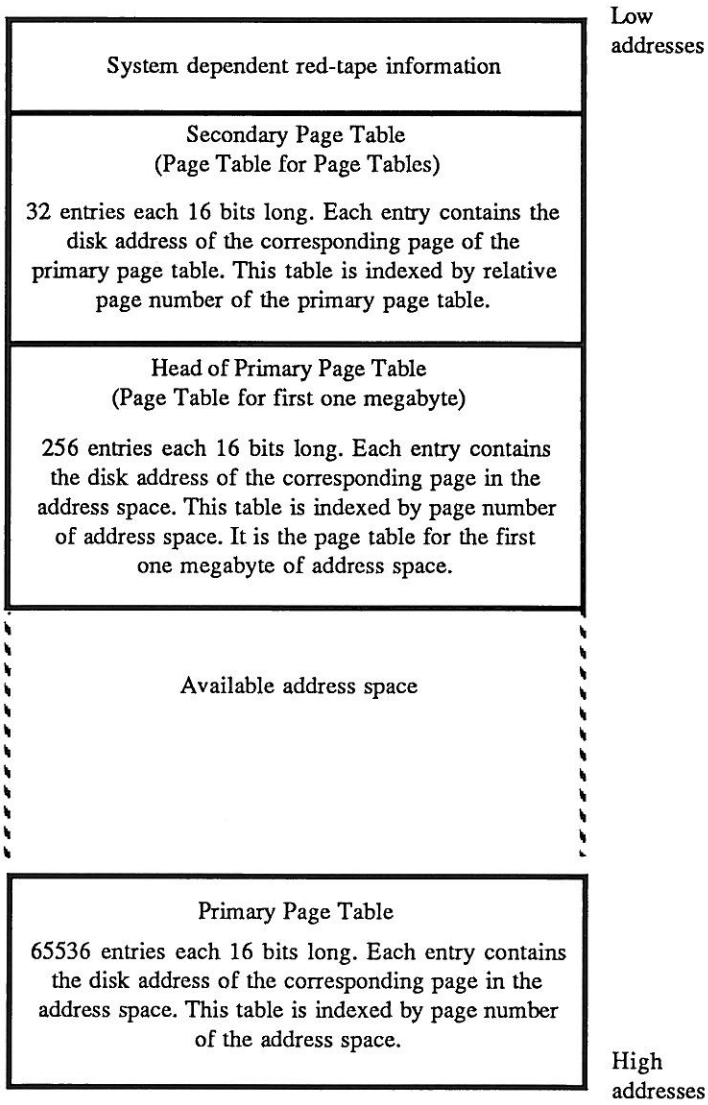


Figure 4.7. The Structure of an Address Space.

Address space zero of a volume is used to store volume specific information such as a free space map and a directory of the address spaces stored on the volume. Whenever a new address space is created on a volume, it is assigned the next available relative address space number for the volume, which guarantees that every address space number will be unique. The directory is organised as a hash table keyed on these relative address space numbers. This implementation is used because address space numbers are sparsely distributed and large. The organisation of address space zero is illustrated in figure 4.8.

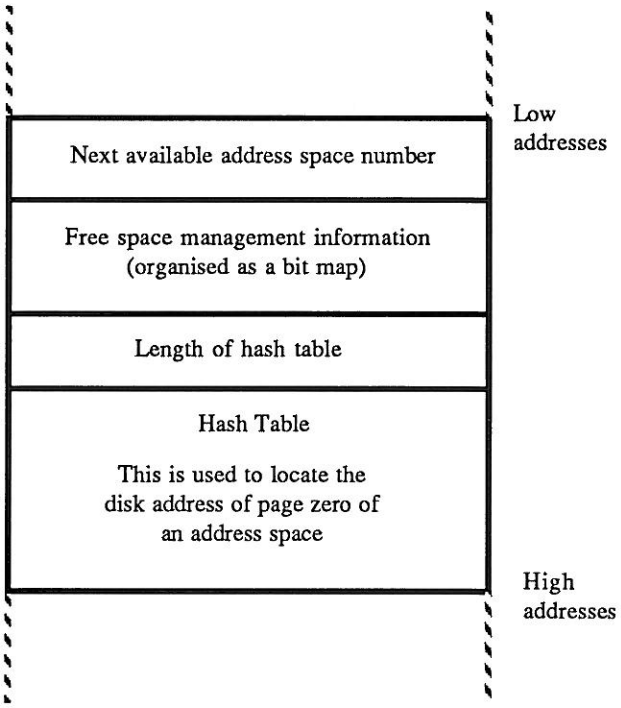


Figure 4.8. The Organisation of Address Space Zero.

The structure of address space zero is identical to all other address spaces in that it contains primary and secondary page tables as previously described. It is necessary, however, to predefine the disk location of the first page of address space zero to enable the secondary page table for the address space to be read. The kernel reads this first page of address space zero for a volume, and locks it into memory, when the volume is mounted.

4.3.3 Evaluation

The separation of virtual to main memory mappings from virtual to disk location mappings results in greater flexibility than provided by conventional virtual memory management systems. Most significant of these is the ability to manage extremely large *virtual* memories with only a *logarithmic* increase in the size of the virtual to main memory

address mapping data. Coupled to this is the ability to manage large *main* memories with only a *linear* increase in this mapping data[5].

A side benefit of the conventional technique of maintaining a separate virtual memory space for every process or group of cooperating processes is that processes in different virtual memory spaces cannot illegally or accidentally access each other's data. Since MONADS processes all exist in the same virtual memory space, a different method must be used to protect data. This is described in the next section.

4.4 Higher Level Memory Management

The use of very large virtual memory negates the need for re-use of virtual memory addresses, as required by conventional systems, meaning that all processes and data can co-exist in a single virtual memory space. The use of separate virtual memory spaces for each process or group of co-operating processes in conventional systems has the advantage that data in such a memory space is automatically protected from illegal access by external processes. This means of protection is not available within a single large virtual memory space. On the other hand, it is not necessary to use contorted schemes for the sharing of data between processes. An example of such a scheme is the mapping of data out of one virtual space and the subsequent mapping into another used by Amoeba [84].

The approach to security taken in some experimental systems such as Napier88 [82] is to restrict code generation to trusted programs such as compilers, thus ensuring that programs cannot generate illegal addresses. This has some merit in that it shifts most of the protection overheads to compile time. However there are major disadvantages in terms of flexibility, particularly in a mixed language environment.

An alternative scheme, adopted in MONADS, is to use a higher level architecture based on division of the virtual memory space into logical units called *segments*. Access to these segments, and the structures

---

[5] The Main Memory Table, which is used to store the disk locations of virtual pages currently in main memory, also increases linearly with the size of main memory.

built with them, is controlled using protection based on *capabilities* [39, 47, 64, 100]. Support for capabilities is provided at the architectural level.

4.4.1 Segmented Virtual Memory

Programmers and compilers see the MONADS virtual memory as a collection of *segments* which may be of arbitrary size[6]. Segment boundaries are orthogonal to page boundaries [61, 63] thus avoiding the internal fragmentation problems associated with most combined paging and segmentation schemes [86, 93]. All segments have the same basic format, allowing access to them to be handled in a uniform manner [102]. Segments contain data and capabilities for other segments, allowing arbitrarily complex graph structures of segments to be constructed. The architecture ensures that capabilities cannot be arbitrarily modified.

Processes never directly use virtual addresses, rather they use offsets relative to a segment [102]. Segments are addressed using *segment capabilities*. These consist of three fields, the *start address*, the *length*, and the *type and access*, as shown in figure 4.9. The type field defines the type of the data contained in the segment, and the access field defines the allowable operations on the data. The use of segment capabilities in addressing the virtual memory is illustrated in figure 4.10.

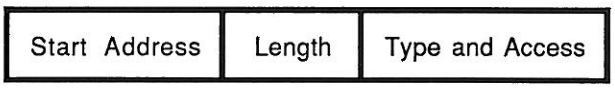


Figure 4.9. The Structure of a Segment Capability.

To improve efficiency of access to segments, the architecture provides a set of special purpose registers called *capability registers*. To enable access to the data stored in a segment, the appropriate segment capability must be loaded into a capability register. Machine instructions then access the data using addresses of the form:

---

[6] Up to the size of an address space, as described later.

<capability register number><offset within segment>.

Any attempt to access beyond the bounds defined by the length field, or in conflict with the rights defined in the type and access field will cause an exception condition.

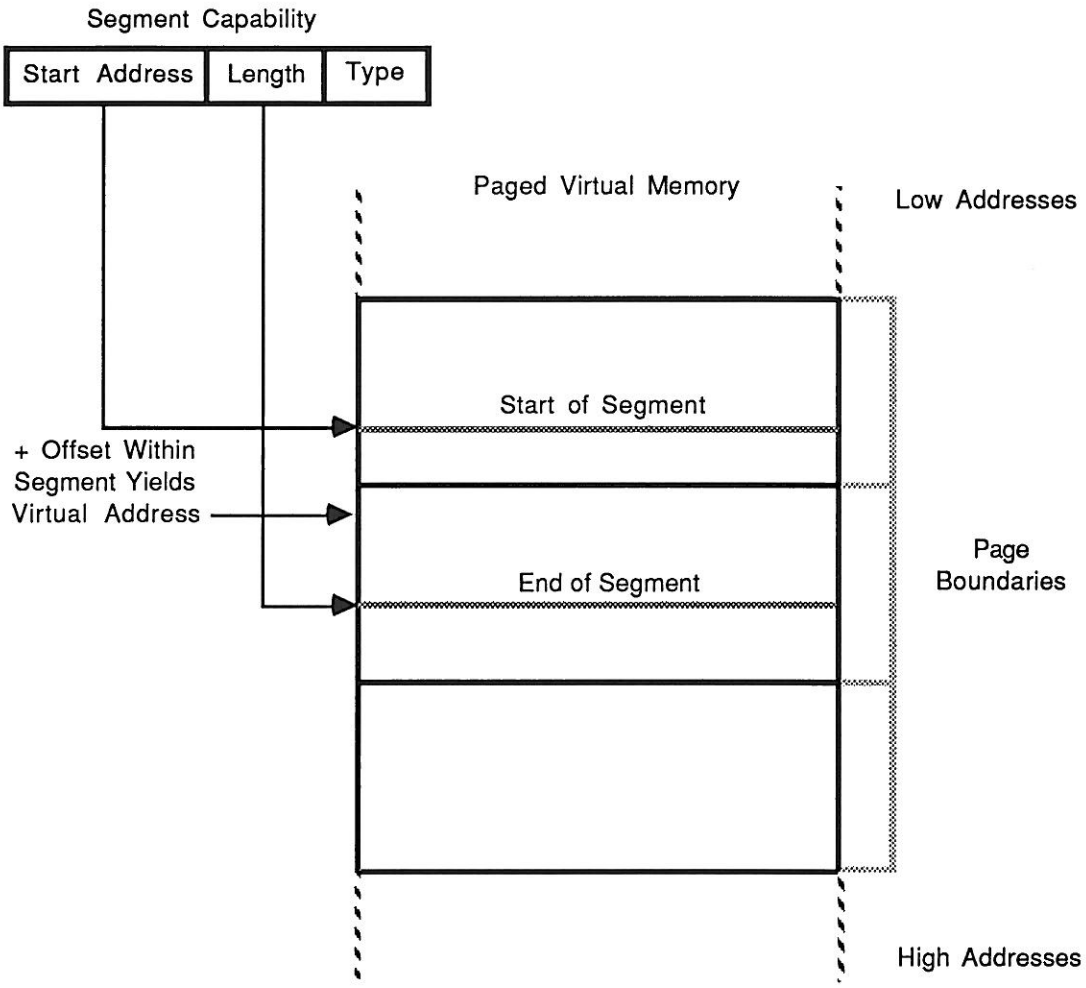


Figure 4.10. Addressing Using Segment Capabilities.

As mentioned, segment capabilities may be stored within a segment. Such storage occurs in a segregated section of the segment, and is achieved using a special machine instruction. A special machine instruction is also used to load capability registers. The instruction will only load a capability register with a segment capability that is stored in the segregated region of another segment, and that segment must be pointed to by another capability register. It is thus possible to securely traverse arbitrary data structures.

Segments are grouped together into information-hiding *modules* [87, 88]. Each module presents a purely procedural interface. Access to modules is controlled by a *module capability*. A module capability consists of three fields, the *module name*, *access list*, and *status*, as shown in figure 4.11. The access list defines the interface procedures useable by the owner of the capability, and the status field defines the operations that can be performed on the capability, such as whether it can be copied. Module capabilities may only be stored in the data area of segments of type "module capability". Such segments have the type and access fields appropriately set to prevent illegal modification of the stored module capabilities.

Figure 4.11. The Structure of a Module Capability.

The first time that a process accesses a module, it uses the *open* system call, providing the module capability as a parameter. A special segment called a *module call segment* (MCS) is created containing, amongst other things, segment capabilities for each of the interface procedures available to the owner of the module capability[7] and for the data encapsulated in the module. The module call segment is used for subsequent calls on the module until the module is closed, and improves the efficiency of such calls [67]. To create an MCS the calling process must access the module's *red tape* information. This is contained in the root page for the module. Included in the red tape is a count of processes which currently have the module open. This count is incremented whenever a process opens the module, and decremented when the module is closed. The system will only allow a user to delete a module if the user's module capability has the appropriate access rights, and if the open count for the module is zero.

As described earlier, a capability register can only be loaded from a segment pointed to by another capability register. This means that

---

[7] This is in a *logical* sense. In fact the MCS contains a pointer to the set of capabilities for the module interface.

there must be some starting position from which capability registers may be loaded. Every process has a special segment called its *base table* which is the root of addressing for the process. The system sets up the base table, and maintains it, for example by modifying its contents to change protection domains on inter-module calls.

Each MONADS module resides in a separate and unique address space. Such address spaces are never re-used, even if the module is deleted. The maximum aggregate size of the segments of a MONADS module is thus 256 megabytes. The module name in a module capability is simply the address space number of the address space occupied by the module. Such a name is guaranteed to be unique because address spaces are never re-used. An attempt to access a deleted module results in an *invalid module* exception.

There are two distinct types of address space

(1) those which contain *module*s, and

(2) those which contain *stack*s.

The major difference between these is their handling of pointers. Stack address spaces may contain pointers into other address spaces, allowing, for instance, segments on a stack to contain pointers to code and data segments in other modules. Such usage requires that all pointers held in a stack address space are *long pointers* containing volume number and within volume address space number information.

Module address spaces and are not permitted to contain pointers into other address spaces. Thus pointers stored in a module address space point to segments within the address space itself. These pointers are *short pointers*, and do not contain volume number and within volume address space number information. This is because these values are the same as for the address space containing the pointers (i.e. the module address space). Such pointers are in effect simply offsets relative to the address space that contains them.

4.5 Conclusion

Virtual memory management techniques provide the abstraction of larger memories than are actually physically present. These techniques permit the running of programs whose code size exceeds the main memory size, and the running of multiple processes when the total size of the processes and their data is greater than the main memory size. Virtual memory management is based on the division of the memory space into blocks called pages, and the division of main memory into page frames of the same size. Hardware and software is used to transfer pages of virtual memory into and out of main memory as required.

The traditional approach to the management of such paging is to use a page table for the virtual memory space. This table contains mapping data enabling the location of pages both on disk and in main memory. It has been shown that the use of such conventional page tables restricts the possible size of the virtual memory space.

Much larger virtual memories may be efficiently managed if the conventional page table function is split, and virtual memory disk location mappings are separated from virtual to main memory mappings. This technique has been used to implement 60 bit wide virtual addresses on the experimental MONADS-PC, and will be used to implement 128 bit wide virtual addresses on the proposed MONADS-MM.

The use of such large virtual memories precludes the need for re-use of virtual addresses. This is in contrast to conventional virtual memory machines in which virtual addresses are re-used for every process address space. The problem of security of access then arises, because every process exists in the same virtual address space, namely the complete virtual memory space. Security may be provided using capabilities.

The MONADS implementation of capabilities is based on the division of the memory space into logical segments, with the segment boundaries orthogonal to page boundaries, and the grouping of segments into information-hiding modules with purely procedural interfaces. Such modules, which coincide with partitions of the virtual memory called

address spaces, are sparsely distributed throughout the virtual address space. Access to modules is controlled by module capabilities, and access to the segments within a module is controlled by segment capabilities. A module capability allows access to interface procedures of the module by permitting the use of the segment capabilities needed to access the code segments that implement the interface.

Chapter 5 THE MONADS DISTRIBUTED SHARED MEMORY MODEL

5.0 Introduction

The MONADS Distributed Shared Memory (DSM) model is based on a single very large virtual memory space which encompasses all nodes in the network. Processes running on these nodes have access to the *whole* virtual memory space (subject to protection considerations), without the need for knowledge of the storage location of the program code and data they access. The model was initially proposed in [3], and extended by the author in [19, 54, 55, 56]. Related schemes have been reported in the literature [36, 74]. However these schemes allow processes to share only a limited portion of their total address space, and still maintain a separate file store.

We have substantially modified and extended the Abramson and Keedy model [3] in our work on DSM [19, 54, 55, 56]. These extensions and modifications have resulted in a more durable and efficient network. In this chapter we describe our model for a distributed shared memory. Initially we make the assumption that modules are never moved from the node on which they are created. We also assume that a message transmitted on the network is always correctly received, and that no message is ever lost or corrupted in transmission. In chapter 6 we relax some of these restrictions and describe how modules and volumes may be moved. In chapter 7 we discuss the issues of network reliability, node shutdown, stability, and how the DSM is made stable. In chapter 8 we discuss implementation issues.

5.1 The Network

The proposed network is designed to support the interconnection of *homogeneous* machines using a Local Area Network (LAN). We do not make any assumptions about the topology of the LAN, merely that it must allow processors to be able to communicate with each other to exchange both data and protocol messages. Topologies that satisfy this criterion, making them suitable for use with our proposed network, include *bus* (e.g. Ethernet [40, 79]) and *ring* (e.g. Token Ring [48] and

Cambridge Ring [85]). A minimum configuration node consists of a processor and some private memory. Nodes may also have local disks, as well as other peripherals such as printers, tape units, and terminals.

Nodes communicate by transmitting messages on the network. These messages, which are listed in appendix 1, are used to implement the up/down protocol, the address resolution protocol, the shared memory protocol, and the data coherence protocol as described in the balance of this chapter.

The DSM is formed by extending the paged virtual memory addressed by each node to a single network-wide paged virtual memory. Pages, which are of fixed size, form the unit of transfer between nodes. Each processor has its own Address Translation Unit (ATU), as described in chapter 4. This ATU is essential for virtual memory management at each node because it provides a mapping between the virtual memory pages and the page frames of the node's main memory. If a mapping does not exist for a given virtual address, then the corresponding page is not currently in the node's main memory and the ATU raises a page fault interrupt. When such an interrupt is raised, the page fault handling software is activated. The ATU supports the mapping of pages as read-only or read-write, and maintains a *modify bit* which indicates whether any byte in the page has been modified. An attempt to write to a read-only page results in a *write fault* exception.

Other DSM implementations use an essentially conventional page table in managing the shared virtual memory [36, 74]. As described in chapter 2, the page table is used for mapping the shared virtual pages to main memory page frames, for defining the storage location for shared pages not currently in main memory, and for status information used by the coherence algorithm. Before a node can obtain a copy of a shared memory page, it must know which node is acting as the server for the page. Since such information may only be efficiently obtained from the page table, a *full* shared memory page table is maintained at every node. The existence of this full page table enables each node to efficiently resolve local page faults, and is also used to implement shared memory coherence strategies. However it has major disadvantages in terms of scaleability.

As described in Chapter 4, the MONADS architecture partitions the virtual memory into regions with fixed maximum size. These regions are called *address spaces*. Each address space contains a table, called the *address space page table*, which maps the virtual pages of the address space to pages on a logical disk or *volume*. Thus separate disk page tables are maintained for every address space in the virtual memory. These disk page tables are completely independent of the ATU, which maps virtual addresses to main memory addresses. This separation of the page tables for main memories and the disk page tables means that the mapping from virtual pages to main memory page frames can be maintained independently for each node. Since it is necessary to maintain mappings only for those virtual pages that are currently in the node's main memory, the table's size is proportional to the size of the node's main memory. The virtual page to main memory page frame mappings for a node are held in the node's ATU.

As we show in this chapter, the single node MONADS architecture can be extended to enable efficient management of a DSM. This is achieved without the need for a single large shared memory page table. In contrast to both IVY [74] and Memnet [36], which require the maintenance of a complete virtual memory page table at each node, the MONADS scheme is thus scaleable to extremely large shared virtual memories. To facilitate understanding we have included an alternate *state transition* view of the model in appendix 2.

5.2 Addressing The Distributed Shared Memory

The addressing scheme described in chapter 4 provides a single-level store for a single MONADS-PC node. The approach taken in the network implementation is to extend this virtual store to encompass the entire network. This is achieved in a completely transparent fashion so that any byte within the entire DSM may be directly and uniquely addressed from any machine in the network. The ability to uniquely identify each of the networked nodes is crucial to our DSM addressing scheme. To this end, a unique *node number* is allocated to

each MONADS system at the time of its manufacture. This number is maintained in a dedicated read-only memory (ROM) in the node[1].

5.2.1 Node Numbers

As previously described, a MONADS address consists of four parts (see figure 4.7), with the high order bits defining the volume on which the address space containing the address is stored. For a network of MONADS-PC computers, we partition this volume number into a unique *node number* and *within node volume number*[2]. This means that a full virtual address now consists of a unique node number, a volume number within node, an address space number within volume, a page number within volume, and an offset within page. This structure is shown in figure 5.1. When a new address space is created it is allocated an address space number with the unique node number of the creator embedded within it. Thus every address space number, and therefore every module number, is unique network-wide. If the address space and/or the volume containing the address space is never moved, the address space number defines the node and volume on which the address space is stored.

The node at which an address space is stored is called the *owner node* for the virtual pages that comprise the address space. Owner nodes act as *page servers* for their pages in the resolution of page faults at other nodes.

---

[1] In the case of processor board replacement due to malfunction, the original node number is transferred to the replacement processor board by moving the appropriate ROM chip.

[2] For the modified MONADS-PC the node number is 2 bits and the within node volume number is 4 bits. The experimental network is restricted, then, to a maximum of 3 nodes plus a "broadcast node" (see section 5.2.3). The MONADS-MM has 32 bit node numbers and 32 bit within node volume numbers.

Figure 5.1. The Structure of a MONADS DSM Virtual Address.

5.2.2 Page Faults

A page fault interrupt occurs at a node because of an attempt to access an address in a page which is not in the node's main memory and therefore not mapped into the node's ATU. For a centralised MONADS computer, the page fault handling software is able to quickly locate the page table for the address space containing the page because it is maintained in a well-known location within the address space itself. As described in section 4.3.2, in the case when the page table itself is not in main memory, the volume number of the volume containing the page's address space is derived from the faulting address. Each volume contains a table of the address spaces stored on it, enabling the root page of the address space page table to be located. The disk location of any virtual memory page can be rapidly found from this root page, allowing the faulting page to be transferred into main memory and mapped into the ATU and the faulting instruction to be repeated.

When a page fault occurs on a networked machine, the kernel inspects the node number field of the virtual address to see whether the page containing the address is stored on a local disk. If the node number is the local one, then the local node is the owner node for the page and resolution of the page fault can occur as described above (subject to the issue of coherence discussed in section 5.3). If the node number is not the local one the page fault is considered to be a *remote page fault.*

5.2.2.1 Remote Page Faults

To resolve a remote page fault the local node *L* must send a message to the page's owner node *O*. This message requests node *O* to act as a page server and to transmit a copy of the page back to node *L*. The identity of the owner node *O* is the node number contained in the faulting

address itself. It is thus not necessary to reference a shared memory page table, as required by [36, 74], to determine the identity of the owner node.

So that it is able to transmit the requested page to node *L*, the kernel at node *O* must bring the page into its own main memory if the page is not already there. To do so (ignoring coherency issues), the kernel at node *O* attempts to read an address in the requested page. If the attempt succeeds, the page is currently in the main memory of node *O*. If not, a page fault occurs and the page is brought into main memory. When the page is in main memory at node *O*, it is mapped into the ATU as a read-only page and locked down, thus preventing it from being discarded as part of normal paging. A message containing the page as data is then transmitted to *L*. On completion of this transmission the page is unlocked, thus allowing its eventual discard.

When node *L* receives the page it is mapped into its ATU in the same way as after retrieval from local disk[3]. The waiting process(es) are then activated as for a local page fault.

In the above description we have assumed that, given the node number of the source and destination, it is possible for messages to be transmitted between nodes. This assumes the existence of a mapping between the node numbers used by the DSM protocols and the physical network addresses used in the actual transmission of messages between nodes. In the following section we describe how this mapping is achieved.

5.2.3 Node Address Resolution

The unique node number for a MONADS computer is defined in its hardware when it is manufactured. These node numbers are used to create a *virtual network* in the same way that IP addresses are used to form an internet [30]. The separation of node numbers from the physical network addresses of nodes means that we avoid making

---

[3] However, the page is mapped into the ATU as read-only whereas a page read from local disk would be mapped in as read/write. This is because of the coherency protocol (see section 5.3).

assumptions about the underlying network hardware or type. As a result we can safely embed node numbers in virtual addresses while still allowing flexibility in the choice of underlying network[4].

The use of proprietary networks such as Ethernet [40, 79] and Token Ring [48] requires that machines be allocated network addresses suitable for the network type. This means that translation is necessary between the virtual node numbers used at the logical level by the network protocols and the network specific addresses used at the physical level.

The translations from node numbers to network addresses and vice versa are achieved at each node using the local *Network Addresses Table* (NAT). This table is of the format shown in figure 5.2 and is maintained at each node by the local kernel. The protocol used to maintain NATs is similar to the ARP protocol used for binding between virtual TCP/IP addresses and physical network addresses [30].

Logical Node Number	Physical Network Address

Figure 5.2. The Structure of the Network Addresses Table.

When a node boots, it initialises the NAT table with its own node number to network address mapping, and with a mapping between the logical *broadcast node* number and the equivalent network broadcast address. A message transmitted to the broadcast virtual node is received by *all* nodes physically connected to the network. Most LAN architectures support such a broadcast address. For example, Ethernet uses the physical broadcast address FFFFFFFFFFFF [114].

---

[4] In the prototype implementation all machines are connected to an ethernet.

After initialising the NAT, the node broadcasts a *here_i_am* message with its node number and physical network address as parameters. On receipt of this message each of the other active nodes updates its local NAT, and replies with a *here_i_am_too* message containing its node number and physical network address as parameters. When a node is shut down, it broadcasts a *node_going_down* message, with the result that all other nodes remove the appropriate NAT entry.

It should be noted that a protocol similar to RARP [30] is not needed for diskless nodes in the MONADS network because each machine's logical node number is embedded in its hardware. This is in contrast to IP addresses, which are allocated by the local system manager, and thus cannot be stored between successive bootstraps of a diskless TCP/IP node.

5.3 Data Coherency

According to the described page fault resolution protocol, an owner node provides a copy of a page to any remote node on receipt of a request from that node. This means that any time there may be copies of a particular page P in the main memories of different nodes, including the memory of the owner node. For the multiple views of the data contained in page P to be considered *coherent*, each of the processors must see the same version of the page. If, for instance, each of the nodes with P in main memory have only *read* locations in the page since obtaining it from the memory server, then the copies are identical and each processor has a coherent view of P.

If one or more of the processors *writes* to locations in P, then the copies in the main memories are no longer identical, and so the processors do not have a coherent view of data in the page. The provision of a coherent view of shared data in a centralised system with shared memory is implicit. Coherence in the DSM could be achieved by higher level code that enforced some coherency algorithm. This would require that programmers include specialised code in programs to implement the algorithm. Since we aim to provide shared virtual memory whose distribution is *transparent* to the user (and thus the

programmer), such a solution to the memory coherency problem is not suitable.

The problem of maintaining a coherent view of the data stored in a page when multiple copies of the page can exist across the network is similar to that of *cache coherence* in multiprocessors [36, 71, 75] because, in effect, the main memory of each node is a local cache of virtual memory pages. Consequently similar techniques to those used to achieve cache coherence may be applied. In this section we describe the *multiple reader/single writer* protocol used to ensure data coherence. Protocol messages are only sent to nodes listed by the owner node as having copies of the page, requiring fewer messages than schemes such as "shares" [57], which broadcasts protocol messages, requiring the processing of messages by nodes with no interest in the subject page.

The protocol is based on hardware support (in the ATU) for *read-only* and *read/write* pages, as provided by the MONADS ATU (see chapter 4) and most commercial memory management units. The ATU also records whether a page has been *modified* since being mapped in. Such support provides two types of memory access fault and associated interrupt, the usual page fault, and a *write fault* which is raised if an attempt is made to write to a read-only page. The ATU may also be queried to check the modify status of a page.

5.3.1 Multiple Reader/Single Writer Protocol

As suggested by its name, this protocol allows for any number of simultaneous readers of any page, or exactly one writer to the page. Each node with disk(s) is potentially a memory server for the pages stored on its disk(s), and is responsible for ensuring that the multiple reader/single writer protocol is adhered to for these pages. To achieve this the kernel at every node maintains an *Exported Pages Table* (XPT). This table contains an entry for every owned page for which a copy currently exists in the main memory of another node. These entries consist of an *reading nodes* field, a *page number* field which identifies the page, a *disk page* field which defines where the page is stored on disk, and a *current version* field, as shown in figure 5.3. The reading nodes field, which is similar to the *copy set* used in IVY [74], contains

the node numbers of nodes with a read-only copy of the page in their main memory (excluding the node whose number appears in the current version field). The disk page field is used to create the MMT entry for a modified exported page when it is returned to the owner. The current version field contains the node number of the node with a read/write copy of the page, or if there are read-only copies, the field defines which node has the most recent version of the page. If a copy of the page is present in the main memory of the owner node as well as in the memories of other nodes, the protocol guarantees that the current version field contains the owner node number[5]. If the owner node has an up-to-date copy of the page, but not currently in its main memory, and an XPT entry exists for the page, the current version field is set to null. The full use of the current version field is explained later in this section. The XPT for a diskless node is always empty because diskless nodes cannot act as page servers.

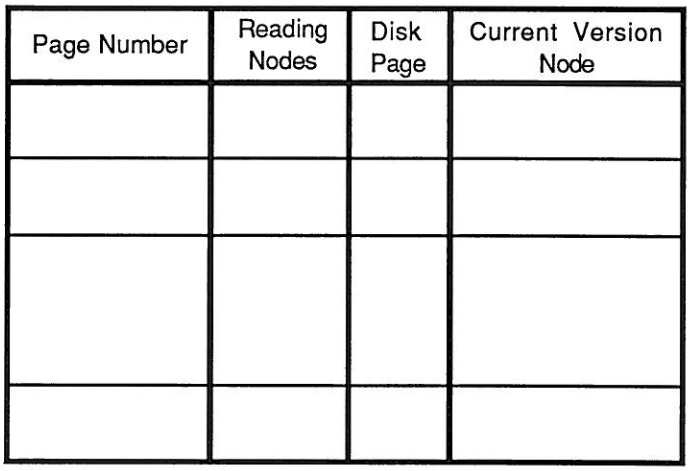


Figure 5.3. The Structure of the Exported Pages Table.

Nodes also maintain a table listing imported pages currently in their main memory. This table is called the *Imported Pages Table* (IPT), with entries consisting of a *server node number* field, a *page number* field which identifies the page, and a main memory page frame number field. The structure of the IPT is shown in figure 5.4.

---

[5] If the page is present in the memory of the owner node *only*, no XPT entry exists for the page.

Virtual Page Number	Server Node	Main Memory Page Frame Number

Figure 5.4. The Structure of the Imported Pages Table.

Pages in the physical memory of a node may be marked as read-only or read/write in the node's ATU. The default state for a page read from a local disk for local use is read/write, whereas the default state for a remote page is read-only. Any number of read-only copies of a page are allowed to exist in the physical memories of nodes in the network at any one time. The coherency protocol guarantees that at any time there is either

(a) zero or more read-only copies of the page or

(b) exactly one read-write copy.

Thus, if a read-write copy of a page exists in the physical memory of a node, then it is the only copy of the page in physical memory of any node in the network. The kernel of the owner node maintains, in the XPT, a record of any copies of a page that have been sent to other nodes. This is analogous to the way page owners in IVY [74] keep a copy set for each page[6].

Unlike the "shares" approach proposed in [57], which allows *any* node to distribute copies of a "granule" (subject to its share status), a MONADS owner node is responsible for controlling the distribution of its pages. This is similar to the method of page distribution used in the distributed Napier architecture [72, 121]. Distributed Napier uses a

---

[6] Page ownership status in IVY reflects the owner node's right to modify the page. MONADS page ownership reflects the fact that the page is stored at the owner node.

centralised Stable Store Server which acts as a page server for the entire store, and to which all page requests are made. This server is responsible for controlling the supply of pages to the connected client nodes and the granting of appropriate page access rights. When a page request is received by the server it either supplies the page from its own main memory, or it instructs a client holding a copy of the page to transmit a copy of the page to the requesting client. The MONADS DSM extends this approach by distributing the page server role across all nodes with local disk.

5.3.1.1 Obtaining a Read-only Copy of a Page

When a page fault for a remote page occurs at a node, the kernel determines the owner node for the page from the page address, and transmits a *request_page* message to the owner node. This message has the requesting node number and the requested page number as parameters. At the time of receipt of a request_page message by the owner node, one of several scenarios may apply:

(1) no copy of the page exists in main memory network-wide, or

(2) one or more read-only copies of the page exist in main memory network-wide but *not* in the main memory of the owner node, or

(3) one or more read-only copies of the page exist in main memory network-wide *including* a copy in the main memory of the owner node, or

(4) a read/write copy of the page exists in the main memory of a node, or

(5) it has no knowledge of the page.

The requesting node must have sent the request_page message as a result of a page fault, so the requested page is not in the main memory of the requesting node. The action taken in each of the above situations is:

(1) A read-only copy of the page is transmitted to the requesting node using a *supply_page* message with the page number and page data as parameters. Because the page is not in the owner node's main memory at the time of the request, a page fault situation is created at the owner node by reading from an arbitrary address in the page. When the page has been read from the disk into the owner node's main memory, it is mapped into the ATU as read-only, and the supply_page message is transmitted. The owner node's number is stored in the current version field of the XPT entry for the page, indicating that the most up-to-date version of the page is in the main memory of the owner node, and the requesting node's number is stored in the read nodes field.

(2) Since the page is not in the main memory of the owner node in this case, an XPT current version field entry for the page, if it exists, is not for the owner node[7]. If a current version entry does exist, a *send_page* message is sent to the node listed in the current version field. Parameters to the send_page message are the node number of the node requesting the page, and the page number. If the XPT current version field entry for the page is a null entry[8], then the send_page message is transmitted to an arbitrarily selected node whose number appears in the read nodes field. In either case the receiver of the send_page message responds by transmitting a supply_page message to the requesting node, and the requesting node's number is added to the read nodes field of the owner's XPT.

(3) The owner node transmits a supply_page message to the requesting node. If no entry for the page exists in the owner node's XPT[9], an XPT entry is created for the page and the owner node is entered in the current version field. The requesting

---

[7] The existence of a non-null entry in the current version field indicates, in this case, that the owner node does not have the most recent version of the page.

[8] This indicates that the owner node does have the most recent version of the page, but in this case not currently in its main memory.

[9] This indicates that no node other than the owner had the page in main memory prior to the request_page message.

node's number is added to the read nodes field of the owner's XPT.

(4) If the read/write copy is in the main memory of the owner node, the page is marked as modified read-only in the owner node's ATU and an entry is created for the page in the XPT. This entry has the owner node number in the current version field and the requesting node in the read nodes field. The owner node then transmits a supply_page message to the requesting node. If the read/write copy is not in the main memory of the owner node, a send_page message is sent to the node with the read/write copy. When the writing node receives this message, it marks the page as read-only in its ATU[10], and transmits a supply_page message to the requesting node. The owner node enters the requesting node in the read nodes field of the XPT entry for the page.

(5) All the server node can do in this situation is transmit an *invalid_address_space* message to the requesting node. This message contains the requested page number as a parameter.

In all cases except case (5), the page is mapped into the requesting node's ATU as read-only, the occupied main memory page frame is marked as imported in the MMT, and an entry is made in the IPT recording the page number and owner node number. In case (5) an exception condition exists at the requesting node.

If a page fault occurs at the owner node, the kernel must determine whether copies of the page exist at other nodes. To do this it checks the XPT. If no XPT entry exists for the page, then the page is brought into main memory from local disk, and mapped into the ATU as read/write. If a remote copy of the page does exist, then it is usually more efficient for a copy of the page to be retrieved from such a remote node than to load it into main memory from local disk [27], particularly

---

[10] The page would have been marked as *modified* in the node's ATU when modification occurred. The combination of modified and read-only in the ATU indicates that the node has the most recent version of the page, and must return a copy of the page to the owner node prior to discarding the page.

when the possibility of recursive page faults is considered (see section 4.3.2). The send_page message is used by the owner node to obtain a copy of the page. The recipient of the send_page message is determined by reference to the XPT. If a current version node is listed the message is transmitted to it, otherwise the message is sent to one of the read nodes.

5.3.1.2 Obtaining a Read/write Copy of a Page

A node obtains read/write access to a page by increasing the access rights of a read-only page that already resides in the node's main memory. To do this a remote node transmits a *request_changed_access_rights* message to the page owner. Parameters to the message are the requesting node number, the page number, and the required access rights (in this case read/write).

The situation at the time of the read/write request is that either

(1) a read-only copy of the page exists in the main memory of the requesting node only, or

(2) one or more read-only copies of the page exist in main memory system-wide, including the memory of the requesting node.

The actions taken in each case are:

(1) the owner node transmits an *access_rights_changed* message to the requesting node granting a change in page status to read/write. Parameters to the message are the page number and the new access rights.

(2) an *invalidate_page* message is sent to all nodes with a copy of the page (excepting the requesting node). This message, which has the page number as its parameter, requests that the receiving node's copy of the page be invalidated. The recipients of the invalidate_page message are determined from the read nodes field and the current version field of the page entry in the owner node's XPT. The reply to an invalidate_page message is the *page_invalidated* message, which has the sending node's

node number and the page number as parameters. As each page_invalidated message is received, the owner node removes the entry for the sending node from the page entry in the XPT. When the only node remaining in the XPT entry is the requesting node, all page_invalidated messages have been received, so the owner node transmits an access_rights_changed message to the requesting node granting a change in page status to read/write.

In each case the owner node enters the requesting node's number in the current version field of the Exported Pages Table entry for the page (also removing the requesting node from the read nodes field if necessary). This indicates to the owner that the requesting node now holds the most recent version of the page because the page is held with read/write access. The requesting node also marks the page as read/write in its ATU. A copy of the page in the main memory of the owner node would, of course, also be invalidated prior to the transmission of the access_rights_changed message, but this invalidation would be an internal kernel operation and would not require transmission of a message.

5.3.1.3 Page Discard

As part of the management of virtual memory at a node, page discard may occur. If the page to be removed is local, the kernel checks the XPT to determine whether an entry exists for the page. If an entry does exist, then the current version field indicates the owner node. The entry in this field must be replaced by a null entry, indicating that the page is no longer in the main memory of the owner node. The modified status of the page in the ATU indicates whether the page must be flushed to disk prior to discard. If the page is marked as modified then the copy in main memory is more recent than the copy on disk and so the page must be flushed to disk. If the page is marked as unmodified then the copy on disk is the most recent and the copy in main memory may be discarded.

If the page to be removed is not local, as indicated by the presence of an IPT entry for the page, then either

(a) the page is read-only, or unmodified read/write[11], in which cases a page_invalidated message is sent to the owner node indicating that the page has been removed, allowing the owner node to update its XPT by deleting the entry for the removed page. In the case of the unmodified read/write page, or in the case of the removal of the last exported read-only copy of the page, the XPT entry for the page is removed because no copy of the page exists in main memory network-wide, or

(b) the page is modified read/write or modified read-only, in which case the page status is reduced to read-only (thus preventing further modification of the page in the case of modified read/write) and a copy of the page is sent to the owner node using a *return_page* message. The owner node receives the page into its main memory and maps it in as modified read-only. The disk page field in the XPT entry for the page is used to create the appropriate MMT entry. The owner node then updates its XPT by replacing the current version field in the entry for the returned page with its own node number. If this replacement results in an XPT entry with no read nodes for the page[12], the XPT entry may be removed. The owner node then confirms receipt of the page with a *page_received* message back to the returning node. The page_received message has the receiving node number and the page number as parameters. The page is not removed from the memory of the returning node until the page_received message is received by the returning node. At this stage the IPT entry for the page is also removed.

The orderly shut down of a node is a related problem because shutting down a node may involve firstly flushing virtual pages to disk. We discuss node shutdown in the following section.

---

[11] A read/write page may be unmodified if the process wishing to modify the page had not yet been reactivated since receipt of the access_rights_changed message that granted the read/write access.

[12] This would be the case if the discarded page had been held with read/write access rights immediately prior to the discard.

5.3.1.4 Node Shutdown

When a node is shut down in an orderly fashion by an operating system utility similar to the Unix *halt* instruction [111], modified virtual pages held in main memory are flushed to disk. In the case of the MONADS DSM, the orderly shut down of a node may involve other nodes in the network because of the presence of

(1) imported pages in the node's main memory, and

(2) pages owned by the node in other node's main memories.

An imported page in the node's main memory may be either modified or unmodified. A copy of any modified page must be returned to the owner node prior to discard of the page to ensure that the modifications are not lost. This is achieved using the return_page message. On receipt of this message the owner node adjusts its XPT by either removing the entry for the page (if the shut down node was the only remote node with a copy of the page), or writing its own name in the current version field of the XPT entry (thus indicating that the current version of the page resides in the owner's main memory). An unmodified page may be simply discarded[13]. Such discard is signalled to the owner node using the page_invalidated message. On receipt of this message the owner node adjusts its XPT entry for the page by removing the shut down node from the read nodes field, and removing the entry entirely if no remote nodes have copies of the page.

Prior to shutdown a node must retrieve any pages owned by it for which another node holds the current version. This is necessary to enable the most recent version of these pages to be flushed to disk at the owner node. The owner's XPT contains an entry for every page exported by the node. An invalidate_page message is transmitted for each page to the remote nodes listed in the XPT. When this message is received, the receiving node invalidates the page if it is unmodified. If the page is modified, the receiving node

---

[13] The owner node or a node listed in the current version field of the owner's XPT entry for the page is responsible for ensuring that the page is flushed to disk if necessary.

(1) reduces the page access to read-only if it is held with read/write access,

(2) returns a copy of the page to the owner node using a return_page message, and

(3) waits for a page_received message confirming that the owner node has received the up-to-date copy. When this message is received the page is invalidated.

An alternate approach to the shutdown of an owner node is to use a *node_shutting_down* message with the sending node number as a parameter. This message is transmitted to every remote node listed in the sending node's XPT as having pages owned by the node. On receipt of such a message, a node must remove the sending node from its NAT and return an up-to-date copy of every modified page owned by the sending node as described above. The importing nodes may then

(1) invalidate any pages owned by the shut down node, as described above, or

(2) continue to use the pages with read-only access until the pages are no longer needed and discarded. Any attempt to modify such a page would result in an access violation condition. The fact that the owner for these pages is off-line can be detected from the lack of a NAT entry for the owner, so on page discard no page_invalidated message would be transmitted. This option allows, for instance, a library segment already imported into main memory to remain available to a compiler even though the owner node is shut down.

5.3.1.5 Discussion

The described protocol guarantees that nodes have a coherent view of the virtual memory. The DSM system can, however, experience page thrashing, meaning that a page is constantly being transferred between nodes whilst not being in the nodes' main memory long enough to be used by them. This situation occurs if a number of nodes concurrently attempt a series of writes to the page. Initially it appears that higher

level synchronisation mechanisms would prevent such thrashing. However, because page and segment boundaries are decoupled, there is no guarantee that the nodes are attempting to write to logically related data in the page, because they may in fact be attempting to write to different segments. In such a situation higher level synchronisation mechanisms alone would not prevent page thrashing.

The likelihood of page thrashing can be greatly reduced by guaranteeing a writer a small time interval (a few milliseconds) of uninterrupted write access to a page. This can be implemented by the writing node, which waits the appropriate interval before acting on a send_page message. On expiration of the time interval, the writing node, in accordance with the coherency protocol, reduces its access to the page to read-only, and transmits a copy of the page to the node indicated in the send_page message.

The issue of higher level process synchronisation is discussed in the next section.

5.4 Process Synchronisation

Process synchronisation is necessary to prevent two or more processes from accessing some shared resource (e.g. memory, file) when the final result depends on the order of the process' access to the data. The problem in such situations is that more than one process is able to access the shared resource *at the same time.* The solution is to implement *mutual exclusion,* that is to ensure that if one process is accessing the shared resource, then no other process is able to access the resource. The part of the program code in which the shared resource is accessed is called a *critical section.*

There are a number of well known methods of achieving mutual exclusion, including *busy waiting* [89, 90], and semaphores [41, 60, 66]. Busy wait and test and set are inefficient both in terms of processor utilisation and network traffic. Semaphores are supported by the MONADS kernel [69], and their use will successfully synchronise access to the DSM. The problem with the use of semaphores in the DSM is that processes blocked on a semaphore S are potentially suspended on a number of different nodes in the network. Associated with this is the

possibility that the kernel at a node may have to activate a process suspended at another node.

A MONADS semaphore *S* is a shared data structure consisting of a counter and a set of waiting processes. Let us consider a process wishing to enter a critical section protected by *S*. The process uses the hardware P instruction with *S* as a parameter to register its desire to enter the critical section. The result of this operation is that the semaphore counter is decremented and either the process is immediately allowed to enter the critical section or the process is suspended waiting for the section to become free. If the process is suspended it is added to the set of waiting processes.

When a process leaves the critical section it signals this fact with a V operation on *S*. The effect of the V operation is to increment the counter, and if any processes are suspended on the semaphore, one of them is activated. On a single MONADS node, such activation simply involves moving the newly activated process from the suspended to the waiting kernel queues, from which the process will be activated by the process scheduler. When the nodes are part of a network, a V operation may involve the activation of a process suspended on a different node. Thus the node number must be maintained for suspended processes, allowing a node-to-node message to be used to activate them.

The message used to activate a remote process is the *activate_process* message, which has the process number as a parameter. Since each MONADS process exists in a unique *stack address space*, and the creating node number is embedded in the address space number, the destination node for the activate_process message forms part of the process number itself. When the activate_process message is received, the receiving kernel uses the process number to identify the process, after which it moves the process from the suspended queue to the waiting queue.

The *in-process* model used for MONADS processes [62] which results in a manageable number of processes system-wide, coupled with the unique naming of address spaces, results in unique process identification without the need for special process naming protocols (e.g. [27, 84]). Network-wide process synchronisation is thus achieved

with the addition of only one message to those needed for the memory server functions and coherency control.

5.5 Conclusion

In this chapter we described the MONADS DSM model. This model exploits the wide addresses provided by the MONADS architecture by including an owner node number in every virtual address. This embedded owner node number allows the kernel at a node to determine whether a page fault is resolveable locally. If not, the node number is used by the kernel as the logical destination address for a message requesting provision of a copy of the appropriate virtual memory page. When the page is received, it is used to resolve the page fault in the same way that page faults are resolved by access to local disk in non-networked machines. Data coherency is maintained using a multiple reader/single writer protocol.

The creation of the DSM involved changes only to the page fault handling and process scheduling parts of the centralised MONADS kernel. User programs written for centralised use can run unmodified on the networked machines because data is accessed by address rather than by location and name. The design supports the use of one or more nodes with attached disks which perform a memory server function, and allows for diskless nodes.

In describing the scheme we made the assumption that modules and volumes always remain in their original locations. This assumption, whilst necessary to enable addresses to define owner nodes, results in an overly restrictive system because it precludes the mounting of removeable disks on foreign nodes and the movement of modules between nodes. In chapter 6 we show how the MONADS DSM model may be extended to allow the movement of modules between volumes and volumes between nodes.

Chapter 6 ADDRESSING MOVED MODULES

6.0 Introduction

The discussion in chapter 5 assumes that once a volume is mounted at a node it is never moved from that node, and that an address space (or module) is never moved from the volume on which it was originally created. This assumption allows us to rely on the fact that the creating node number and within node address space number embedded in the module name and the virtual address of data within the module accurately describes the location of the module and its data. When a remote page fault occurs the kernel uses this embedded information to determine the storage or *owner node* for the page. The kernel then transmits a request_page message to the owner node as part of the page fault resolution protocol.

Disallowing the migration of volumes between nodes and the migration of modules between volumes results in an architecture that is unacceptably restrictive. It is sensible, for instance, to allow a disk from a failed node to be mounted at another functional node, thus allowing the data stored on the disk to be accessed while the failed node is being repaired. It should also be possible to mount removeable disks on any node with a suitable disk drive unit. When a user moves permanently from one node in the network to another, it is sensible to move the modules owned by the user to a volume mounted on the new node, thus allowing the user to access his or her modules without using network bandwidth.

Achieving such migration is not a trivial task because the location information embedded in addresses is crucial to efficient access to data in the network. Location information cannot be simply changed to reflect the new node and volume numbers when data migrates. This is because such a change would effectively move each module to a different address space, thus changing the module's name. The name field of a *module capability* used to access a module contains the address space number of the address space occupied by the module. Existing module capabilities for such a moved module would thus

contain the address space number of the *old* address space in the name field. This means that these existing module capabilities would instantly become invalid, and their subsequent use would result in an invalid module exception.

In this chapter, which expands work previously published by the author [19], we discuss how volumes are permitted to migrate between nodes in the network, and how modules are permitted to migrate to different volumes. In the following section, section 6.1, we discuss the relocation of *volumes*. We then discuss, in section 6.2, the movement of *modules* between volumes.

6.1 Relocating Volumes

In this section we consider the relocation of a volume from one node to another. The relocation of a volume from one physical disk to another disk mounted on the same node[1] is logically equivalent to no movement at all, and as such is not discussed further. The relocation of a complete volume may involve

(a) placement of a removeable disk in a disk drive unit connected to a different node in the network, or

(b) connection of a fixed disk to a different node in the network.

When a volume is relocated to a different *node*, the node number embedded in addresses stored on the volume describes the *creating* node rather than the *owner* node. This means that the node number/volume number part of the module capabilities used to access modules on the volume may no longer be sufficient to allow the page fault handler to locate pages of the module.

From the viewpoint of a node wishing to resolve a page fault, such relocated volumes fall into two categories:

(1) volumes created on another node and mounted locally, and

---

[1] This would occur when restoring from backup media after a disk failure, or when moving a volume to a faster disk to improve performance.

(2) volumes mounted on some remote node A but created on another node B.

In the following sub-sections we discuss resolution of page faults for pages stored on relocated volumes in each of these categories.

6.1.1 Locally Mounted Volumes

In chapter 5 we assumed that all locally mounted volumes had been locally created. Each entry in the Local Mount Table (LMT, see section 4.3.2) thus consisted of fields defining the volume number and the disk drive number. The LMT did not need to define which node created the mounted volume because this was, by implication, the local node. Allowing volumes to migrate between nodes renders such a LMT inadequate. For example, using the LMT described in section 4.3.2, an entry for volume V created at node A and now mounted on node B is indistinguishable from an entry for volume V created at node B and still mounted on node B.

To allow differentiation between volumes created and mounted locally and volumes created on another node but mounted locally, an extra field, the *creating node number* field is added to the LMT, as shown in figure 6.1. This extended LMT allows the kernel at a node to recognise *foreign* volumes mounted on its *local* disk drives. Using the above example, the creating node number field in the extended LMT differentiates between entries for volume AV created at node A and currently mounted at node B and volume BV created and mounted at node B.

When the page fault handler attempts to resolve a local page fault, it compares the unique node number/volume number pair in the faulting address with the creating node number/volume number fields in the LMT. If a match is found, the required volume is locally mounted. In section 4.3.2 we showed how a virtual memory page, stored on a volume V_1 which was created on some node A and currently still mounted on node A, may be accessed by a process running on node A. This same technique is used to allow the pages stored on volume V_2, which was created on node A but currently mounted on node B, to be accessed by a process running on node B. The extension to the LMT

has thus allowed all locally resolveable page faults to be identified. These page faults are then resolved by access to the appropriate locally mounted volume.

Volume Number	Disk Drive Number	Disk Address of Page Zero of Address Space Zero	Creating Node

Figure 6.1. The Structure of the Extended Local Mount Table.

In this section we showed how a page, stored on volume V_2 which was created on node A but currently mounted on node B, may be accessed by processes running on node B. The kernel on node B recognises that the volume is locally mounted by referring to its Local Mount Table (LMT). In the next section we describe how processes running on nodes other than node B (including processes running on node A) access pages stored on volume V_2.

6.1.2 Moved Volumes Mounted on Remote Nodes

The problem with access by other nodes to a page stored on a relocated volume such as V_2 is that the requesting kernel has no idea where to send the request_page message. This is because the node number embedded in the virtual address of the required page is the number of the volume's *creating* node. Since the volume has been relocated, this creating node number does not indicate where the volume is currently mounted. The solution is for every kernel to maintain a *Foreign Mount Table* (FMT) with entries included on a *need to know* basis. The FMT describes the nodes on which *relocated* volumes are currently mounted. The table has entries consisting of a *creating node number/within node volume number* field and a *node where mounted* field, as shown in figure 6.2.

Creating Node/ Within Node Volume	Node Where Currently Mounted

Figure 6.2. The Structure of the Foreign Mount Table.

The FMT at a node N contains an entry for each relocated volume that is of interest to N. Relocated volumes of interest are either:

(1) volumes not mounted on N which contain virtual memory pages being used by processes running on N, or

(2) volumes which were originally created on N and that are currently mounted on some other node. Such a volume is of interest to N if a node other than the mounting node is using virtual memory pages contained in the volume.

As suggested by this definition of "relocated volumes of interest", the FMT is used by the kernel for two purposes:

(1) to enable a non-owner kernel to transmit request_page messages to the correct owner node for the page, and

(2) to enable an ex-owner kernel to advise nodes about the current location of relocated volumes created by it.

In the following sub-sections we describe the use of the FMT for each of these purposes.

6.1.2.1 Obtaining Pages From Relocated Volumes

When a process running on node N generates an address, the address indicates a virtual memory page that is stored either

(a) on a volume mounted on node N, or

(b) on a remote volume mounted on its creating node, or

(c) on remote volume not mounted on its creating node.

If a copy of the virtual memory page is not in main memory, the ATU generates a page fault. When a page fault occurs, the kernel extracts the node number/volume number pair from the address and attempts to find a matching entry in the LMT. Such an entry exists if the volume containing the page is in category (a). If the LMT entry does exist, the page fault is resolved as described in section 6.1.1.

If the volume is not in category (a), an LMT entry does not exist for the volume. This means that the page fault is remote, and that a request_page message must be transmitted to the owner node. The owner node is the mounting node for the volume containing the page. The volume, of course, may or may not be mounted on its creating node, meaning that the owner node may not be the creating node. When a page fault occurs, the kernel must determine the mounting node for the volume containing the faulting page. Once the kernel knows where the volume is mounted, it can continue resolution of the page fault.

A category (b) volume, once it has been identified as such, is not a problem because its mounting node is the node whose number is embedded in the faulting address. Pages stored on such a volume are obtained as described in section 5.3.1.1.

The mounting node for a category (c) volume is not defined by the faulting address, and is not described in the LMT. It is to distinguish such volumes from category (b) volumes that we use the FMT, which identifies those remote volumes not mounted on their creating nodes. Let us assume at this stage that the required FMT entries exist. In section 6.1.2.2 we describe how FMT entries are created.

The full sequence of steps taken by the kernel to determine where the volume is mounted is:

(1) Compare the node number/volume number for the volume with the equivalent field in the LMT. If a matching entry is found, this means that the volume is locally mounted and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

(2) Compare the node number/volume number for the volume with the equivalent field in the FMT. If a matching entry is found, transmit a request_page message to the node indicated in the FMT. If the appropriate FMT entry does not exist, then

(3) Assume that the volume is still mounted on the creating node, and transmit a request_page message to the creating node indicated in the faulting address.

By taking these steps the kernel is able to determine the mounting node for any currently mounted volume. In defining the steps, however, we simply assumed that all necessary FMT entries are present. In the next section we describe how the FMT entries are created.

6.1.2.2 Creating the Foreign Mount Table

Let us consider again the situation in which a volume V_2, which was created on node A, is unmounted from A and subsequently mounted on node B. The first time after this move that any node, except for nodes A or B, attempts to access a page from V_2 it expects that the volume is still mounted on the creating node A. Node A knows that V_2 *is not* locally mounted because its kernel has no LMT entry for the volume[2]. Node B knows that the volume *is* locally mounted because its kernel does have a corresponding LMT entry. Thus the only node with accurate knowledge of the mounting node for V_2 is node B.

The kernel at the creating node, node A, acts as an adviser for other nodes with an interest in the location of V_2. To enable it to fulfil this role, the kernel at A must itself determine where V_2 is mounted. We consider that two strategies for this determination are worthy of discussion.

---

[2] However, node A does not, necessarily know *where* the volume is mounted.

(1) The creating node is informed whenever a volume created by it is mounted on another node. This strategy gives advance notice of volume location to creating nodes.

(2) The creating node determines where a volume created by it is mounted only when it detects an expression of interest in the volume. This strategy extends the need to know basis for FMT entries to the creating node.

The implementation of strategy (1) is simple. Whenever a moved volume is mounted, the kernel at the mounting node transmits a *volume_mounted* message to the creating node. This message includes the *mounting node number* and the *volume number* as parameters. On receipt of this message the creating node creates an FMT entry for the volume. In the example above, B transmits a message to A informing it that V_2 is now mounted at B. The kernel at node A creates an FMT entry recording that node B is the mounting node for V_2 . At this stage only nodes A and B are aware of the correct mounting node for V_2.

If node A receives a request_page message sent by node C and asking for a page from V_2, node A responds to the request, and to all such subsequent requests from other nodes, with a volume_mounted message back to the requesting node. This volume_mounted message has node B (the mounting node number) and V_2 (the mounted volume) as parameters. On receipt of this message, node C updates its FMT and repeats the request_page message, this time to the current owner node (in this case B). The problems with this strategy are that

(a) the FMT at the creating node contains an entry for every moved volume created at the node, even if the volume is not being accessed by the creating node or remote nodes other than the mounting node. Such entries are wasteful of space, and

(b) if the creating node is off-line when such a moved volume is mounted, it does not receive the volume_mounted message, and so does not create an FMT entry for the volume. In this situation the node must follow strategy (2) when it detects an expression of interest in the volume.

The implementation of strategy (2) is as follows. Since FMT entries are included on a need to know basis, A waits for an expression of interest in V_2 before attempting to determine where the volume is currently mounted. If, for instance, V_2 is a partition of a removeable disk mounted at node B purely for use at that node, the kernel at A need never be aware of such mounting.

From the viewpoint of node A, an expression of interest in V_2 is either

(a) a page fault for a page on V_2 that occurs at node A itself, or

(b) the receipt by the kernel at A of a request_page message asking for a page stored on V_2. Let us assume that such a message was sent by a third node C[3].

In either of these situations, the kernel at A must determine where V_2 is currently mounted. In the case of the local page fault, the location of V_2 is needed so that the kernel at A can request the page from the new owner node. In the case of the receipt of a request_page message from node C, the kernel at A must determine the location of V_2 so that it can advise the kernel at C of the new owner node's number.

To determine the new location of V_2, node A broadcasts a *where_is_volume_mounted* message. This message has the *requesting node number* (A) and the *required volume number* (V_2) as parameters. When the mounting node receives a where_is_volume_mounted message, its kernel responds with a volume_mounted message, including the mounting node number and the volume number as parameters.

On receipt of the volume_mounted message, node A updates its FMT, in this example with an entry indicating that V_2 is currently mounted on node B. If the original interest in V_2 was a page fault at A, this page fault may now be resolved by the transmission of a request_page message to B. If the original interest in V_2 was a request_page message from node C, node A responds to the request, and to all such subsequent requests from other nodes, with a volume_mounted

---

[3] Node C is, at this stage, unaware of the relocation of V_2. It therefore assumes that its page fault can be resolved by the creating node A.

message back to the requesting node. This volume_mounted message has node B (the mounting node number) and V_2 (the mounted volume) as parameters. On receipt of this message, node C updates its FMT and repeats the request_page message, this time to the current owner node (in this case B).

The described scheme assumes that the creating node, A, is on-line at the time of an attempted access to volume V_2. If this is not the case, and node A is off-line, node C does not receive a reply to its request_page message and eventually times out on its request. Simply failing the request at this stage is not good enough, because V_2 may have been moved to node B for the express purpose of maintaining access to it during rectification of some breakdown at node A. On time-out of the request_page message, node C broadcasts a where_is_volume_mounted message. Node B responds by transmitting a *node_mounted* message to C. On receipt of this message, the kernel at C updates its FMT and transmits the request_page message to node B. Thus if the creating node A is off-line, other nodes may still determine the mounting node for V_2, meaning that pages stored on the volume are accessible.

It is possible, of course, that both the mounting node, B, and the creating node A, are off-line[4]. In this case, C does not receive a reply to its request_page message or to its subsequent where_is_volume_mounted message. Time-out on the reply to *both* of these messages indicates that V_2 is not mounted anywhere, and thus the required page is not available. In this case any existing FMT entry for the volume is removed, indicating that the kernel has no knowledge of its current location. When the volume is remounted, and node C subsequently re-accesses a page on it, the FMT entry is recreated if necessary as described above. In the unlikely event of the FMT becoming too large for the space allocated to it by the kernel, the kernel simply removes all entries, and rebuilds the table incrementally according to the current DSM access requirements.

---

[4] In the general case, the mounting node and the creating node may be one-and-the-same.

6.1.3 Discussion

The described protocol ensures that the pages on a mounted volume may be accessed *by name* even if the volume is not mounted on its creating node[5]. This is significant because the node number of the creating node is embedded in virtual addresses as an integral part of the DSM addressing scheme. The result is that a volume may be relocated without invalidating the existing capabilities that allow access to the segments stored on the volume.

Two techniques for locating a moved volume are presented. In both of these techniques the creating node for a volume performs an advisory role in informing remote nodes of the current location of the volume.

The first technique involves the creating node maintaining location information about all of its moved mounted volumes. The advantage of this technique is that if the creating node is on-line when such a volume is mounted, broadcast messages requesting location information are avoided. If the creating node is off-line when the volume is mounted, the use of the second technique still allows the creating node to fulfil its advisory role. The disadvantage of the technique is that the FMT at a creating node contains an entry for all of its remotely mounted volumes. An entry would exist, for instance, for a removeable disk moved to another node for processing exclusively at that node. In such a situation the FMT entry at the creating node is never used. A compromise solution is to use a parameterised *mount* operation, with the parameter defining whether the creating node is to be informed of the mount if the volume is foreign. This parameter would normally be false if the mount is temporary (as in the case of a removeable disk), and true for long-term mounts (as in the case of a fixed disk).

According to the second technique, the creating node determines where one of its moved volumes is mounted only when it needs to know this information to perform its advisory role. This technique has the disadvantage that it requires broadcast messages, but the advantage that entries are only made in the owner node's FMT when such entries

---

[5] At this stage we assume that modules are never moved from their original volume. The problem of moved modules is discussed in section 6.2.

are needed. This means that the second technique generally results in smaller FMTs.

An alternate solution to the problem of locating moved volumes was seriously considered and eventually rejected. This option involved the mounting node broadcasting a volume_mounted message whenever a volume was mounted. This option was rejected for the following reasons:

(1) all nodes would process and act on the broadcast message, even those which will never have an interest in the pages stored on the volume, and

(2) it potentially results in unnecessarily large FMT tables because nodes would maintain information about *every* relocated mounted volume.

These reasons for rejection particularly apply to *removeable disks*, which are commonly mounted to enable *local* access to the volumes on the disk, with no intention of remote access. Network-wide knowledge of the mounting of such removeable disks seems a waste of network bandwidth and processing time at remote nodes.

6.2 Relocating Modules

In some circumstances it may be desirable to move individual modules from one volume to another, possibly but not necessarily between disks mounted at different nodes[6]. A reason for this may be that a user moves to a different site and wishes to move his data from a shared volume on the node at the old site to a volume mounted on the node local to the new site. Another reason may be that a user wishes to move a module off a volume mounted at his or her home site to a removeable disk prior to travelling.

If the module were *copied* onto the new volume, the copy would be to a new address space, and so the *name* of the module would change,

---

[6] Process *stacks*, whilst stored in address spaces, are not modules, and are not included in this discussion. The system at this stage does not support movement of stacks.

meaning that existing module capabilities would not allow access to the copy. In the case, for example, of user data for which no-one but the owner has a capability, this would be satisfactory, because the owner could simply replace his old module capability with a new one. If, however, several copies of the module capability exist, then it may not be possible to replace all of them with the new capability because when a module owner distributes capabilities allowing access to a module he does not necessarily maintain any record of the recipients of these capabilities. In this case the ability to move a module must be achieved without changing the name of the module.

The DSM addressing scheme described in chapter 5 relied on the owner node information embedded in virtual addresses to provide the destination for request_page messages. In section 6.1 we described how virtual pages stored on moved volumes are accessed by remote nodes. This required that the kernel at each interested node maintain information about such relocated volumes. The movement of individual modules between volumes presents a further addressing problem, because the volume information embedded in the virtual addresses of the module does not describe the location of the module. Any scheme that allows the movement of an individual module must allow the module to be accessed by its original name.

Addressing of moved modules involves two distinct phases as follows.

(1) The first access to a module will always be an *open* operation (see section 4.4.1). During this operation the kernel has access to a module capability for the module, and from this it can determine the original virtual address of the root page of the module. By extending the structure of a module capability to include additional location information, we can take advantage of the availability of a module capability when opening a moved module.

(2) Once a module is open, it may be necessary to access other pages from it. Such accesses, for a non-moved module, are achieved using information embedded in each page address (see sections 5.2 and 6.1.2.1). This embedded information does not correctly describe the location of pages for a moved module.

Since the module containing the pages is already open, the accesses are not accompanied by a module capability, and thus no additional information is available to assist in location of the pages.

Addressing of a moved module is achieved using structures established when the module is opened. Allowing open modules to be moved introduces the extra complexity of dynamic modification of such structures. We avoid such complexity by restricting movement of modules to those which are *closed*. This is not seen as a severe restriction. It is the equivalent of restricting the movement of open files on a conventional system. As explained in section 4.4.1, a count of the number of processes that have a particular module open is maintained in the module. An attempt to move a module fails if its open count is greater than zero.

6.2.1 Addressing Moved Modules

When a process wishes to access a module, it executes the *open* system call. This call, which includes the module capability as a parameter, causes the creation of a *Module Call Segment* (MCS) for the module. The MCS contains segment capabilities for the root of the data encapsulated in the module and for the available module interface procedures. To set up the MCS, the root page of the module must be obtained from disk, assuming it is not currently in main memory (see section 4.4.1). To facilitate this initial read operation, module capabilities are extended to include information which advises the probable storage location for moved modules. This advisory data is readily available to the kernel when the module is opened because the module capability must accompany the open call.

The probable storage location is provided by the inclusion of an additional *advisory* field in module capabilities. The advisory field consists of a node number/volume number pair, as shown in figure 6.3. Unlike the other fields of module capabilities, this information is not privileged, and the owner of the capability can cause arbitrary values to be placed in it. The purpose of the advisory field is to provide a place for the capability owner to store the new volume number for moved

modules. When the capability owner issues the *move_module* command, providing the *module capability* and *destination volume* number as parameters, the advisory field of the module capability is changed to the destination volume as part of the move_module routine. Similarly, a user may change the advisory field in an owned module capability if he becomes aware that another user has moved the module. As explained in the following sections, the system also changes the value of the advisory field when it detects that the current value is incorrect.

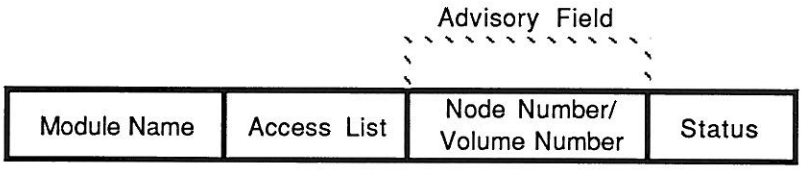


Figure 6.3. The Structure of the Extended Module Capability.

Using the module capability advisory information the requesting node transmits a message to the probable owner node asking for provision of the module root page. This message contains the requested page number, with embedded *creating* node and volume numbers as a parameter[7]. On receipt of such a message for a page from a moved module, an owner node has no idea of the storage volume for the module containing the page because the storage volume number is not that embedded in the page address. The storage volume number is, however, available to the requesting node because it is included in the advisory information.

The storage volume number for moved modules is made available to owner nodes by extending the request_page message used to obtain a copy of a remote page. This message is extended to include a *storage volume number* field containing the full identity of the volume that

[7] For moved modules the creating node and volume numbers are not the same as the storage node and volume numbers.

probably contains the required page[8]. When an owner node receives a request_page message its kernel checks the storage volume number field. If the field is not null, its contents are used by the kernel to find the module containing the requested page[9].

Once the module has been opened, a subsequent page fault for a page in the module presents the virtual address of the page to the kernel, and this, of course, does not include the advisory information. The resolution of such page faults is difficult for pages in moved modules because the new location must be determined for each individual page unless either

(1) the page fault handler maintains a list of open moved modules so that it can detect that the page comes from a module for which it holds advisory information, and uses this information to direct the request_page message to the appropriate node. This is the approach described by the author in [19], or

(2) the addresses used to access data in the moved module have the *new location* of the module embedded in them, meaning that, in effect, the open module takes on the identity of the address space that it now occupies as an *alias*.

We describe both of these options in the following sections.

6.2.2 Maintenance of Open Moved Module Data

This scheme requires four modifications to allow access to moved modules.

These are:

---

[8] Providing the within-node volume number is not sufficient to uniquely identify the volume. The *full* identity of the volume is necessary because the volume concerned may be a moved volume (see section 6.1.1).

[9] If the volume number is incorrect because, for instance, the volume has been moved, the receiving node signals this to the requesting node using an invalid_address_space message.

(1) the use of the *full address space number* in the volume directory stored in address space zero of volumes,

(2) the inclusion of the additional *advisory* field in module capabilities, as described in section 6.2.1,

(3) the addition of a *storage volume number* field to request_page messages as described in section 6.2.1, and

(4) the maintenance of a *Moved Object Table* (MOT) at each node.

6.2.2.1 Extended Volume Directory

Address space zero of every MONADS volume is used to store the red-tape information for the volume (see section 4.3.2). Included in this red-tape is a directory of the address spaces stored on the volume. Because of the creation and deletion of address spaces over time, the address space numbers are typically sparsely distributed, so the directory is implemented as a hash table keyed on address space number to provide fast lookup and a reasonably sized table. The original non-networked architecture did not allow for movement of address spaces between volumes. It was thus sufficient to use an address space number *relative to the volume* as a key into the volume directory.

Recall that each MONADS module resides in a separate address space with a unique number (section 4.4.1). The movement of modules between volumes, then, involves moving the address space in which the module resides. This movement must allow the address space number to remain unchanged, otherwise the module name would similarly change. To allow differentiation between original and imported address spaces stored on a volume, the relative address space number field in the volume directory is extended to include the full address space number, thus including the creating node number and the original volume number. This extended volume directory is shown in figure 6.4. When a module is moved to a new volume, it retains its old module (address space) number, and this number is entered into the new volume directory. Similarly every new module (address space) created on a volume has its full address space number entered into the volume directory.

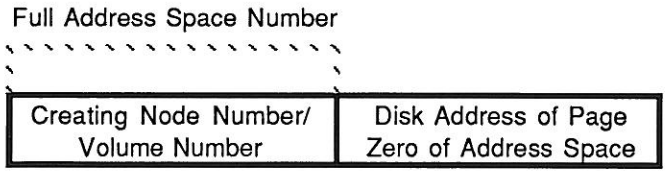


Figure 6.4. The Structure of an Extended Volume Directory Entry.

When a module is moved to another volume, the destination volume number, in the form of a node number/volume number pair, is a parameter to the system call used to achieve the move. The entry for the module in the source volume's directory is altered to indicate the destination volume as a *forwarding address* for the module.

This forwarding address provides a simple indirection mechanism. The operation of this mechanism is described later in section 6.2.2.2. It relies on the source volume being mounted at the time of an attempt to access the module. There is, however, no guarantee of this in systems with removeable disks and/or out of service nodes or disks. If the source volume is not available, the indirection mechanism is useless because the forwarding address cannot be obtained. To improve the probability of locating a moved module in such circumstances, the *advisory field* is added to module capabilities.

The question of removal of forwarding addresses is important in order to control the size of the volume directory for volumes that commonly store transient modules. Such volumes would, for instance, include removeable disks used to take work home at nights. The possibilities for control of forwarding addresses include

(1) the inclusion of a boolean flag as a parameter to the system call used to move a module. Such a flag, which would default to *true*, indicates whether a forwarding address for the module is to be left on the source volume,

(2) the provision of a system call which follows the forwarding addresses for a module. This facility removes the volume directory entry from each volume in the forwarding address chain, and

(3) the use of a higher level ageing mechanism which removes the chain of forwarding addresses for modules that have resided on their current volumes for some arbitrary period of time. This mechanism is implemented using the system call described in (2).

6.2.2.2 The Moved Object Table

The extended volume directory, forwarding address mechanism, and module capability advisory field described in the previous sub-sections allow the location of the root page of any module as required to *open* the module. To allow efficient resolution of page faults for pages from an open module the kernel maintains location information about such modules in the *Moved Object Table* (MOT)[10].

A Moved Object Table is maintained by the kernel at every node, and has entries consisting of a *moved module number* field describing the creating node number, volume number, and address space number for the moved module, a *location* field describing the node number and volume number for the new location of the module, and a *MCS count* field, as shown in figure 6.5. Each MOT entry maps a unique module number to a new location.

An entry is made in the MOT whenever a process executes the open system call for a module that has been moved from the volume on which it was created. The open call, which includes the module capability as a parameter, causes the creation of a *Module Call Segment* (MCS) for the module. The MCS contains segment capabilities for the available module interface procedures and for the encapsulated data of the module. To set up the MCS the root page of the module must be read (see section 4.4.1).

---

[10] This name is historical. It could not be changed to Moved Module Table because the acronym MMT clashes with that of the Main Memory Table.

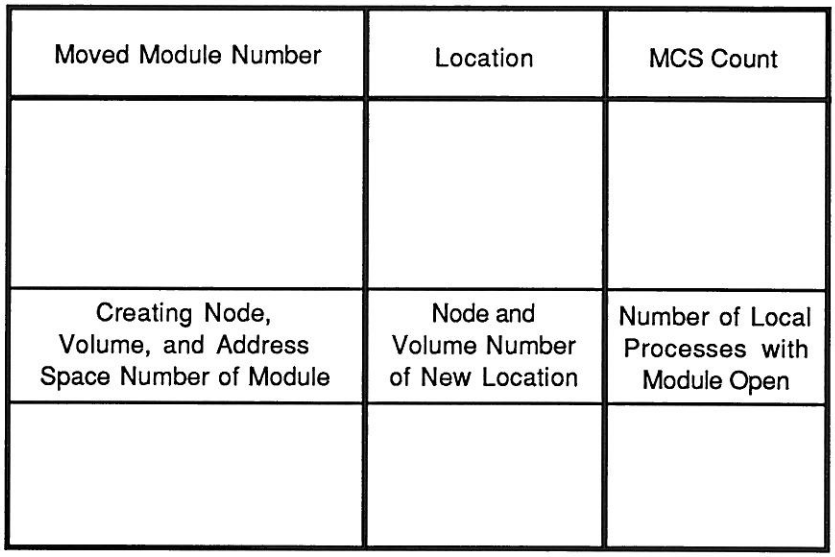


Figure 6.5. The Structure of a Typical Moved Object Table Entry.

If the advisory field in the presented module capability is not null, the MOT is checked to see whether an entry already exists for the module. If not, an entry is created in the MOT using the module number for the moved module field, the contents of the advisory field for the location field, and one (1) for the count field. If an entry does exist, the count field for the existing entry is incremented by one to indicate that another local process has the module open, and the location information in the MOT is used to access the root page of the module.

It is possible that the owner of a module capability is unaware of the fact that the module has been moved from its creating volume, and that the module capability has not been used since the move, meaning that the advisory field is null. It is also possible that the module has been moved since the user or system last updated the advisory field. In both of these cases the contents of the advisory field are incorrect. Prior to any attempt to read the root page, the kernel checks the MOT to see whether an entry exists for the module. If a corresponding entry does exist, the kernel

(1) updates the advisory field in the presented module capability, and

(2) uses the location information in the MOT to access the page.

If not, the incorrect information in the advisory field is used, meaning that the attempt to read the module red-tape results in the return of a forwarding address. In this case the kernel

(1) updates its MOT,

(2) updates the advisory field in the presented module capability, and

(3) repeats the read using the updated location information.

The return of the forwarding address information to the kernel is straightforward if the read is local. If the read is remote, the remote node answers the request_page message with a *module_moved* message, including the *module number* and the *new volume number* as parameters. On receipt of a module_moved message, the kernel updates its MOT accordingly.

It is possible for the return of a forwarding address to occur several times before the read operation is successful. This would happen, for instance, if a module had been moved several times since the last update of the module capability advisory field. Under these circumstances the module capability advisory field would be out-of-date, resulting in the following of forwarding addresses prior to finding the module. When such a moved module is finally located, the system updates the module capability with the new location information, enabling the open operation to proceed more efficiently next time.

As indicated in this section, an open operation on a moved module causes an increment of the MCS count field of the MOT entry for the module. This field is used to record how many processes at the node are accessing the module. As each of these processes *closes* the module, meaning that the corresponding MCS is deleted, the MCS count field is decremented. When the MCS count value becomes zero, no local processes have the module open, meaning that the MOT entry for the module may be removed from the table.

6.2.2.3 The *Open* Page Fault Handler Algorithm Using the MOT

The full sequence of steps taken by the page fault handler to determine where the root page of a module is located during an *open* operation is

(1) Compare the module name (address space number) with the equivalent field in the MOT. If a matching entry is found, use the node number and volume number in the MOT entry to access the page. If a matching entry is not found, then use the node number and volume number from the module capability advisory field, or from the page address if the advisory field is blank.

(2) Compare the node number/volume number with the equivalent field in the LMT. If a matching entry is found, this means that the volume that probably contains the page is locally mounted, and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

(3) Compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a request_page message to the node indicated in the FMT, including any module capability advisory information in the storage volume number field of the message. If the appropriate FMT entry does not exist, then

(4) Assume that the volume containing the page is still mounted on the creating node, and transmit a request_page message to the creating node indicated in the faulting address, then

(5) If the result of the read attempt is a forwarding address, update the MOT and go back to step (1).

When the root page is successfully obtained, the advisory field of the module capability used to open the module is corrected if necessary to reflect the true location of the module.

6.2.2.4 The General Page Fault Handler Algorithm Using the MOT

The full sequence of steps taken by the page fault handler to determine where a page is located is

(1) Compare the module name (address space number) with the equivalent field in the MOT. If a matching entry is found, use the node number and volume number in the MOT entry to access the page, including the node and volume numbers obtained from the MOT in the storage volume number field of the request_page message. If a matching entry is not found, then use the node number and volume number from the page address.

(2) Compare the node number/volume number with the equivalent field in the LMT. If a matching entry is found, this means that the volume that probably contains the page is locally mounted, and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

(3) Compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a request_page message to the node indicated in the FMT, setting the storage volume number field of the message to null[11]. If the appropriate FMT entry does not exist, then

(4) Assume that the volume containing the page is still mounted on the creating node, and transmit a request_page message to the creating node indicated in the faulting address, setting the storage volume number field of the message to null.

Notice that this sequence of steps involves checking the MOT for *every* page fault. In the usual case such a check will result in a negative result because, in general, a module will not move independently of the volume on which it was created.

---

[11] Since there is no MOT entry for the module (as indicated by the fact that the fault was not handled at step 1), the module is not a moved module. The null storage volume entry reflects this fact.

6.2.2.5 Discussion

The scheme described in section 6.2.2 can handle movements of modules between volumes in the MONADS DSM in a manner that is completely transparent to users. The scheme benefits from user assistance in modifying the module capability advisory field. Note that the algorithm for accessing a general page from an open module is almost the same as the algorithm for accessing the root page of a module during an open operation. The major problem with the scheme is that the MOT must be checked as part of the resolution of *every* page fault.

It is more usual for a module to remain on its original volume than it is for the module to have been moved. Recall that the MOT contains location information for modules that no longer reside on their creating volume. In most cases, then, the check of the MOT that occurs as the first step in the resolution of *every* page fault (see section 6.2.2.3) simply indicates that no advisory information exists for the module. The overhead of checking the MOT, however, applies to the resolution of page faults for *every* page whether part of a moved or a non-moved module. This means that the efficiency of page fault resolution in the typical case suffers because of the atypical case. In contrast, the scheme described in section 6.1 for accessing pages from a module that remains on its original but relocated volume incurs no additional overhead for the typical page fault, that is a page fault for a page on a locally mounted volume.

We feel that handling of the common situation should be as efficient as possible, and that efficiency certainly should not suffer because of the atypical case. Any strategy used should initially assume that the problem to be solved is of the most commonly occurring type. In the following section we describe an alternative scheme which incurs no overhead for pages of modules which have not been moved.

6.2.3 Use of an Alias in Addressing Moved Modules

The capability-based addressing scheme used in the MONADS architecture relies on the fact that a module resides in a single unique address space throughout its life. The name of this address space is

used as the name of the module, and is embedded in the module capabilities used to access the module. When a module is moved, it must retain its original name so that existing module capabilities still allow access to it. Presentation of such a module capability is only required *when the module is opened.* Subsequent accesses to the pages of an open module are achieved using virtual addresses, and do not require the presentation of a module capability. The essence of the technique presented in this section is that the identity of an open module may be temporarily altered to reflect its current location, thus allowing efficient access to the pages of the module.

The implementation of this technique for accessing moved modules requires that

(1) A new unique address space number defining the new node, volume, and (logical) within-volume address space is allocated to a moved module. This number is called the *current name* for the module, and is used for internal system purposes only. The name by which the module is known to users remains the name allocated when the module was created, meaning that all existing module capabilities still allow access to the module. The current name may be viewed as an alias for the original name.

(2) An additional table is maintained in address space zero of each volume. This table is called the *Foreign Address Space Table* (FAST), and contains mappings between module names and current names for moved modules currently stored on the volume as shown in figure 6.6. The FAST is accessed using the original module name as a key, and allows the current name for any moved module stored on the volume to be determined.

(3) The method for moving modules is modified slightly so that the volume directory entry for a moved module (as described in section 6.2.2.1) contains the current name for the module rather than the original module name. This effectively returns the volume directory to its simple form, as described in section 4.3.2.

(4) When a module is moved from a volume the volume directory of the source volume is changed to link the current name used on the source volume to a forwarding address in the same manner as described in the previous section.

(5) A new message is used to request the root page of a module during an open operation on the module. This is called the *request_root_page* message, and it contains *requesting node number*, *requested page number*, and *storage volume number* fields. The *request_page* message is returned to its simple form as described in section 5.3.1.1.

(6) A new message is used to supply the root page of a module during an open operation. This is called the *supply_root_page* message, and it contains *supplied page number*, *page data*, and *requested page number* fields.

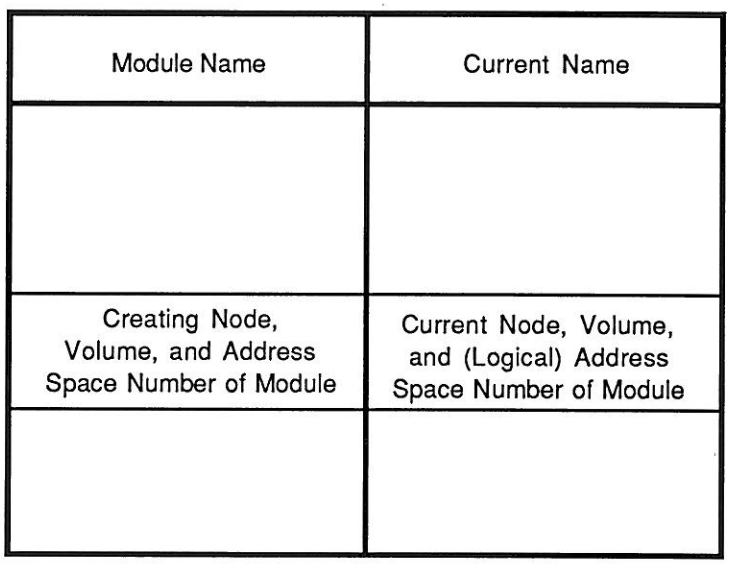


Figure 6.6. The Structure of a Typical Foreign Address Space Table Entry.

When a module is opened, the root page of the module must be accessed to allow creation of the MCS. This means that when the MCS is set up the kernel knows whether the module is stored on its original volume or has been moved to a different volume. If the module has been moved, the kernel knows its new location because it has successfully obtained a copy of the module's root page.

Efficient access to the pages of the module is achieved by altering the segment capabilities used to access the module's data as they are stored in the MCS. This alteration replaces the original node number, volume number, and address space number fields with values indicating the current node, volume, and address space numbers. Subsequent accesses to these data segments generate virtual addresses containing the *current name* rather than the *original name*, meaning that the page fault handler can obtain pages from these segments as if the module had never been moved. Since pointers to data *within* the module are relative to the address space itself, they do not need to be changed as a result of the change of address space name (see section 4.4.1).

The issue of forwarding addresses was fully discussed in section 6.2.2.1, and this discussion also applies to their use with the FAST. If the attempt to read the root page of a module results in the return of a forwarding address, the kernel

(1) updates the advisory field of the presented module capability, and

(2) repeats the read using the updated location information.

Forwarding addresses effectively form a chain which leads to the current location of a moved module. When a module is moved then either

(1) it is being moved for the first time, from the volume on which it was created, or

(2) the module has been previously moved.

When the module is moved, the forwarding address is mapped with the module name in the source volume directory. In case (1), no FAST entry for the module is necessary at the source volume because the volume directory entry for the module contains the original module name.

In case (2), the FAST entry must be retained at the source volume. This mapping from the module name to the source current name allows the module name to be associated with the forwarding address.

If all volumes in the forwarding address chain are mounted on a node in the network, this sequence will eventually find the moved module. The use of a Foreign Address Space Table (FAST) in accessing the pages of moved modules allows page faults for moved modules to be resolved efficiently whilst not increasing the overhead of resolution of page faults for non-moved modules.

6.2.3.1 The Page Server Algorithm Using the FAST

Every node with attached disk(s) acts as a page server for virtual pages stored on its disk(s). A request for a page that is not a module root page is received in a *request_page* message, and is processed as described in section 5.3.1.1.

When a server node receives a *request_root_page* message, it must check the stored volume number field to determine whether the requesting node has included module location information in the message. If the field is null, the server node attempts to find the requested page on the volume whose number is embedded in the page address is used.

To find the page, the kernel at the server node checks the LMT to see whether the indicated volume is locally mounted. If it is, then the volume directory is checked to see whether an entry exists for the module. If an entry does exist, then the requested page is either

(1) from a module that has not been moved from its original volume, or

(2) from a module which was originally stored on the indicated volume but has been subsequently moved.

In case (1) the page is supplied to the requesting node as described in section 5.3.1.1, but using the *supply_root_page* message with the *supplied page number* field and *requested page number* fields set to the same value, indicating that the module name and current name are the same. In case (2) a forwarding address is returned to the requesting node as described in section 6.2.2.1.

If the stored volume number field of the request message is not null, the kernel at the server node knows that the requested root page comes from a moved module, and the FAST for the indicated volume is checked. If no FAST entry exists for the module, then an *invalid_address_space* message is returned to the requesting node. If a FAST entry does exist, then it maps the module name to the corresponding current name. The current name is then found in the volume directory. The volume directory entry indicates either

(1) a forwarding address for the module, or

(2) the disk address of the required root page.

In case (1), the server node transmits a *module_moved* message back to the requesting node. This message provides the requesting node with a forwarding address for the module.

In case (2), the *supplied page number* field of the *supply_root_page* message is used to inform the requesting node of the current name for the module. The page number used in the request is returned in the *requested page number* field. Since the owner node has the mapping between the module and current name at hand, it does not use the *send_page* message to supply the page even if the XPT indicates that another node has a copy of the page. Rather, subject to the coherency issues discussed in section 5.3, the owner node always supplies the root page for moved modules under the FAST scheme, if necessary after retrieving an up-to-date copy from an importing node.

6.2.3.2 The *Open* Page Fault Handler Algorithm Using the FAST

The full sequence of steps taken by the page fault handler to determine where the root page of a module is located during an *open* operation is:

(1) compare the node number/volume number from the module capability advisory field, or that embedded in the original module number if the advisory field is empty, with the equivalent field in the LMT. If a matching entry is found, this means that the volume that probably contains the page is locally

mounted, and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

(2) compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a request_root_page message to the node indicated in the FMT, with any module capability advisory information included in the storage volume number field of the message. If the appropriate FMT entry does not exist, then

(3) assume that the volume containing the page is still mounted on the creating node, and transmit a request_root_page message to the creating node indicated in the faulting address, with any module capability advisory information included in the storage volume number field of the message, then

(4) if the result of the read attempt is a forwarding address, go back to step (1).

When the root page is successfully received in a supply_root_page message, the advisory field of the module capability used to open the module is corrected if necessary to reflect the true location of the module. A difference between the *supplied page number* and *requested page number* fields of the message indicates that

(1) the current name for the module differs from the original module name, and

(2) the module name embedded in the supplied page number field is an alias for the module. This alias contains current location information for the module.

If the supplied and requested page numbers are the same, this means that the module has not been moved from its original volume, and the current and original module names are also the same. As the MCS is built, the *current name* is used within the root segment capabilities for the moved module, meaning that the process may directly access the pages of the module whilst it keeps the module open.

6.2.3.3 The General Page Fault Handler Algorithm Using the FAST

The full sequence of steps taken by the page fault handler to determine where a page is located is

(1) compare the node number/volume number with the equivalent field in the LMT. If a matching entry is found, this means that the volume that contains the page is locally mounted, and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

(2) compare the node number/volume with the equivalent field in the FMT. If a matching entry is found, transmit a request_page message to the node indicated in the FMT. If the appropriate FMT entry does not exist, then

(3) assume that the volume containing the page is still mounted on the creating node, and transmit a request_page message to the creating node indicated in the faulting address.

By taking these steps the kernel is able to determine the location of pages for any module stored on a mounted volume. It should be noted that this sequence of steps is exactly the same for a moved or non-moved module, meaning that the use of the FAST technique has greatly simplified access to pages compared to the use of the MOT.

6.3 Conclusion

The described schemes can handle movements of volumes between nodes and modules between volumes in the MONADS DSM in a manner that is completely transparent to users. This transparency is achieved even though the location information embedded in virtual addresses is used to obtain pages of the virtual address space. As a result the module capability provided to the owner of a module when it was created, and any copies of the capability subsequently distributed to other users continue to enable access to such a moved module. Such movement may involve relocation of the entire volume on which the module is stored, or transfer of the individual module between volumes.

The scheme is able to transparently handle the movement of individual modules provided that forwarding addresses remain on mounted disks. With fixed disks, this will normally be the case. However, if the forwarding address is on an unmounted disk, or if the forwarding address information has been deleted, the forwarding address mechanism fails. The inclusion of an advisory field in module capabilities allows the location of moved modules without the need to rely on forwarding addresses, provided that the advisory field information is current. Such advisory information is readily available when a process first accesses the module because this first access must be accompanied by the module capability. Any subsequent access of the module is not accompanied by the module capability, meaning that the advisory information is not readily available to allow resolution of a possible page fault caused by such access. Two solutions to this problem are presented, the first making advisory information available for use in resolution of every page fault. This solution is inefficient because the advisory information must be checked as part of the resolution of every page fault, even those for which it is not needed. The second more efficient solution allows a moved module to, in a logical sense, adopt the identity of a module that had been created on its current storage volume. This change of identity allows the system to access the module pages using addresses that describe its current location whilst still allowing processes to open the module using its original name.

The MONADS DSM architecture assigns every module a unique and permanent name that includes location information for the module. The significance of the techniques presented in this chapter is that, despite the fact that this location information is essential for accessing the module, the module and/or the volume on which it resides can be moved without making the module inaccessible.

Chapter 7 STABILITY OF THE DISTRIBUTED SHARED MEMORY

7.0 Introduction

When a computer system shuts down unexpectedly due to hardware or software failure the contents of volatile memory or RAM is typically lost, whereas data stored in non-volatile memory such as disk or tape usually remains available on system restart. Since writing to RAM is much faster than writing to disk or tape, processors typically modify data in RAM. A more permanent copy of such data may be obtained by regularly copying it to disk. This is achieved at the user level by the user issuing a command such as *save* with a filename parameter, or at the system level by the system regularly calling a utility such as *sync* [123]. The result of an unexpected system shut down is the loss of modifications made to the data since it was last saved to disk.

As explained in chapter 3, the MONADS DSM provides a *persistent store*. This means that the storage and retrieval of both data and its interrelationships occurs in a uniform manner that is totally unrelated to the lifetime of the data. Uniformity of storage and retrieval of data is achieved by hiding the separation between disk and RAM storage, thus providing a *flat* virtual memory space. The transfer of pages of data between the disk store and RAM, and indeed between nodes in the network, is controlled by the MONADS system in a manner transparent to the user.

All data in the MONADS DSM resides in the persistent store, including the data used by the system itself. At any instance in time the version of such data held in RAM may differ from the version stored on disk. For example, system data held in RAM at a node typically includes parts of the address space page tables. Such data constantly changes with normal system operation, and must correctly reflect the system state as seen by the node. User data moving between RAM and disk may include segments that have not yet been saved to disk, and modified copies of pages of segments containing pointers into other segments.

At any time the true picture of the state of the store is a combination of the contents of the virtual pages held in RAM and the contents of disk

pages. Thus it is crucial to effective system management that the integrity of the store be guaranteed, particularly after an occurrence such as a system crash or a hardware failure. A store is deemed to have such integrity if, following a failure[1], it always reverts to a consistent state that existed prior to the failure. Such a store is said to exhibit *stability.*

File-based systems use utilities such as fsck [111] to restore file system integrity after an unexpected system shut down. The possible resultant loss of files, whilst potentially annoying to the user, is usually not critical to the system because such files are independent entities. The MONADS persistent store, on the other hand, contains all data including system management information and arbitrary cross references between data segments. Inconsistencies in such data are critical to system integrity and security, and can lead to problems such as incorrect volume page tables and dangling references to lost segments. These problems may well compromise the integrity of the store. In this sense the problem of failure recovery within a persistent store is closely related to recovery in database systems [7].

In this chapter, which expands on work published by the author [54, 101], we discuss, in section 7.1, the topic of stability in general. We then, in sections 7.2 and 7.3, show how to modify the single node MONADS architecture so that the store exhibits stability. Lastly, in sections 7.4, 7.5, and 7.6 we extend this stable single node architecture to produce a stable DSM.

7.1 Stability and Shadow Paging

A number of proposals for stable stores have already appeared in the literature. Many of these are based on a scheme proposed by Lorie [76]. Lorie's proposal was oriented towards recovery in database systems, and presented a new technique called *shadow paging.* The technique steps the database from one stable state to the next using a process called *checkpointing.* Between checkpoints, two versions of modified pages of the database are maintained:

---

[1] We exclude here the question of total media failure, which is a separate issue best handled by a backup or dumping strategy.

- the version of the page at the last checkpoint. This version is called the *shadow page*, and

- the *current page*, which is the shadow page plus all modifications made since the last checkpoint.

At each checkpoint, the current pages are written to disk and become the shadow pages, and the previous shadow pages are reclaimed as free disk pages. If a system failure occurs, the system reverts to the state at the last checkpoint. This state is represented by the set of unmodified pages plus the set of shadow pages.

Shadow paging has been used in a number of subsequent stable store designs. Traiger [118] and later Thatte [117] proposed that hardware support for shadow paging would result in a simpler implementation and a more efficient system. Shadow paging has been used with memory mapped files in the implementation of several stable stores. For example VAX/VMS memory mapped files were used by Ross [104] and SunOS 4 memory mapped files were used by Brown [21]. The MONADS stable store is also based on shadow paging [54, 101].

A number of techniques have been developed for achieving stability, particularly in the context of database systems. Some of these used shadow paging [21, 76, 104, 117, 118], and others were based on non-paged object mappings [10, 12, 22, 53]. Whilst such methods of achieving stability differ, the techniques have two basic features in common. These common features are:

(1) the ability to perform an *atomic update* operation, and

(2) the ability to separately identify the old data and the new data prior to the checkpoint operation.

In sections 7.2 and 7.3 we describe how these features are implemented in the *single node* stable MONADS store.

7.2 Atomic Update

The achievement of stability in a persistent store requires that, as the last stage of the checkpoint operation, the store moves from the previous consistent state to the next as an *atomic* operation. This means that the transition between stable states is conceptually a *single step* which is either taken or not taken, and that it is impossible for the system to be left part-way through the transition. The atomic operation that completes a checkpoint is analogous to the *commit* operation for database updates [34].

To achieve this atomic update of the state of the store, we use Challis' algorithm [25]. Our use of this algorithm requires that we maintain information about the state of the persistent store. In describing our use of Challis' algorithm we describe the store state information in a conceptual sense. As a result, we have ignored efficiency. In section 7.3 we describe how efficiency is achieved by avoiding unnecessary duplication.

7.2.1 Challis' Algorithm

Challis' algorithm provides a means for achieving the effect of an atomic write operation to a disk block. This is a significant achievement because writing of a block of data to a disk is typically implemented as a sequence of bit storage operations.

The key to the algorithm is the existence of a *time stamp* as the first and last words of the block. These time stamps are re-written every time a block is stored on disk, and are used to determine the correctness or otherwise of the block when reading data from it. When a block is stored on disk, the same value is written to both the time stamp words for the block.

Challis' rules for determining the correctness of a disk block are:

(1) If the time stamps are identical, then the last store operation on the block completed successfully. In this case the data stored on the block probably correctly reflects what was written to it.

(2) If the time stamps are different, then the last store operation on the block was interrupted, and did not complete successfully. In this case the data stored on the block is assumed to be corrupt.

Our use of Challis' algorithm allows the transition from the last stable system state to the next to occur as an atomic operation. Its use makes the following assumptions.

(1) There exists a mapping table from virtual persistent store addresses to physical disk addresses. Such an address map is required in systems where the virtual address space is not mapped in 1-to-1 correspondence with the physical address space. In systems such as MONADS in which data is sparsely distributed throughout the virtual address space, this mapping table is called the *disk page table.* The disk page table allows efficient utilisation of disk space because disk pages are not allocated to unused virtual pages.

(2) On system start up and after each stabilise operation a new copy of the mapping table and the data is made. Subsequent updates to the data are made to these copies, meaning that the data as it existed at the last stabilise operation (the *shadow* data) is never overwritten[2].

Two *root blocks*, with their integrity guaranteed by the use of Challis' algorithm, are stored at well known disk addresses. The most recent correct root block contains information that allows the mapping table for the last stabilised state to be found. The two root blocks usually record the locations of the mapping table for the two previous stabilised states of the system. It should be noted, however, that only the most recent version of the mapping table exists because, as part of a successful stabilise operation, the disk space allocated to the previous stable state is returned to the free disk space pool.

---

[2] This copy operation is very expensive in terms of the time taken to perform the copy and the disk space needed for the storage of the copies. The existence of two *completely* separate copies of the data allows the model to be more easily understood. In fact the two separate copies exist in a *virtual* sense only, as explained in the implementation section.

If a crash occurs during the writing of a root block then the write is incomplete and the information stored in the block is potentially incorrect. In this case only one of the root blocks records a previous stabilised state of the system, a condition which continues until the next successful stabilise operation.

A *version number* is written as the first and last word of each root block. This version number is used for two purposes

(1) to enable the system to determine which of the root blocks points to the mapping table for the most recent stable system state, and

(2) to enable the system to determine whether the root block was written correctly. The version numbers for a correct root block are the same. If the writing of a root block is interrupted by a system failure, the version numbers at the start and end of the block are different and the block is incorrect.

The use of Challis' algorithm is illustrated in figures 7.1 and 7.2. Figure 7.1 shows the system state prior to the $n+1^{th}$ stabilise operation. Root block 0 points to the mapping data for the $n-1^{th}$ stable state[3], and root block 1 points to the mapping data for the n^{th} stable state. Neither root block points to the most recent, but at present unstable system state, n+1.

The atomic update operation entails overwriting the root block with the oldest version number, in this case block 0 with version number n-1, with the new version number n+1 and a pointer to the newest mapping table. The disk space occupied by the old stabilised state n may now be reused, as shown in figure 7.2.

---

[3] The disk pages that stored this state were returned to the free disk space pool as part of the nth stabilise operation, so that the pointer contained in this block is in fact meaningless.

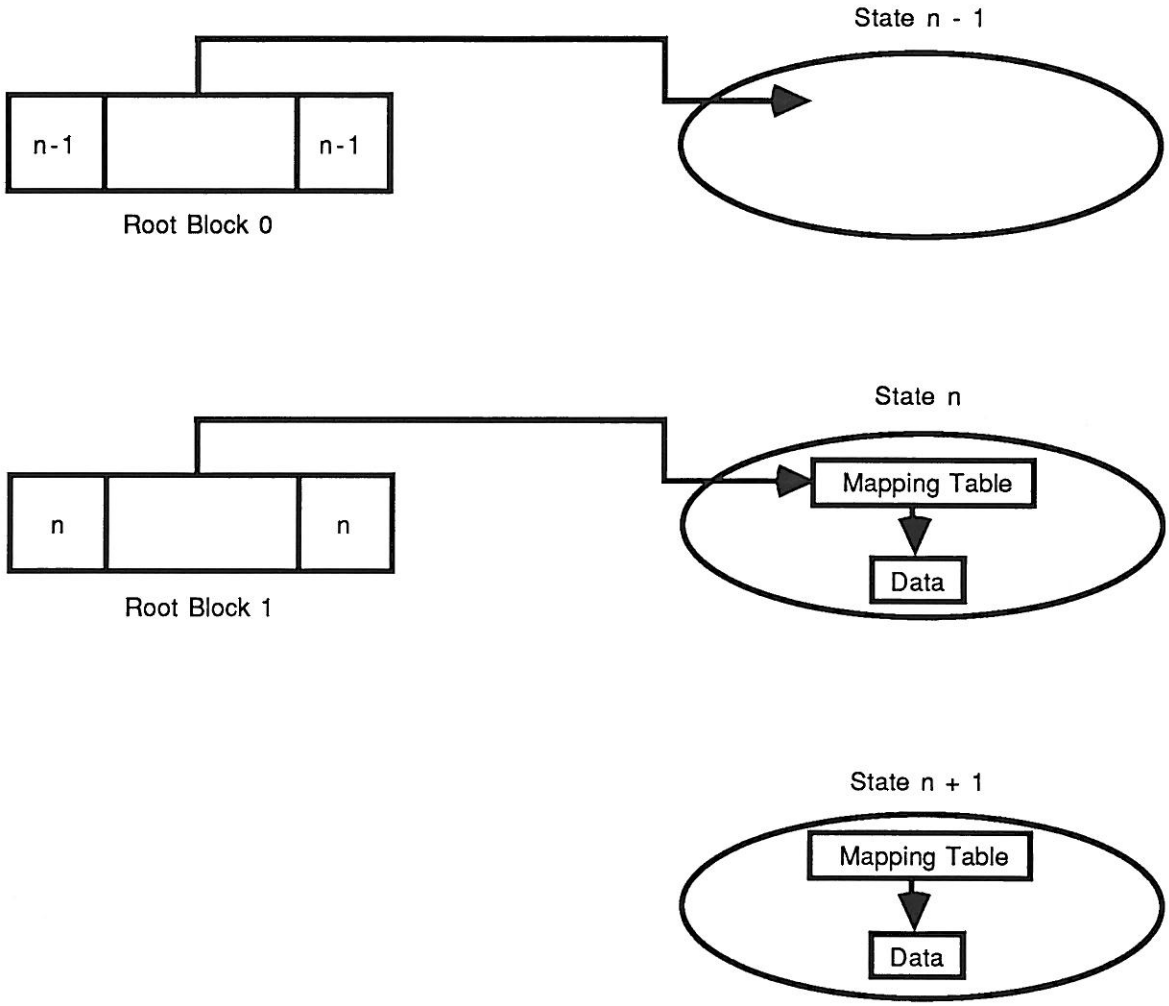


Figure 7.1 The State of the System Prior to the n+1th Stabilise Operation.

On system startup or on reversion to the previous stable state following a failure, the root blocks are inspected. If the version numbers for each root block are consistent then the most up-to-date version of the system is that pointed to by the root block with the highest version number. If the version numbers for a root block are inconsistent then this can only apply to *one* of the blocks at any time unless some catastrophic failure such as a partial head crash has occurred, in which case the integrity of the system data itself must be doubtful. Given that such catastrophes do not occur frequently, the usual case is that either *one* or *both* of the root blocks are correct. Subject to the discussion above, it is not possible for both the root blocks to be incorrect because the algorithm involves rewriting the incorrect root block or the least

recent correct root block to complete a stabilise operation. A single correct root block will *never* be overwritten as part of the algorithm.

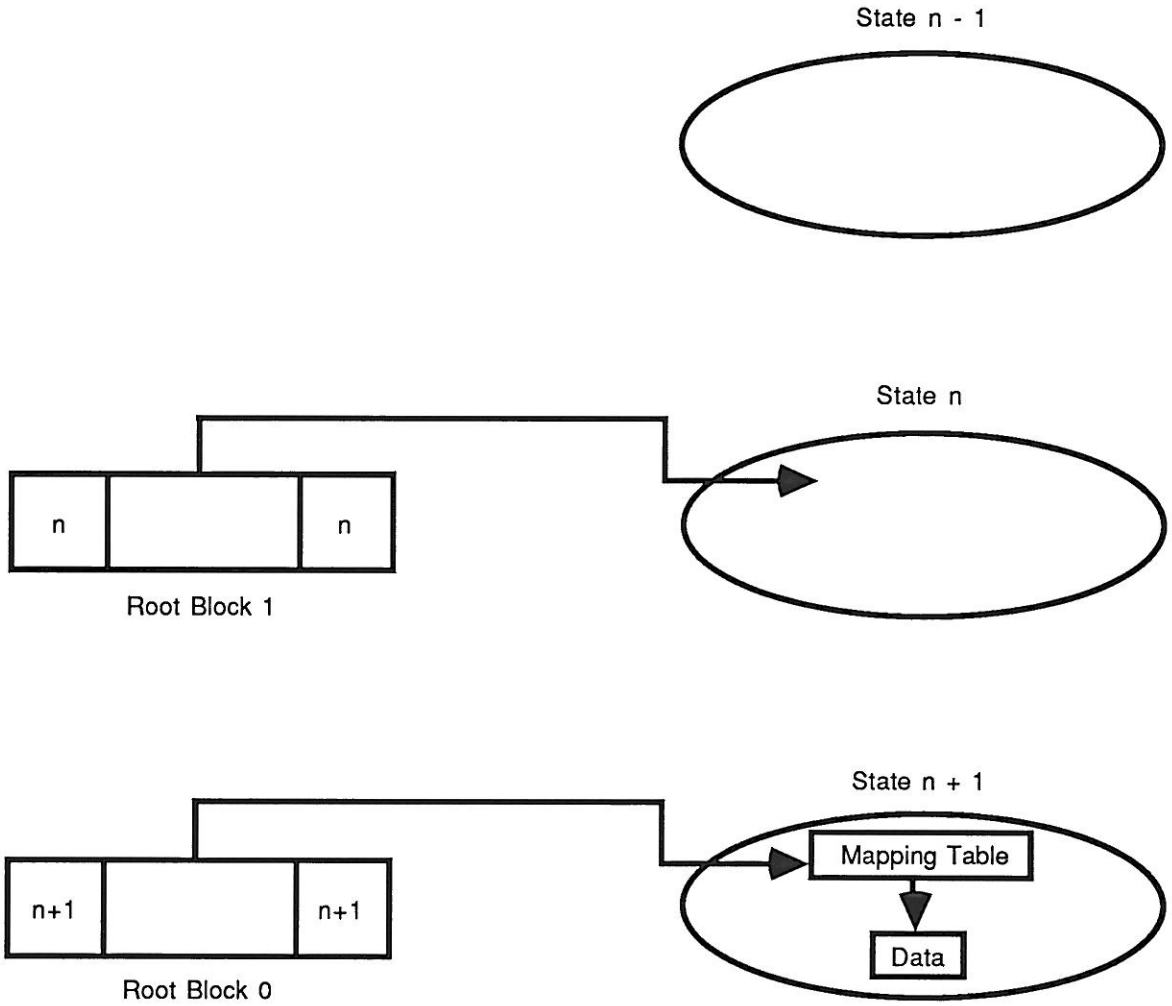


Figure 7.2 The State of the System after the n+1th Stabilise Operation.

7.2.2 Challis' Algorithm and the Single Node MONADS Store

As described in section 4.3, the MONADS virtual store is paged. Virtual pages are copied into main memory to resolve page faults, and are discarded from main memory as required to free up main memory space. Pages that have been modified are written back to disk as part of the page discard operation whilst unmodified pages may be safely discarded without writing to disk. The essence of the shadow paging technique [76] is that when a modified page is written back to disk

from main memory it is *never* written back to the place it was first read from after the previous stabilise operation. This means that the version of the page at the last stabilise operation, the *shadow* page, is not overwritten as part of page discard.

The disk locations of the pages in a module are stored in the page table for the address space in which the module resides. The address space page table exists in the virtual memory of the address space itself with the root of the page table being in page zero of the address space. The disk locations of page zero for every address space stored on a volume are stored in the volume directory. This volume directory, together with the free space map for the volume, is stored in address space zero of the volume (see section 4.3.2). Page zero of address space zero is the root of the page table for the volume directory address space itself, and is thus also the root of the page table for all pages stored on the volume. By allocating *two* disk pages for page zero of address space zero it is possible, then, to implement Challis' algorithm to achieve atomic update of state information for MONADS volumes.

In the following section we describe how old and new data are identified prior to a stabilise operation, and how the atomic update is used to complete the transition between stable states.

7.3 Shadow Paging and the Single Node MONADS Store

The description of our use of Challis' algorithm in section 7.2 requires that all data be copied prior to commencement of processing following a stabilise operation or system startup. The original data is retained as *shadow data*, and modifications are made to the copy or *current version* of the data. Each stabilise operation involves an atomic operation in which the current version becomes the shadow data, the disk space allocated to the old shadow data is released, and a new copy of the data is created for future processing. This operation is inefficient because of the requirement that *all* data be copied in preparation for potential subsequent modification. In fact it is only necessary to duplicate data when the intention to modify it becomes apparent. Much data remains unchanged between stabilise operations, and need not be duplicated.

Since the unit of data transfer between main memory and disk is the virtual page, the single node MONADS implementation of shadowing applies to the MONADS virtual page. Such an implementation has the following requirements

(1) The maintenance of a *shadow list* indicating the virtual memory pages for which a corresponding current version page has been allocated on disk. This list is *transient*, and is not part of the virtual address space.

(2) That the disk page table and the disk free space list is maintained in virtual memory so that they are shadowed, and so that modifications to them occur in corresponding current version pages.

(3) The ability to mark a main memory copy of a virtual page as read-only or read/write, and the ability to generate a write fault if an attempt is made to write to a read-only page.

(4) The ability to detect whether a main memory copy of a virtual memory page has been modified, meaning that at least one byte in the page has been written to since the page was brought into main memory.

Requirements (2), (3), and (4) are already satisfied by the MONADS architecture, as described in chapter 4. To satisfy requirement (1) we create a new table, the *Shadowed Pages Table* (SPT), which is maintained by the kernel. A separate SPT is maintained for each mounted volume.

7.3.1 The Shadowed Pages Table

Our implementation of shadow paging provides for at most *one* shadow page for any virtual memory page. If a process wishes to modify a virtual page, the kernel must determine whether the page has been shadowed since the last stabilise operation on the volume containing the page. If the virtual page has been shadowed, then a *current version* disk page has been allocated to it. If a shadow page does not exist, then a current version disk page is allocated and the old disk page becomes the

shadow page. Otherwise the existing shadow and current version pages suffice.

The Shadowed Pages Table is a transient table containing an entry for each virtual memory page that has been modified since the last stabilise operation on the corresponding volume. This table is checked every time the kernel detects that a process wishes to write to a read-only page. If an entry already exists for the page, then write access may be granted immediately. If an entry does not exist, then the page has not been shadowed, meaning that a current version disk page has not been allocated to the virtual page. It is necessary to allocate the current version disk page for a virtual page *prior* to the first modification to ensure that sufficient disk space is available to enable the writing of the modified page to disk. If free disk space is not available, the system stabilises the volume, thus freeing up pages allocated to shadow pages for the volume. If stabilising the volume does not result in free disk space, the volume is full, and an exception condition occurs.

The SPT is necessary because, although the ATU can detect an attempted write and does indicate modified pages, it only contains entries for virtual pages currently in *main* memory. Consider a virtual page which is modified and then discarded to disk. In this case a current version disk page will be allocated, and the discarded virtual page is written to this disk page. At a later stage the same virtual page may be brought back into main memory and again modified. The SPT will indicate in this case that a current version disk page has already been allocated. As a result

(1) a new current version disk page is not allocated to the virtual page, and

(2) the virtual page is read into main memory from the existing current version disk page and *not* from the shadow page,

(3) the page is mapped into the ATU with read/write access.

We will see later that the SPT is also required to manage the release of disk space used by shadow pages which are part of the previous checkpoint. At this stage we will consider the SPT to be a linear table, but will later look at alternative implementations (see section 7.3.5).

The SPT for a volume contains two values for each entry. These are the disk address at the last checkpoint for the corresponding page, called the *old disk address*[4], and the disk address to which the page will be written on page discard or at the next checkpoint, called the *new disk address.*

7.3.2 Read-only and Read/write Pages

On a write access to a read-only page an exception, called a *write-fault,* occurs. This is used to detect the first occasion on which a page is modified. This should not be confused with the *modify* bit in each entry of the ATU which indicates whether the corresponding page has been modified since it was brought into memory. The distinction is important for a situation where a page is marked as read/write immediately on being brought into the main memory, in which case the modify bit indicates if the page has been changed. This is used by the page fault handler to determine whether a discarded page needs to be copied to disk.

7.3.3 Management of Disk Pages

As described in section 4.3.2, the root page of a volume is effectively the root of a tree of disk addresses of pages on the volume. From the root page the disk address of any given page on that volume can be located. Following a checkpoint every page on the volume will either be in that tree or in the free space bit map, which itself is in a page described by that tree. The key to implementing stability is to leave that tree undisturbed and to incrementally construct a new tree. This new tree can be pointed to by the in-memory copy of the root page, leaving the disk copy of the page pointing at the checkpoint version. Provided that none of the checkpoint pages is modified then, following a system crash, the system will return to the last checkpointed state without any processing being required. The new state described by the in-memory root page can be made the checkpointed state by a single disk write of the root page. In order to ensure that this write is atomic

---

[4] This is in fact the disk address of the shadow page.

two root pages are maintained and Challis' algorithm, described earlier, is employed. Both of the root pages are placed at well known disk addresses so that they may be located at system start-up.

7.3.4 Operations on the Stable Store

There are six operations on the shadow paged single node store that concern us. They are:

(1) *Create a new page.* To create a new page a new disk block is allocated using the volume free space bit map. If there are no free blocks then a stabilise must be initiated[5]. An entry for this page, containing a null old disk address (since this is a new page, no shadow page exists) and the new disk address, is added to the SPT. The new disk address is inserted into the MMT entry and the page table entry for the page is updated. If this is page zero of an address space then an entry is added to the volume hash table.

(2) *Modify a page in main memory.* The effect of modifying a page in main memory depends on the read-only bit for the corresponding entry in the ATU. If the page is marked as read/write then the modify bit in the ATU is set and the access proceeds, otherwise a write fault exception occurs. In this latter case the original disk address of the faulting page is obtained from the MMT. A new disk block is allocated using the free space bit map. If there are no free blocks then a stabilise must be initiated. An entry for the page containing the original disk address (the shadow page) and the new disk address (the current version page) is added to the SPT. The MMT is updated with the new disk address and the page table entry for the page

---

[5] Given the page table structure it is possible for the page fault handler to statically calculate the maximum number of pages which may be modified as a result of processing a write fault and to ensure that there is sufficient disk space for each of these pages before granting write access to the page. This number of pages never exceeds five in the MONADS scheme (see section 4.3.2).

is updated. If this is page zero of an address space the hash table is updated with the new disk address.

(3) *Page fault.* On a page fault the disk address of the required page is obtained from the disk page table and the page is read into a free page frame. This may involve resolving further page faults The disk address is looked up in the new address column of the SPT. If it is found then the page has already been shadowed, and thus it may be mapped into the ATU as read-write, otherwise it is mapped in as read-only. In either case an entry containing the disk address of the page is added to the MMT.

(4) *Page discard.* On a page discard, that is when a page is removed from memory, if the modify bit for the corresponding entry in the ATU is set then the page is written to the disk address indicated in the MMT. Note that if the page has been modified rules (1), (2) and (3) will guarantee that a new disk block has already been allocated.

(5) *Stabilise a volume.* A stabilise operation for a volume may either be automatically generated (for example to free up disk space on the volume) or explicitly requested by a user/program. In either case the following must be performed. For each entry in the SPT the old disk (or shadow page) address is extracted and the corresponding bit in the free space bit map is set, marking the disk pages as free. The free space bit map is locked into main memory so that this access cannot cause a page fault. All pages for this volume which are in main memory and which have been modified are copied back to disk. These can be easily located using the MMT and the ATU[6]. The pages are then marked as read-only in the ATU so that, if they are subsequently modified, a new current version disk page will be allocated. Finally the root page for the volume is atomically written back to disk. This

---

[6] Note that the order in which these are written to disk is not important since if there was to be a system crash, the old state described by the old root page on disk would be restored and thus all of the current version disk pages allocated as part of the new state would be in the old free space bit map.

final step makes the new state the stable state. At this point the SPT is cleared and the checkpoint operation is complete.

(6) *Restore the persistent store.* A restore operation takes place following a crash of a volume. Since the entire state as at the last checkpoint still exists on secondary storage, and all disk pages used since that checkpoint will still be in the free list of that checkpoint state, no modification to the secondary store needs to be performed in order to restore to the last checkpoint. The SPT for the volume is cleared and any pages from that volume in main memory must be removed from the ATU. The root page from the last checkpoint is then retrieved and system operation may continue.

7.3.5 Implementation of the Shadowed Pages Table

We now return to the question of implementation of the SPT. There are three operations which must be performed on the SPT. These are:

(1) insert a new entry,

(2) check if an entry with a particular new disk address is in the table, and

(3) cycle through each entry in the table.

We can suggest two alternative implementations. The first is a hash table, using selected bits of the current version disk address as the hash key. This would provide good performance on all of the required operations. However, the size of the hash table is a potential problem since, in theory, it can grow quite large, with entries for half the number of disk pages on the volume. However, in practice this is not likely to be a problem since it is sensible to checkpoint frequently, meaning that many less than half the number of pages on the disk would be shadowed between checkpoints. In any case, if the hash table became full a checkpoint could be forced. Since a checkpoint can be performed at any time it is possible for the system to enforce a policy on checkpoints to avoid this situation, for example checkpoint after n pages have been modified.

An alternative implementation of the SPT is to use two bit lists, both of which have one bit for each disk page on the volume. For the maximum size MONADS volume of 256 megabytes, this is only 8 kilobytes each (see section 4.3.2), which it is feasible to lock into main memory. The bit lists effectively correspond to the two columns of the SPT, the first indicating whether the corresponding disk page currently stores a shadow page, and the second indicating whether the corresponding disk page is allocated to a current version page. This allows the three required operations to be performed efficiently and in a fixed amount of store. In fact these bit lists operate in a similar manner to Lorie's MAP and shadow bits [76], but have the advantage that, since they are not distributed throughout the page tables, they may easily be cleared following a stabilise operation.

7.3.6 Multi-volume Stabilise

As described in section 4.4.1, MONADS segments are grouped together into information-hiding modules, each of which resides in a unique address space. Each MONADS process is represented by a *stack address space* (see section 5.4). Segments within a stack address space may contain pointers into other address spaces, including address spaces stored on other volumes. Such pointers constitute cross references between volumes, and independent checkpointing when these exist could result in inconsistencies following a crash. A crash could, for instance, result in a capability pointing to a non-existent segment. To cater for between volume references, we provide a *multi-volume stabilise* mechanism, implemented as a two phase commit.

Two copies of the root page for each of the dependent volumes are maintained as before, and one of the volumes, presumably on a fixed disk, is designated the *master volume*. The master volume records which root page is used for each of the dependent volumes. Each dependent volume is stabilised as above, with only the older of the two root pages being updated with the new timestamp. The master volume is updated last, with both root pages being written to guarantee that the write is successful. Following a crash the timestamps can be inspected to determine the most recent consistent state.

A potential disadvantage of this scheme is that the set of dependent volumes must be stabilised at one time. For a large dependency graph with many volumes this could become quite expensive since all processes accessing those volumes must be stopped during the stabilise. However, the situation is not as bad as it at first seems. Much of the work takes place in parallel with the normal operation of the system as part of the page discard task, and at most the entire memory of the machine, but usually much less, must be copied to disk at a checkpoint.

This scheme can be generalised to allow for a very flexible stable store in which volumes are stabilised in groups in such a way that the groupings may be changed as required. For example, it may be that a particular volume supports a self-contained related group of users and their data. In such a case that volume could be stabilised by itself. Given an appropriate mechanism it would then be possible to group that volume with another so that the two are stabilised together. As another example consider bringing a volume from another site and mounting it on a machine. It may be desirable for it to be stabilised with other volumes on that machine. This can easily be achieved by the proposed scheme.

7.3.7 Processes

Processes can be included in such a scheme by saving the current state of each process, including the contents of screen buffers, etc., before commencing the checkpoint operation. At restart this state information can be retrieved and the processes continued. The process state information could either be saved on the individual process stacks or in a central object pointed to by the master volume.

7.3.8 Discussion

Persistent systems have the potential to provide a powerful and flexible software development environment. However, if they are to achieve that goal they must be both efficient and robust. We have addressed the former issue by providing a purpose-built architecture specifically designed to support a large virtual store. This architecture was

described in chapter 4. In this chapter we have described a scheme to make this virtual store stable.

The scheme is based on shadow paging but has the advantage that disk space allocation is fully dynamic. A minimum of disk space is used. At any time there are at most two copies of any page on disk, the last checkpoint version and the current version if the page has been modified. Following a checkpoint, there is only one copy of each page. At no time do pages have to be copied, either in memory or on disk. Following a system crash the system automatically returns to the last consistent state with no post-processing of the disks. The checkpoint process can be expensive, but much of the work may be overlapped with the normal operation of the system.

The scheme gains simplicity through two techniques. The first is the maintenance of all of the virtual memory tables, free space bit maps, etc. within the store. This allows the shadowing technique to be used recursively on the page tables themselves, considerably simplifying the implementation. The second technique is the use of very large addresses. This allows the virtual address space to be partitioned to support multiple volumes, without fragmenting the primary or secondary store.

The effects of the above scheme on the placement of pages on the disk warrants consideration. After some period of time the pages of an address space may be randomly distributed across the disk. This is acceptable if the pages are to be randomly accessed. However, for sequential access it would be better if the pages were physically sequential. This was achieved in Brown's scheme [21] by creating a pre-copy of pages on disk and overwriting the original page in place, maintaining the original physical structure of the store. This has two disadvantages. First, each modified page must be physically copied and second, the store must be partitioned into two areas, store and shadow pages, potentially reducing the disk space utilisation. However, it is desirable to support efficient sequential access.

Lorie [76] has suggested a solution to this problem. The disk, or volume in our case, is organised into physical clusters. Each cluster consists of a set of disk blocks such that the head movement time between blocks

in the same cluster is much smaller than the head movement time between blocks in different clusters. Each address space is associated with a cluster and when a new disk page is required for an address space it is allocated in this cluster, if possible. By careful choice of the cluster size it should be possible to achieve good locality for sequential access. Clustering need not be implemented globally, but can be an option on an address space basis.

In several of the schemes described in the literature [21, 117] there is a disk space overhead, even following a checkpoint operation. In the proposed scheme disk space is allocated fully dynamically. There is no static division of store into shadow and main store and once a stabilise has taken place all shadow store is immediately released. By implementing a clustering scheme as described above, this improvement can be achieved without serious performance degradation for sequential access.

There is an overhead in terms of both disk space and execution time in performing the shadow paging algorithm. It is quite likely that for certain address spaces containing temporary objects stability is of no importance. In these cases it is desirable to disable the shadowing. The MONADS scheme can be enhanced to support this option. Each address space can be flagged as either shadowed or non-shadowed. In the latter case the page table and red-tape information would still be shadowed, but not the data. The page table and red-tape information must be shadowed in order to ensure the integrity of store management data.

An interesting area for further research is the use of an *uninterruptable power supply (UPS).* Technology in this area has improved considerably and it is now quite possible to provide battery backup to ensure maintenance of power to disks and memory for an extended time, at least in the order of hours [32]. Given this sort of technology the question of coping with power failure is no longer an issue. Following a power failure all data can simply be copied to disk. However, this does not cope with the situation of a system software failure or, even more seriously, a hardware failure (e.g. processor error) where the power to processor and memory must be removed in order to rectify the fault. These situations will still require another mechanism such as shadow paging. A future research direction is the investigation of possible

simplifications and improvements to the proposed mechanism based on the use of a UPS. In particular it should be possible to considerably reduce the I/O overheads by shadowing within main memory.

7.4 Network-wide Stability

In this section we show how the single node stability scheme presented in sections 7.2 and 7.3 may be extended to achieve stability of the entire DSM.

Checkpointing a volume in the DSM may involve recovering modified pages from remote nodes. This introduces the potential problems of remote node failure and connecting medium failure. In this section we modify the single node stability scheme to achieve stability of the DSM, and discuss strategies for recovering from network failures.

Exported read-only pages pose no threat to the stability of the volume on which they are stored provided such pages have not been modified since the last checkpoint[7]. Stability of pages for which the current version is currently in the main memory of a remote node is not ensured by the single node stability scheme[8]. This is because such pages are not in the physical memory of the owner node at the time of initiation of a checkpoint operation, and are thus not detected by simply scanning the ATU for modified pages from the appropriate volume.

The first stage in ensuring network-wide stability requires the owner node to send return_page messages to any remote nodes marked in the XPT as holding the current version of modified pages from the volume about to be stabilised. As each page is returned it is marked as read-only in the remote node's ATU in accordance with the coherence algorithm (see section 5.3). When the page arrives at the owner node, it is stored in one of the pool of main memory page frames allocated by

---

[7] Recall that a remote modifying node may be instructed to provide a copy of a page to another remote node, resulting in exported modified read-only copies of the page (see section 5.3.1.1). In this case the owner node may not have an up-to-date copy of the page.

[8] This situation occurs if the page has been modified by the remote node since the last checkpoint on the volume containing the page.

the kernel for receipt of incoming messages. Once the transfer of the virtual page into main memory is complete, it is mapped into the ATU as modified read-only and an entry is made for the page in the MMT. Part of the MMT entry is the disk page to which the virtual page will be written on checkpoint, so a new disk page must be available for completion of the entry.

7.4.1 Allocating Disk Space for a Remote Read/write Page

As described in section 7.3, the lack of free disk space on a volume is one of the reasons for initiation of a checkpoint. It is clearly unsatisfactory to assume that free disk space will be available for storage of modified pages returned by remote nodes. The solution is to allocate the new disk page to an exported read-write page *prior* to the granting of read-write access. This is analogous to allocating a disk page on a write fault for a local page.

When a request_changed_access_rights message is received, the kernel at the owner node checks the SPT to determine whether the page has been shadowed since the last checkpoint on the volume containing the page. If the page has been shadowed, a new disk page already exists and the read/write access may be granted. If this is the first write fault for the page since the last checkpoint, a new disk page must be allocated before read/write access is granted. At the same time

(1) an entry for the page is added to the SPT,

(2) the disk page field of the XPT entry for the page is changed to indicate the new storage location, and

(3) the new disk page address is entered in the address space page table as the storage location for the virtual page.

In this way we guarantee that space exists on the disk to save the virtual page during the next checkpoint.

The single node stability scheme resolves page faults by bringing pages from disk into main memory with read/write access if an entry for the page already exists in the SPT. This is not appropriate if the node is

networked and a copy of the page exists in the physical memory of another node, because the page coherence algorithm does not allow a read/write page to appear in the physical memory of multiple nodes simultaneously. As described in section 5.3.1.1, the XPT is checked when a page fault occurs as part of the page coherence algorithm. The method used to resolve the page fault depends on page status information read from the XPT, as follows.

(1) If a remote current version copy of the page exists, a read-only copy of the page will be provided by the current version node indicated in the XPT, thus making disk access at the owner node unnecessary. If the current version node has read/write access to the page, this access is reduced to read-only before the page is transmitted.

(2) If an exported read-only copy exists a read-only copy of the page is provided by one of the read nodes in the XPT, thus making disk access at the owner node unnecessary.

(3) If the page fault is remote, and no exported copies of the page exist (indicated by the lack of an XPT entry for the page), a read-only copy of the page is provided by the owner node. This implies that, at the time of provision, the owner node has a copy of the page in its main memory.

In each of these cases the page is mapped into the ATU of the requesting node as read-only in accordance with the coherence algorithm.

If the page fault is at the owner node, and no entry for the page exists in the owner node's XPT, then the page is brought into main memory from local disk. Since the owner's XPT has no entry for the page, it is not in the memory of any other node in the network, and so it can be safely mapped into the ATU with read-only or read/write access depending on the corresponding SPT entry (according to the rules defined in section 7.3.4).

The page coherence algorithm (see section 5.3) does not allow a read/write page to appear in the physical memory of multiple nodes

simultaneously. Thus no MMT[9] entry can exist at the owner node for an exported read/write page. When such an exported page is returned to its owner node, the page must be entered into the MMT, which involves knowledge of the corresponding disk page number. The disk page number for exported pages is stored in the XPT, which is held in locked-down memory in the kernel of the system (see section 5.3.1).

The use of return_page messages and the allocation of a new disk page prior to granting of remote read/write access allows correct operation of checkpointing across the network with little change to the single node stability scheme provided that modified versions of pages are always available at the time of a checkpoint operation. In the following section we discuss the problem of checkpointing when such a page is not available.

7.5 System Failure

A modified page may not be available at the time of a checkpoint operation because of either

(1) the failure of an importing node, or

(2) the failure of an exporting node, or

(3) the failure of the interconnecting media.

In the following sections we examine each of these cases separately.

7.5.1 Failure of an Importing Node

As described in section 7.4, the volume checkpoint operation results in the owner node requesting the return of all exported modified pages. If an importing current version node fails, it obviously cannot respond to such a request. It is clear that any modifications to pages stored in the physical memory of a node are lost if the node fails, so examination of

---

[9] As explained in section 4.3 the MMT describes, for each main memory page frame, the disk storage location for the frame, that the frame contains an imported page, or that the frame is unoccupied.

importing node failure reduces to analysis of the behaviour of the exporting node. In our analysis we presume, for simplicity, that a single page has been imported and modified by the failed node. If multiple pages are affected, the described single-page scheme may be applied to each page individually.

After several attempts to retrieve the page the exporting node concludes that the importing node has failed, and that modifications to the exported page are lost. It is tempting to simply

(a) remove the SPT entry for the page,

(b) appropriately modify the volume and address space red-tape information thus making the shadow page the current page and returning the previously allocated new disk page to the volume free list, and then

(c) continue the checkpoint operation as if the lost page had never been exported or modified.

This proposition is unsatisfactory if the following sequence of events relating to virtual pages *X* and *Y* occurs after the most recent checkpoint operation:

(1) pages *X* and *Y* are transferred to a remote node with read-write access.

(2) a segment is created in page *Y*, and pointed to by a segment capability in page *X*.

(3) as part of the normal page discard process on the importing node, page *X* is returned to the owner node.

(4) the importing node fails, and subsequent to that the owner node attempts to stabilise the volume containing pages *X* and *Y*.

If the above sequence of events were to occur, and the stabilise proceeded without the inclusion of the modified version of page *Y*, then page *X* would continue to contain a reference to the now non-existent segment that had been created in page *Y*. The problem is potentially extremely serious from a security point of view because a new segment

may subsequently be created in the same location in page *Y* as that pointed to by the segment capability in page *X*. This would allow access to a segment in violation of the capability protection scheme that is designed to protect it.

In order to maintain full consistency and security it is essential that only a consistent set of pages be saved at the time of a checkpoint. If this is not possible because a node containing a page from the volume does not respond, then the volume cannot be stabilised at that time. The four available options are to:

(1) stabilise without including the unavailable page, or

(2) not stabilise at this time, or

(3) delay the stabilise until the non-responding node returns the necessary page, or

(4) revert to the previous checkpoint state.

Option (1) can only be allowed if it can be determined that excluding the missing page would not result in a dangling reference. This means that it is necessary to ascertain that the missing page contains no reachable segments that do not also exist in the shadow page. Such a determination can only be made by computing the transitive closure for the volume being stabilised. In general, this would be prohibitively expensive, because all pointers must be followed to ensure that they refer to existing segments.

Option (2) may not always be possible because the stabilise may be essential before system operation can proceed. This may be necessary if

(a) the stabilise operation was instigated because of lack of disk space, or

(b) the stabilise was required as part of the implementation of a higher level transaction mechanism.

Option (3) must be subject to time-out since operation of the owner node on the volume cannot proceed until the stabilise operation has completed, and the remote node may be unavailable for an extended

period. This option has the advantage that it allows for temporary and easily rectifiable problems such as the breaking of the network to insert an extra node.

We propose that option (3) be implemented with a parameterised time-out period. If the time-out is reached the system concludes that the stabilise cannot take place and option (4) occurs. This means that after the time-out the volume reverts to its last checkpoint state. Reverting to the last checkpoint state involves instructing all contactable nodes with pages from the volume to invalidate these pages. The importing nodes can be easily identified by reference to the XPT. In addition, pages from the volume in the memory of the owner node must also be invalidated, these pages being identified via the MMT.

The reversion to the last checkpoint state may not be as serious as it at first seems. It is expected that there will be a higher level transaction mechanism for controlling concurrency and serialisability. Such a mechanism could well provide a transaction log on a separate independently stabilised volume to allow a roll forward from the reverted state, thus recovering at least some of the lost modifications to the volume. Initial work on such a mechanism is described in [20].

7.5.2 Failure of an Exporting Node

If an exporting node fails, it will restart at its last stable state, with any local modifications performed since that checkpoint being lost. It is interesting to consider the fate of a read-write page, *X*, exported by the failed node since the last checkpoint. Attempts by the importing node to page out *X* are unsuccessful, and *X* is effectively trapped in the physical memory of the importing node. Such a situation would be detrimental to performance at the importing node because its available physical memory would be reduced by the page size for each trapped page, this situation being potentially critical for a diskless node.

A solution is to use up/down protocol messages to periodically monitor the status of the owner of imported read/write pages. If the owner of page *X* is found to be non-responding, page *X* is immediately marked as read-only, thus preventing further modifications, and an attempt is made to return a copy of the page to its owner. If the attempted return

fails, the page is invalidated, thus losing any modifications to the page in exactly the same way that post-checkpoint modifications at the owner node are lost following a crash. In a similar fashion, on the detected failure of an exporting node, all read-only pages from that node are invalidated.

If an exporting node suffers a system failure, it will subsequently be restarted at its last checkpoint state. It is possible that other nodes have pages from volumes on the restarted node in their memory. Regardless of whether these pages are read-only or read/write, they may not be consistent with the checkpoint state to which the owner node has reverted. It is therefore imperative that any pages previously exported by the restarted node be invalidated at remote nodes. As described in the previous paragraph this will happen automatically if an importing node detects the exporting node's failure. However a short-term crash may not be detected because the importing node may not attempt to communicate with the crashed node while it is down.

The solution is to link the stability scheme to the up/down protocol described in section 5.2.3. When a node restarts, it broadcasts a here_i_am message informing all other nodes that it is back on-line. When the kernel at a remote node receives such a message, it checks its IPT, and invalidates all pages previously imported from the newly on-line node. Once any such pages have been invalidated, each remote node returns a here_i_am_too message informing its existence, and the node names are stored by the restarted node in its Network Addresses Table (NAT, see section 5.2.3). Any request for provision of a page to a remote node is refused if that node does not appear in the local NAT and the requesting node is again instructed to invalidate all pages imported from the node by the transmission of a here_i_am message. The receipt of an for this instruction results in the requesting node being inserted into the table. We thus guarantee that all inconsistent versions of pages will eventually be invalidated.

7.5.3 Failure of the Interconnecting Media

The effect of failure of the interconnecting media is the same as the failure of an exporting or importing node in that pages cannot be

transferred between the nodes. The timeout described in section 7.5.1 will handle the situation of a media interruption of duration less than the time-out period. A longer term media problem is equivalent to a node failure, and is handled using the techniques described in sections 7.5.1 and 7.5.2. To ensure that exported pages are invalidated after a long media failure, a node removes the NAT entry for any remote node whose response to an invalidate_page message is not received within the time-out period. Any subsequent request from such a remote node receives a here_i_am message in reply as described in section 7.5.2.

There must be flexibility in the setting of the time-out periods to allow for different physical network implementations and for fluctuations in network traffic. It would also be sensible to provide a breakout facility to force a restart before the end of a time-out.

7.6 Multiple Volume Stabilise Across Nodes

As was described in section 7.3.6, it is possible to have cross references between volumes. These are always held in the form of segment capabilities stored in stack address spaces. In order to ensure consistency it is essential that volumes containing cross references are stabilised together. In a network, these volumes can exist on different nodes. As part of the control of between-object references, a dependency graph must be maintained at each node to describe volume inter-relationships. Various protocols for constructing and maintaining the dependency graph are being investigated. The problem is considerably simplified in the MONADS architecture because of the clear distinction between within-address-space references and references to other address spaces, potentially on other volumes.

When a volume is stabilised, the dependency graph is consulted, and if cross references exist, what is essentially a two phase commit is used to cause all the related volumes to be stabilised together. The stabilise may involve volumes stored on more than one node. In principle the multi-volume stabilise described in section 7.3.6 may be utilised across the network. One of the involved nodes acts as a *master*, ensuring that each of the separate volume stabilise operations is completed and then performing the second phase of the two phase commit. Future work on

this topic involves consideration of issues such as selection of a master node, and strategies for handling failure of the master node.

7.7 Conclusion

Stability can be achieved for a single node persistent store by the use of shadow paging. The large virtual memory store can be extended to encompass a network of nodes, with the physical location of objects within the network being totally transparent to the user.

By extending the mechanisms used to achieve single node stability, we can ensure network-wide stability. Such stability is achieved with a minimum of network traffic overhead, and uses existing memory coherency protocol messages. In achieving stability, the service offered to the remote user is equivalent to that offered to local users, in that, at worst, modifications made since the last checkpoint for a volume on a failed node are lost.

The proposed scheme guarantees the security of data within the system. The use of time-out periods means that temporary interruptions to the physical network media can occur without loss of data. Where cross references exist between volumes, the volumes may be stabilised together utilising a two phase commit protocol over the network. When combined with an appropriate higher level transaction mechanism, it should be possible to roll forward to recover most modifications made to a volume between the last checkpoint on the volume and a system failure.

Chapter 8 IMPLEMENTATION

8.0 Introduction

The MONADS DSM model described in chapter 5, and the volume and module movement scheme described in chapter 6 have been implemented for a network of three MONADS computers. The stability techniques described in chapter 7 have not yet been implemented. It is the author's intention to further investigate these techniques as the subject of future research. In this chapter we describe the structure of the MONADS kernel, and how the kernel implements the DSM.

8.1 The MONADS Kernel

The MONADS architecture is implemented using a 32 bit micro-programmed computer called the MONADS-PC [100]. The microcode, which is described in [119], implements the MONADS-PC instruction set [97].

The kernel, which sits directly above the MONADS-PC hardware, is implemented as a set of co-operating processes each of which performs a specific kernel task. These processes are written in MONADS assembly language [96]. The kernel code and data is locked down into main memory, meaning that a page fault cannot occur on access to an address in the kernel.

The kernel processes have static priority, and are scheduled by a simple priority-based process scheduler. This scheduler is based on the *automatic scheduler* described in [66], and is achieved through the microcoded implementation of four kernel process scheduling assembly language instructions. These instructions are

(1) *ksusp*. This instruction suspends the currently active kernel process.

(2) *kactp(process number)*. This instruction activates a kernel process suspended with ksusp.

(3) *ksemp(semaphore name).* This instruction implement's Dijkstra's *P* operation [41]. It indivisibly decrements the indicated semaphore's counter, and if necessary suspends the kernel process.

(4) *ksemv(semaphore name).* This instruction implements Dijkstra's *V* operation [41]. It indivisibly increments the indicated semaphore's counter, and if necessary causes a suspended kernel process to be activated.

Each hardware interrupt is individually and statically associated with a kernel process which acts as an interrupt handler for it. On a hardware interrupt the microcoded scheduler performs a *kactp* on the corresponding process, which it is assumed will be suspended.

Kernel processes belong to one of three categories. Each kernel process is either

(a) A *Server Process.* The sole role of such a process is to service requests from other processes. A server process has no associated interrupt, and never directly interacts with hardware devices.

(b) A *Synchronous Device Process.* Such a process interacts with other kernel processes and with hardware devices. The interaction with hardware is of a cause and effect nature, meaning that hardware interrupts are totally predictable and synchronous with the operation of the process.

(c) An *Asynchronous Device Process.* Such a process interacts with other kernel processes and with hardware devices. This interaction is of an essentially random nature, and it is not possible to predict whether the next activation of the process will be to service a hardware (i.e. interrupt) or software request.

Communication between kernel processes is achieved using *kernel message blocks.* Every kernel process has an associated queue onto which these message blocks may be linked. If kernel process *A* wishes to request some task of kernel process *B*, it builds an appropriate

kernel message block and links it onto process *B*'s message queue. It then signals the request by either

(1) performing a *V* operation (using *ksemv*) on process *B*'s queue semaphore if process *B* is a server or synchronous device process, or

(2) using the *kactp* assembler instruction if *B* is an asynchronous device process.

On completion of the requested task, kernel process *B* notifies the requesting process *A* by

(1) modifying the message block,

(2) linking the message block to process *A*'s message queue, and

(3) notifying process *A* using either *ksemv* or *kactp* as appropriate.

The method of scheduling kernel processes depends on the process category. With respect to the categories above, scheduling is achieved by

(a) *Semaphores only* for server processes.

(b) A combination of *hardware interrupts and semaphores* for synchronous device processes.

(c) A combination of *hardware interrupts and software interrupts* for asynchronous device processes.

The algorithms for the operation of the three categories of kernel process are shown in appendix 3.

User processes reside above the kernel, and are scheduled separately from the kernel processes by a kernel process called the *user process scheduler* which is dedicated to that task. User processes see the kernel as a set of modules with interfaces that provide kernel services. These services are accessed using module capabilities in exactly the same way as services offered by user modules. This means that an operating system built above the kernel may control use of the various

features of the kernel by controlling the distribution of the module capabilities used to access these features. Examples of services offered by kernel interface modules include terminal input/output and virtual memory management.

To gain access to a kernel service, a user process calls the appropriate kernel module interface procedure. The interface procedure

(1) builds an appropriate kernel message block,

(2) links the message block to the appropriate kernel process queue,

(3) notifies the kernel process using *ksemv* or *kactp*, and

(4) requests the user process scheduler to suspend the user process.

On completion of such a request, the kernel process ascertains whether the request originated from a kernel or user process. If the request originated from a user process, the kernel process notifies the completion by requesting the user process scheduler to reactivate the user process. If the request originated from a kernel process, it is notified as described above.

The pre DSM kernel processes and their purpose are described in section 8.1.1. In section 8.1.2, we describe in overview the modifications to the kernel structure which support distribution of the virtual memory. A more detailed description of the implementation of the DSM then follows.

8.1.1 The Pre DSM MONADS Kernel Processes

In this section we describe the kernel processes as they existed prior to the implementation of the DSM. The original MONADS kernel processes were

(a) *disk process.* This synchronous device process provides disk oriented functions such as mount, read disk page, and write disk page.

(b) *page fault interrupt process.* This server process is activated whenever a user process page fault occurs. It requests the virtual memory process to load the faulting virtual page into main memory. If there is already an outstanding request for the subject virtual page, the page fault interrupt process links this latest request to the existing one. It then suspends the user process which caused the page fault and invokes the user process scheduler.

(c) *virtual memory process.* This server process manages the use of main memory, and the movement of virtual memory pages between disk and the main memory. It maintains the Main Memory Table (MMT, see section 4.3), and implements page discard to ensure that main memory page frames are available as needed.

(d) *logon process.* This server process allows users to log on to the computer. It generates a login prompt, waits for user input, verifies the username, and requests the user process scheduler to activate the user's process. Terminal input and output is achieved by making appropriate requests to the terminal process.

(e) *timer process.* This asynchronous device process is regularly activated by an interrupt generated by the clock chip. When activated it calls the user process scheduler to activate the next "ready" user process.

(f) *user process scheduler.* This server kernel process handles user processes. As such it can start a user process at login, long suspend a user process at logout, suspend a user process on a page fault and activate it again when the fault is resolved, suspend and activate processes on an event, suspend and activate processes in general, reschedule processes, and handle exceptions.

(g) *terminal process.* This asynchronous device process handles input and output from terminals. In this role it communicates

with the logon process, the terminal interface procedures, and the serial controller hardware.

(i) *idle process.* This process is executed when all other kernel processes are inactive. It executes at the lowest priority level, and can be used to perform various house-keeping operations.

All kernel processes except for the idle process suspend themselves waiting until there is some work for them to do. The idle process is never suspended. It is simply interrupted whenever another process is scheduled to run.

The kernel maintains heaps of the various kernel message blocks used for communication between the kernel processes. These message blocks are

(a) *virtual memory message block* used in kernel message passing related to the management of virtual memory pages,

(b) *user scheduler message block* used in kernel message passing related to user process management, and

(c) *terminal message block* used in kernel message passing related to terminal input/output.

Of these, the format and use of the virtual memory message block is extended in the implementation of the DSM. The format of the extended structure is shown in appendix 4.

8.1.2 The DSM MONADS Kernel Processes

The implementation of the MONADS DSM involves modifications to the virtual memory process and the user process scheduler, and the creation of a new process to handle communication between the linked computers. This new process is called the *network process.* In overview, the function of each modified kernel process is

(a) *virtual memory process.* The extended virtual memory process implements the DSM model by determining whether faulting pages are local or remote, maintaining the IPT and XPT, and

implementing the page coherence protocol. The process also implements the stability scheme and maintains the SPT as part of this role.

(b) *network process.* This process accepts messages for transmission to remote nodes identified by logical node numbers, uses its NAT to convert the logical node number to the corresponding physical node number, encapsulates the message in a special MONADS packet, and finally transmits it. It receives and transmits up/down protocol messages and uses these to maintain the NAT. The network process also receives MONADS packets, extracts the message from the packet, and passes the message on to the appropriate local kernel process.

(c) *user process scheduler.* This process is extended to implement distributed process synchronisation (see section 5.4). The extensions allow the process scheduler to handle activate_process messages received via the network process from remote nodes. The user process scheduler can also cause the activation of a remote process by causing the transmission of an activate_process message to the appropriate remote node.

8.1.3 The Virtual Memory Message Block

Virtual memory message blocks enable kernel processes to exchange messages related to the management of virtual pages. Since distributed process synchronisation becomes necessary with the implementation of DSM, the blocks are also used to convey kernel messages relating to remote process scheduling. The processes which communicate using these blocks are the *disk process*, the *page fault interrupt process*, the *virtual memory process*, the *network process*, and the *user process scheduler.* The structure of the extended virtual memory message block, together with a full description of each of the fields and the possible types of message block indicated by the *request_type* field, are shown in appendix 4.

8.2 The Function of the Network Process

The network process acts as an interface between the local kernel and the network hardware. As such it shields the rest of the kernel from knowledge of the particular network being used. A kernel process wishing to transmit a message on the network simply prepares a virtual memory message block, links it to the network process request queue, and informs the network process of this linkage.

The network process itself has little knowledge of the structure of messages. It handles transmission and receipt of three types of message

(1) *long* messages consisting of a complete virtual page and associated protocol header information,

(2) *short* messages consisting of a protocol header only, and

(3) *up/down protocol* messages, which are a special form of short message.

Since the receipt of messages from other nodes is of an essentially random nature, the network process is implemented as an asynchronous device process. It is necessary to provide buffer space for the receipt and storage of incoming messages, with larger buffers being required for long messages. The nature of network traffic makes it impossible to predict whether the next incoming message is long or short. Because of this, a long message buffer is allocated to *every* incoming message. A list of the network messages handled by the network process is provided in appendix 1.

When a MONADS computer boots, the network process, as part of network initialisation, requests the provision of a number of main memory page frames to be used as buffers for the virtual memory page

part of received long messages[1]. This is achieved using virtual memory message blocks with the request_type field set to get_page_for_network. These page frames, together with smaller buffers maintained within the network process for the storage of MONADS message header information, are made available to the network hardware for the storage of incoming messages in the form of network *receive_requests*.

The network process checks the message_type field of the MONADS header section of every incoming message for the codes here_i_am, i_am_here_too, or node_going_down. These codes indicate that the message implements the up/down protocol described in section 5.2.3. Such messages are short, and do not require the use of a full page frame. The process automatically makes the page frame associated with up/down protocol messages available to the network hardware by using them to build a new receive_request.

The three up/down protocol messages are used by the network process to maintain the NAT. When a here_i_am message is received, the virtual memory process is informed using a virtual memory message block with the request_type field set to *node_on_line*. The virtual memory process then makes the necessary changes to the IPT and XPT, and invalidates pages if required (see section 7.5.2).

All other messages are passed to the appropriate kernel process unchanged by the network process using a virtual memory message block. The MONADS message header information is copied from the network process header buffer into the *monads_info* field of the virtual memory message block, and the main memory page frame containing the message data is pointed to by the *physical_address* field[2].

When a network process page buffer is used for an unsolicited incoming message, its replacement is the responsibility of the kernel process receiving the message. Such replacement is achieved by

---

[1] The received virtual memory page is received directly into the page frame which it will occupy. In this way we avoid the need to locally copy the page after it is received.

[2] For *short* messages there is no data stored in the attached page frame.

(1) obtaining a free main memory page frame[3], and

(2) linking a virtual memory message block with the request_type field set to *receive_request* onto the network process queue.

Whenever a kernel process asks the network process to transmit a message to which there will be a reply, it provides buffer space for the reply using a receive_request virtual memory message block. In this way it is ensured that there is always a sufficiently large pool of outstanding receive requests at the network process. The network process returns the reply message to the appropriate kernel process using a virtual memory message block with the request_type field set to *receive_request_reply*.

Extensions currently being designed will allow MONADS computers to be linked to networks using standards such as TCP/IP [30]. The messages exchanged by such networks do not include a MONADS header, and do not necessarily conform in size to the long, short, and up/down protocol messages used by the MONADS DSM. This inconsistency in size introduces some problems with provision of buffer space for receipt of messages, particularly since we wish to avoid local copying of virtual memory pages received as part of the DSM implementation. The design requires an additional level of message identification included in the network header section of messages[4]. Such information will allow the network process to differentiate between the receipt of MONADS messages, which will be processed as described in this thesis, and other messages, which will be handled by a new kernel process. Fortunately such identification is included as part of the TCP/IP definition.

---

[3] The receiving kernel process checks whether the incoming network message is *long* or *short*. If the message is short, the frame pointed to by the virtual memory request block does not contain any of the incoming message, and is immediately used to build a new *receive_request* page request block. If it is long, the frame pointed to by the virtual memory request block contains part of the message, so a free page frame is obtained prior to building the receive_request message block.

[4] This may be achieved in Ethernet, for example, using the *type* field of the message header.

8.2.1 The Ethernet Interface

The MONADS DSM has been tested using a network of three MONADS-PC computers each connected to a 10 mbps Ethernet using an EXOS 201 Intelligent Ethernet Controller [45]. This controller presents a standard Ethernet Data Link interface [18] to the network process by running the NX 200 Network Executive [46].

The network process communicates with NX 200 using two circular, singly linked lists maintained in shared host memory, and two one byte input/output ports. The circular lists respectively provide *host to Exos* and *Exos to host* message blocks. The input/output ports provide a means for

(1) resetting the Exos controller,

(2) clearing interrupts,

(3) returning Exos status information, and

(4) communicating one byte values to the controller as necessary, for instance, during initialisation of the controller.

The message blocks are used by the network process to convey data for transmission, and by the NX 200 Executive to provide received messages. Each message block contains an *owner* field which is initially set to *host* for blocks in the host to Exos list and to *Exos* for blocks in the Exos to host list.

Data for transmission over the ethernet as part of a single message may be provided in several non-contiguous memory buffers. Transmitting a virtual memory page, for instance, involves providing Exos with

(1) the address of a header buffer consisting of a standard Ethernet header followed by a MONADS header, and

(2) the address of a separate main memory page frame containing the virtual page.

These addresses, together with length and message type information, are provided to Exos by appropriately completing a host to Exos

message block. Informing Exos of a receive_request, on the other hand, involves providing

(1) the address of a header buffer,

(2) the address of a main memory page frame, and

(3) the address of a checksum buffer for error detection calculation,

together with length and message type information. When the message block is ready, the ownership field is set to *Exos* and a word written to one of the input/output ports to signal that Exos should check the host to Exos list.

Exos signals completion of a requested task by appropriately filling in an Exos to host message block defining,

(1) which request this block is answering,

(2) the status of the request, and

(3) the addresses and lengths of main memory buffers containing the transmitted or received data as appropriate.

Lastly Exos changes ownership of the message block to *host* and interrupts the network process to inform it to check the Exos to host queue.

8.3 Processing Page Faults

In the following discussion of the resolution of page faults we use the FAST scheme for addressing moved modules (see section 6.2.3).

The kernel virtual memory process is responsible for coordinating the movement of virtual pages into and out of the local main memory and disks. The process becomes aware of the occurrence of a page fault when it first receives a virtual memory message block for the page. It then categorises the page fault as either

(1) locally resolveable and local, or

(2) remotely resolveable and local, or

(3) locally resolveable and remote, or

(4) unresolveable.

To achieve this categorisation, the virtual memory process must decide whether the fault is

(a) local or remote, and

(b) locally resolveable, remotely resolveable, or not resolveable at all.

8.3.1 Determining Whether a Page Fault is Local or Remote

The virtual memory process first becomes aware of the existence of a page fault for a page when it receives a virtual memory message block with the request_type field set to either

(a) *receive_request_reply*. These blocks, which indicate a remote page fault, always emanate from the network process as a result of the receipt of a message from another node, or

(b) *new_page_fault*. These blocks, which indicate a local page fault, emanate from either the local page fault interrupt process, or the local virtual memory process, as described below.

When the virtual memory process receives a kernel message with the request_type field set to receive_request_reply, it then checks the *monads_info* field of the block. The structure of this field is shown in appendix 4. If the *monads_message_type* field is set to *request_page*, then the block contains a remote request for a local page. The virtual memory process uses the *requested_as*, *requested_page*, and *advisory_info* fields of the monads_info to fill in the *address_space*, *page_number*, and *advisory_info* fields of the virtual memory message block, and sets the request_type field to new_page_fault. The resultant virtual memory message block is very similar to the block used by the page fault interrupt process to inform the virtual memory process of a local page fault. The virtual memory process then links the block onto

its own request queue, and performs a *V* operation on its own semaphore. In this way the processing of a remote request may be achieved using largely the same mechanisms as are used for the processing of local page faults. It is only after the page is in main memory that processing of remote and local faults differs again.

8.3.2 Determining the Resolveability of a Page Fault

As the first step in determining whether a page fault is resolveable, and if so whether locally or remotely, the virtual memory process checks the *advisory_info* field of the virtual memory message block. Such advisory information is provided for moved modules by either

(1) the microcode which implements the module *open* instruction for a local page fault[5], or

(2) the network message requesting provision of the page for a remote page fault.

If the advisory field is not empty, the provided node number/volume number pair is used as the storage location of the page. Otherwise the data stored in the *address_space* field is used. The process then

(1) checks the LMT to see whether the volume probably containing the page is locally mounted. If the appropriate LMT entry does not exist, then

(2) checks the FMT to see whether the volume probably containing the page is listed. If the appropriate FMT entry does not exist, then

(3) assumes that the volume probably containing the page is still at its original location.

---

[5] Once a moved module is open, use of the FAST technique precludes the need for advisory information for subsequent accesses to pages from the module (see section 6.2.3).

After this sequence of steps, the virtual memory process has made a decision as to whether the page fault is locally or remotely resolveable. This decision may, of course, be incorrect because either

(a) the advisory information is out of date, or

(b) the page fault is in fact unresolveable.

8.3.3 Resolution of a Local Page Fault

When a local page fault occurs, the microcode activates the *page interrupt process*. This process obtains a virtual memory message block from the heap of available blocks, fills in the address_space, page_number, and process_number fields, sets the request_type to *new_page_fault*, sets the request_from field to *local*, and sets the disk_number field to -1 since the actual disk containing the page is at this stage unknown. The process that generated the faulting address is then suspended and the virtual memory message block is linked to the virtual memory process queue.

The virtual memory process acts as a state machine, with the current state of the page represented by a virtual memory message block being indicated by the contents of the *request_type* field. Pages, and thus virtual memory request blocks, move between states according to the actions taken by the virtual memory process and the disk process as shown in appendix 2. As previously described, the initial state for a block representing a local page fault is *new_page_fault*.

The first check determines whether the faulting virtual page is actually in main memory but marked as invalid in the ATU. This situation occurs when an occupied and unmodified main memory page frame is returned to the *free list*[6], or a modified page is linked to the *wait list* in preparation for writing to disk prior to being returned to the free memory pool. Such a page is simply removed from the free or wait list and its ATU entry is validated. A user scheduler message block is then completed, marked as indicating that a process is being re-activated

---

[6] This list implements the pool of free main memory page frames.

following a page fault, and linked to the user process scheduler message queue. This effectively reactivates the suspended user process.

If the required page is *really* not in main memory, the virtual memory process

(1) Checks that there are sufficient free main memory page frames to enable resolution of the page fault. If the pool of free main memory page frames is too small[7], processing of the page fault is suspended and page discard is initiated.

(2) Determines whether the fault is locally resolveable, as described in section 8.3.2.

8.3.3.1 Local Resolution of a Page Fault

In order to transfer a locally stored virtual page into a main memory page frame, the disk address of the page must be determined. To do this the address space page table is consulted[8]. As described in section 4.3.2, this page table is two-tiered, consisting of a *primary page table* potentially occupying 32 pages of virtual memory, and a *secondary page table* containing an entry for each of the primary page table pages. The secondary page table is stored in the root page of the address space. To allow efficient access to the pages of small address spaces, the first 256 entries of the primary page table are also stored in the root page of the address space. In order to find the disk address of the faulting page, the virtual memory process must execute the following algorithm.

---

[7] The pool must be big enough to provide for the largest possible number of associated page faults (see section 4.3.2).

[8] In the case of a page fault on the root page of a moved module the page table is consulted in conjunction with the FAST.

```
begin {algorithm for obtaining the disk address of a page}
  if the page is in the lowest 256 pages of the address space then
    begin
      read the page disk address from the root page;
      {if this read is successful then the disk address of the
                                          faulting page is known}
      if the result is a page fault then
        begin
          obtain the disk address of the root page
                    from the volume address space table;
          {this table is implemented as a hash table and is locked
                                          down in main memory}
          with the virtual memory message block do
            begin
              store the disk address in the disk_page field;
              store the root page virtual page number in
                                          the page_disk field;
              set the request type to disk_read;
              obtain a free main memory page frame;
              store the frame address in the physical address field
            end; {with}
          link the message block to the disk process queue;
          perform a V operation on the disk process semaphore;
          wait for the message block to be returned;
          map the root page into the ATU;
          read the disk address of the faulting page
        end {if page fault reading primary page table}
    end {if in lowest 256 pages}
  else {page is above the first 256 pages of the address space}
    begin
      calculate the virtual address of the primary page table entry;
      read the disk address of the faulting page from the primary
                                                page table;
      {if this read is successful then the disk address of the
                                          faulting page is known}
      if the result is a page fault then
        {the primary page table page is not in main memory}
        begin
          read the disk address of the primary page table page
```

```
    from the secondary page table held in the root page of
    the address space;
  if the result is a page fault then
  {address space root page is not in main memory
                                        - bring it in}
    begin
      obtain the disk address of the root page from the
                                  volume address space table;
      with the virtual memory message block do
        begin
          store the disk address in the disk_page field;
          store the root page virtual page number in
                                      the page_disk field;
          set the request type to disk_read;
          obtain a free main memory page frame;
          store the frame address in the physical
                                          address field
        end {with}
      link the message block to the disk process queue;
      perform a V operation on the disk process
                                          semaphore;
      wait for the message block to be returned;
      map the root page into the ATU;
      read the disk address of the primary page table page
    end; {if page fault reading secondary page table}
  {now we have the disk address of the primary page table
                          page - bring it into main memory}
  with the virtual memory message block do
    begin
      store the disk address in the disk_page field;
      store the virtual page number in the page_disk field;
      set the request type to disk_read;
      obtain a free main memory page frame;
      store the frame address in the physical address field
    end; {with}
  link the message block to the disk process queue;
  perform a V operation on the disk process semaphore;
  wait for the message block to be returned;
  map the primary page table page into the ATU;
```

```
                read the disk address of the faulting page from the
                                                    primary page table
            end {if page fault reading primary page table}
        end {else page is above first 256 pages}
        {at this stage we have the disk address of the faulting page}
    end; {of algorithm for finding the disk address of a virtual page}
```

When the virtual memory process knows the disk address of the faulting page, it

(1) completes the disk_function, disk_number, and disk_page fields of the virtual memory message block,

(2) obtains a free main memory page frame and stores its address in the physical_address field, and

(3) links the block to the disk process queue.

The disk process loads the requested disk page into the provided main memory page frame, marks the virtual memory message block as *page_on_way_in*, and links the block to the virtual memory process queue. When the virtual memory process receives the block, it

(1) checks the disk_status field, and if the read was unsuccessful it repeats the read request,

(2) maps the page into the ATU

(3) obtains and completes a user scheduler message block and links the block to the user process scheduler queue, with the result that the waiting user process will be re-activated.

If the page fault occurred as part of a module open operation, the virtual memory process provides enough status information to allow the microcode to modify the MCS segment capabilities as described in section 6.2.3, and if necessary to update the module capability advisory field.

8.3.3.2 Remote Resolution of a Page Fault

If the volume that probably contains the page is not locally mounted, the kernel must obtain a copy of the page from the node on which the volume is mounted, the *owner node* for the page.

The initial step is to prepare for receipt of the virtual page by providing the network process with a page frame buffer. To do this, the virtual memory process

(1) obtains a free virtual memory message block,

(2) obtains a free main memory page frame and stores its address in the physical_address field of the new virtual memory message block,

(2) sets the request_type field of the new block to *receive_request*, and sets the block_source field to *virtual_memory_process*, and

(3) links the receive_request virtual memory message block to the network process queue, and interrupts the process to inform it of the new request.

The next step is to cause transmission of a *short* message requesting provision of the virtual page. To do this the virtual memory process uses the original new_page_fault virtual memory message block, and

(1) sets the request_type field to *send_request_short*, and the block_source field to *virtual memory_process*,

(2) builds the MONADS header information by setting the source field to the *local logical node number*, the reply_to field to *virtual memory process*, the message_type field to *request_page*, the volume field to the volume number indicated in the capability advisory field or page address as appropriate, and copies the requested_as and requested_page fields from the matching fields of the original message block,

(3) completes the MONADS header information by setting the destination field to the logical node number obtained from the FMT, advisory information, or page address as appropriate (see

section 8.3.2), and the volume field to the volume number obtained from the advisory information, or page address as appropriate,

(4) links the completed message block to the network process queue, and

(5) interrupts the network process to inform it of the newly linked message block.

When the network process has transmitted the request_page message, it links the message block back onto the virtual memory process queue and signals this by performing a *V* operation on the process semaphore. The request_type field of the block is set to *send_request_short_reply*. On receipt of such a message block, the virtual memory process

(1) checks the request_status field, and

(2) if the status is satisfactory, links the message block to the *pending requests queue*, which is a list of virtual memory request blocks for which a reply has not been received. This list is maintained by the virtual memory process.

If the request_status field indicates that the destination node is off-line[9], an *invalid_page* exception condition occurs for the user process indicated in the process_number field of the message block.

The reply to the request_page message is received by the virtual memory process as a virtual memory message block with the message_type field set to *receive_request_reply*. If the message_type field of the MONADS header is set to *supply_page*, the required virtual page is contained in the page frame whose address is contained in the physical_address field of the message block. In this case the requested_as and requested_page fields of the MONADS header are used to match the newly arrived page with the appropriate message block on the pending requests queue. Then

---

[9] The fact that a node is off-line is indicated to the network process by the lack of an entry mapping the logical node number to a network address in the NAT (see section 5.2.3).

(1) the supplied_as and physical_address fields are copied from the newly received message block to the block on the pending requests queue,

(2) the newly received block is returned to the pool of free virtual memory request blocks,

(3) the request_type field of the original message block is set to *new_page_on_way_in*,

(4) the block is linked onto the virtual memory process message queue, and

(5) a *V* operation performed on the virtual memory process semaphore.

When the newly linked message block is processed, resolution of the page fault is completed as described in section 8.3.3.

8.4 Conclusion

The MONADS kernel is structured as a set of co-operating processes which communicate using message blocks. Each of these processes performs a specific kernel task. Implementation of the MONADS DSM involved modification to the virtual memory and user process scheduler kernel processes, and the creation of a new network process.

The new network process interfaces with the network hardware, maintains knowledge of the status of other nodes, and provides a network message passing service to the rest of the kernel. Modifications to the virtual memory process allow it to detect the local occurrence of remote page faults, act as a server of local pages in accordance with the coherence protocol, and implement the stability protocol. Changes to the user process scheduler allow it to co-operate with the schedulers on other nodes, thus scheduling user processes on a network-wide basis.

Although no accurate timing is available yet due to lack of support in the higher level system software, initial indications are that network

paging will not be significantly slower than paging from local disk. This is supported by reports from other DSM implementations [74], particularly if the required page already exists in the main memory of another node [27].

Chapter 9 CONCLUSION

In the introduction to this thesis two major aims were identified for the research. These were

(1) The development of a scaleable design for a local area network (LAN) of computers linked to form a distributed system. This design would

- allow the sharing of all resources such as data, programs, and devices,
- provide fine-grained protection of access to resources,
- provide location and naming transparency, and
- provide a coherent store, so that all users at all nodes have the same view of shared data.

(2) The development of a stable virtual store. Such a store would

- allow individual modules or groups of modules to be stabilised independently of the other modules in the store,
- provide a basis for the construction of higher-level transaction based systems by providing for user initiation of stabilise operations,
- make efficient use of disk space and processor cycles, and
- be extendable to a network of connected machines.

Several approaches to the provision of distributed access to resources were examined. These were

(1) Message Passing,

(2) Remote Procedure Call (RPC), and

(3) Distributed Shared Memory (DSM).

Message passing systems allow the sharing of data *by value*, meaning that dynamic data structures must be flattened before they may be shared. Such systems were found to be deficient in the areas of naming and location transparency, access control, and fault tolerance. The message passing facility provided by these systems was, however, found to be useful in the implementation of RPC systems.

RPC systems also require the flattening of dynamic data structures prior to sharing. Some sophisticated RPC-based systems were examined, and these were found to provide transparent machine interconnection, distributed file systems, and distributed load sharing. Some deficiencies were found in the areas of data consistency, access control, and fault tolerance.

DSM allows the sharing of data *by reference*, meaning that all kinds of data structures may be shared. The DSM systems examined successfully provided location and naming transparency, and maintained a coherent view of shared data. Data was not, however, managed in a uniform manner, because the memory space available to processes was partitioned into shared and private regions. This method of management was used to minimise the size of the full shared memory page table maintained at each of the connected nodes, and to provide a means of protecting local data from general access. Each of the shared memory page tables contains data about the location and status of every page of the shared memory, meaning that the size of the table is directly proportional to the size of the shared memory. Deficiencies of the DSM systems examined include the lack of a uniformity of data management, the lack of scaleability of the DSM, the lack of protection of access to data stored in the DSM, and the lack of fault tolerance.

Despite the deficiencies of existing DSM systems, the DSM paradigm was selected as being a natural and elegant approach to the achievement of transparent distribution. The result of the research is the development of a new approach to the construction of DSM. This approach is significant because the the new system offers

(1) uniformity of access to resources,

(2) naming transparency,

(3) location transparency,

(4) fine-grained control of access to resources,

(5) stability,

(6) persistence, and

(7) scaleability.

In the following sections we summarise the achievements in each of these areas.

9.1 Uniformity of Access to Resources

The use of the distributed shared memory paradigm for the entire virtual memory addressable by processes results in uniformity of access to resources. All such resources are accessed *by reference*, that is according to their location in the DSM. Such references are the virtual memory addresses of the code which implements programs or device drivers, or of data which is used and accessed by such programs. The virtual memory space is universal, with every process having a coherent view of this memory regardless of the node on which it is executing.

9.2 Naming Transparency

All resources are identified by a unique name which defines the location of the resource in the virtual memory space. A unique range of virtual memory addresses is allocated to each node, and all objects created by a node reside in the node's address range. The virtual addresses allocated to a node are further divided into within-node disk partitions (called volumes) and within volume address spaces (called modules). The name or address of a resource, then, uniquely identifies the resource by defining the node on which the resource was created, and the original storage location of the resource within that node. This use of structured addresses also facilitates distributed access to resources. Significantly this is achieved without sacrificing location transparency.

9.3 Location Transparency

Despite the fact that structured addresses are used to provide naming transparency, it is possible to move modules between volumes and volumes between nodes. An efficient technique which provides distributed access to the modules stored on a moved volume was described. Alternate approaches to the relocation of individual modules were developed. The first approach, which uses the moved object table (MOT), introduces inefficiency because it imposes an overhead on the location of unmoved as well as moved modules. The second approach, which uses the foreign address space table (FAST), allows distributed access to moved modules without imposing an overhead on access to unmoved modules.

9.4 Control Over Access

The DSM provides a single global virtual memory space. All processes, programs, and data reside in this memory space. The system provides a two level protection scheme which affords fine-grained control over access to objects in the DSM. The highest level of protection is provided by module capabilities over access to information-hiding modules. These capabilities define the module, the set of available module interface procedures, and capability propagation permissions. No user may access a module without the appropriate module capability. A second level of protection controls access within a module. This level of protection is achieved using segment capabilities. These control access to sections of modules, called segments, which contain logically related data such as a specific data structures or programs. The use of such sophisticated protection techniques permits the storage of all data in the DSM without compromising efficiency of access to private data.

9.5 Stability

The stability scheme developed in this thesis is necessary because at any time the state of the DSM is a combination of the contents of the local page caches and the data stored on disk. This means that a failure which prevents the storage of modified data to disk may result in a disk

image of the DSM which presents an inconsistent view of the system state. The disk image of the DSM is, of course, the starting point for the system when it restarts. The stability scheme was developed in two stages. The first stage ensures the stability of a non-distributed virtual store using a technique based on shadow paging. The second stage extends and modifies the non-distributed scheme, using a network message protocol to provide stability for the entire DSM. The scheme represents a significant improvement on alternate designs because it involves no copying of data, makes efficient use of available disk space, does not require static partitioning of disks, and allows sub-sections of the store to be stabilised separately. This last property of the scheme makes it an ideal basis for the development of a higher-level transaction mechanism as required by database management systems.

9.6 Persistence

The distributed store created by this research provides a uniform storage abstraction which is independent of the lifetime of the data being stored. As such the store may be described as a persistent store. The achievement of storage uniformity is a result of several separate features of the DSM. Firstly, there is no concept of a separate file system. All programs and data reside in the virtual memory. The movement of data between main memory and secondary storage is handled by the system and is totally transparent to the user. Secondly, all objects in the store are treated in a uniform manner, so that the persistence of an object is not related to its type. Thirdly, the store is stable, as described above, meaning that the store appears to be error free, and that failure of the store is hidden from the user. Lastly, the size of the store is scaleable and bounded only by implementation, so the store provides the abstraction of having unbounded size.

9.7 Scaleability

The DSM model developed in this thesis is scaleable to larger stores than that implemented, which uses 60 bit virtual addresses. This feature is a major difference between the MONADS DSM model and other DSM models described in the literature. Virtual memory systems

must maintain data structures which map between pages of the virtual memory and main memory page frames for those pages which exist in main memory, and between pages of virtual memory and disk pages for those pages which do not exist in main memory. A single data structure called a page table is typically used to store these mappings. Scaleability of the MONADS DSM is achieved primarily through the separation of these virtual memory to main memory and virtual memory to disk mappings. The result is that disk page tables are maintained in virtual memory together with the modules they describe, whilst main memory page tables are maintained on a per mode basis and describe only the virtual pages currently in main memory at that node. This design results in main memory management data structures which grow logarithmically with the size of the virtual memory. As a result the DSM is scaleable to large virtual memories.

9.8 Future Directions

The design presented in this thesis provides the basis for a practical and scaleable local area network based on the DSM paradigm. However there are a number of areas for future research. These include

(1) further development of the network-wide stability scheme,

(2) the expansion of the architecture to provide for the connection of heterogeneous machines,

(3) examination of the suitability of the stability approach to maintenance of integrity of the distributed store, and

(4) the development of techniques for the efficient addressing and effective utilisation of massive main memories.

The network-wide stability scheme proposes the maintenance of a dependency graph at each node to describe volume inter-relationships. When the volumes whose relationship is described by such a graph are stabilised, this must be achieved as an atomic operation. The protocols used to maintain such graphs, together with protocols for the election of a master node for the stabilise operation, and for recovery from the failure of such a master, require further investigation.

Research into the connection of the DSM to a heterogeneous network has already commenced. The design of TCP/IP connectivity is under development, and will allow the receipt and transmission of standard IP packets. The use of such connectivity to allow heterogeneous access to the DSM is under investigation.

The stability approach to the problem of store integrity, whilst apparently suitable for local area network (LAN) application, may not be the optimum scheme for a wide area network (WAN) implementation of the architecture. It appears, for instance, that the distances involved in WANs, with its effect on message propagation, may result in unacceptable delays during stabilise operations. The use of transaction-based updates to shared data requires investigation. The results of such research may have implications for the problem of network-wide stability in LANs.

Finally, investigation into the use of massive main memories has already commenced [103]. At this stage such work has focused on addressing and utilisation issues. Investigation of a distributed massive memory architecture would involve issues such as the optimum granularity for distribution and coherence, and the possibility of migration of processes to the source of massive data structures.

Appendix 1 Network Message Types

Nodes communicate by transmitting messages on the interconnecting network. These messages are used to exchange both protocol information and virtual memory pages. The *message_type* field of the monads header for a message defines the purpose of the message. Possible messages are

here_i_am(sending node number, physical network address)

here_i_am_too(sending node number, physical network address)

invalid_address_space(requested page number)

request_changed_access_rights(requesting node number, page number, new access rights)

access_rights_changed(page number, new access rights)

return_page(returned from node number, page number, page data)

page_received(receiving node number, page number)

invalidate_page(page number)

page_invalidated(sending node number, page number)

send_page(send to node number, page number)

node_shutting_down(sending node number)

activate_process(process number)

where_is_volume_mounted(requesting node number, volume number)

volume_mounted(mounting node number, volume number)

module_moved(module number, new volume number)

The following messages are used for the MOT implementation of moved modules

request_page(requesting node number, requested page number, storage volume number)

supply_page(supplied page number, page data)}

The following messages are used for the FAST implementation of moved modules

request_page(requesting node number, requested page number)

request_root_page(requesting node number, requested page number, storage volume number)

supply_page(supplied page number, page data)

supply_root_page(supplied page number, page data, requested page number)

Appendix 2 State Transition Diagrams

This appendix provides an alternate view of the DSM model by describing it in terms of the *states* of a virtual memory page. The virtual memory page is the unit of transfer of program code and data between local disk and main memory, and between the main memories of network nodes. When an event (such as an interrupt or receipt of a message) occurs for a page, the action taken by the kernel depends on the current state of the page. Typically, the effect of this action moves the page to a new state.

Fifteen states for a virtual page are identified, and are listed below. Two state transition diagrams are included. The first diagram shows transitions between states in terms of events, with these events being described on the arcs which represent the transition. The second diagram, which shows state transitions in terms of network messages, uses a legend to indicate the message or sequence of messages causing a transition.

A2.1 States of a Virtual Memory Page

1 On Disk at Owner Node

The page is stored on a disk page at the owner node, and no copy of the page exists in the main memory of any node. This is the initial state for every virtual memory page.

2 In Owner's Main Memory Only, RO, Unmodified, Not Locked Down

The page is mapped into the owner node's ATU as read-only. It does not exist in the main memory of any other node. It has not been modified since being brought into main memory, and it is not locked down, meaning that page discard may remove it from main memory at any time.

3 In Owner's Main Memory Only, RO, Unmodified, Locked Down

The page is mapped into the owner node's ATU as read-only. It does not exist in the main memory of any other node. It has not been modified since being brought into main memory, but it is locked down. This means that page discard may not remove it from main memory. This state is used to ensure that the page cannot disappear from main memory whilst awaiting transmission to another node.

4 In Owner's Main Memory, RO, Modified, Not Locked Down

The page is mapped into the owner node's ATU as read-only, but marked as modified. Prior to page discard, the page must be flushed to disk, thus updating the permanent copy of the page.

5 In Owner's Main Memory, RO, Modified, Locked Down

The page is mapped into the owner node's ATU as read-only, but marked as modified. Since the page is locked down, page discard may not remove it from main memory. This state is used to ensure that the page cannot be marked as invalid in the ATU whilst awaiting transmission to another node.

6 In Owner's Main Memory RW

The page is mapped into the owner node's ATU as read/write. In accordance with the coherence strategy, the page does not exist in the main memory of any other node.

7 In Owner's Main Memory Unmodified, RO, and in Main Memory of Remote Nodes Unmodified, RO

The page is mapped into the owner node's ATU as read-only and unmodified. The page is also mapped into the ATU of more than one other one other node, in each case unmodified.

8 In Owner's Main Memory Unmodified, RO, and in Main Memory of One Remote Node Unmodified, RO

The page is mapped into the owner node's ATU as read-only and unmodified. The page is also mapped into the ATU of one other one other node unmodified.

9 In Owner's Main Memory Modified, RO, and in Main Memory of Remote Nodes Unmodified, RO

The page is mapped into the owner node's ATU as read-only and modified. The page is also mapped into the ATU of more than one other node, in each case unmodified.

10 In Owner's Main Memory Modified, RO, and in Main Memory of One Remote Node Unmodified, RO

The page is mapped into the owner node's ATU as read-only and modified. The page is also mapped into the ATU of one other node unmodified.

11 In Main Memory of Remote Nodes Unmodified, RO, Not in Owner's Main Memory

The page is mapped into the ATU of more than one remote node as read-only, but is not mapped into the owner node's ATU. In each case the page is marked as unmodified in the remote node's ATU.

12 In Main Memory of Remote Node Unmodified, RO, Not in Owner's Main Memory

The page is mapped into the ATU of one remote node as read-only, but is not mapped into the owner node's ATU. The page is marked as unmodified in the remote node's ATU.

13 In Main Memory of Remote Node Modified, RO, in Main Memory of Remote Node(s) Unmodified, RO, Not in Owner's Main Memory

The page is mapped into the ATU of multiple remote nodes as read-only, but is not mapped into the owner node's ATU. In the ATU of one of the remote nodes the page is mapped as modified.

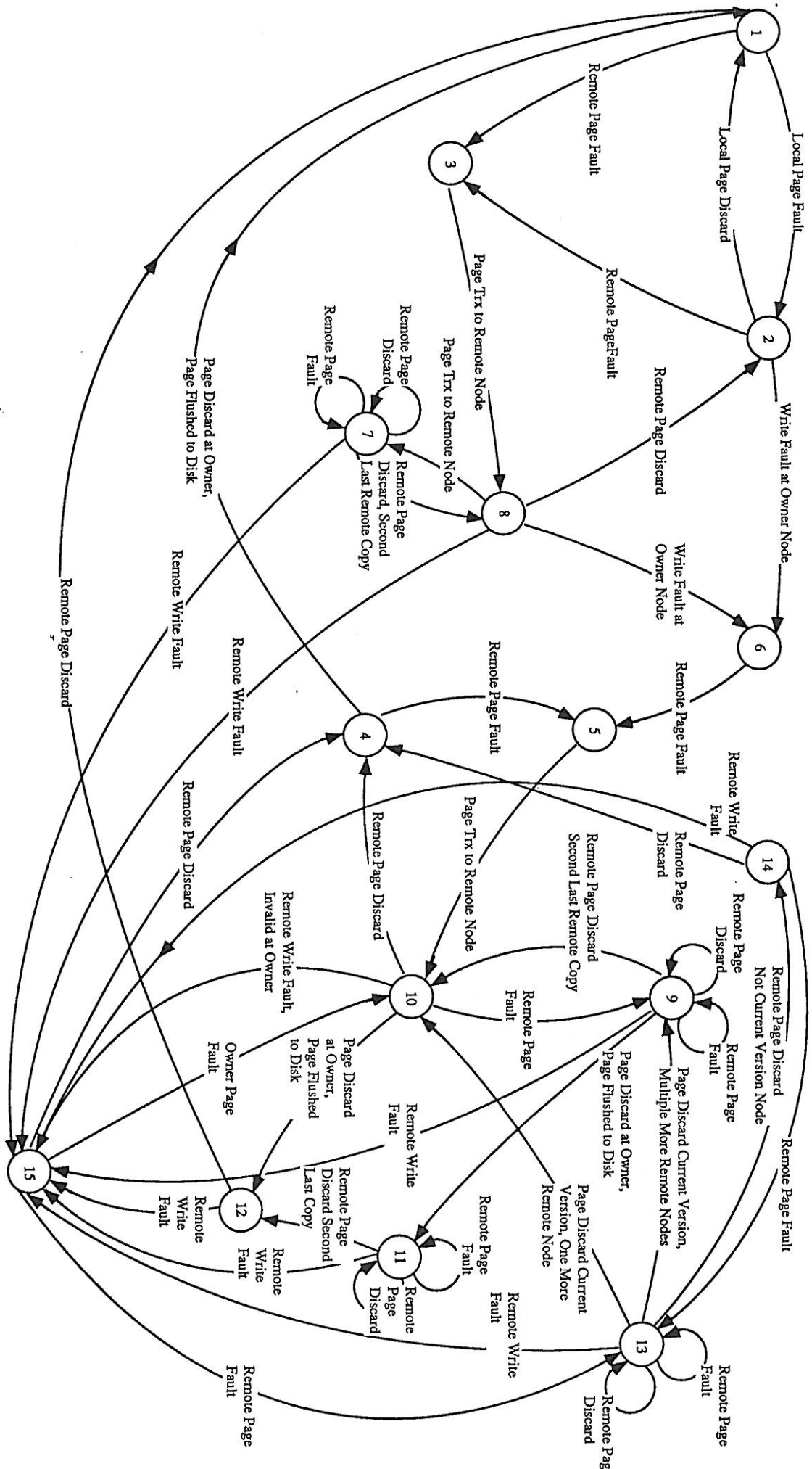
14 In Main Memory of Remote Node Modified, RO, Not in Owner's Main Memory

The page is mapped into the ATU of a remote node as read-only, but is not mapped into the owner node's ATU. In the ATU of the remote node the page is mapped as modified.

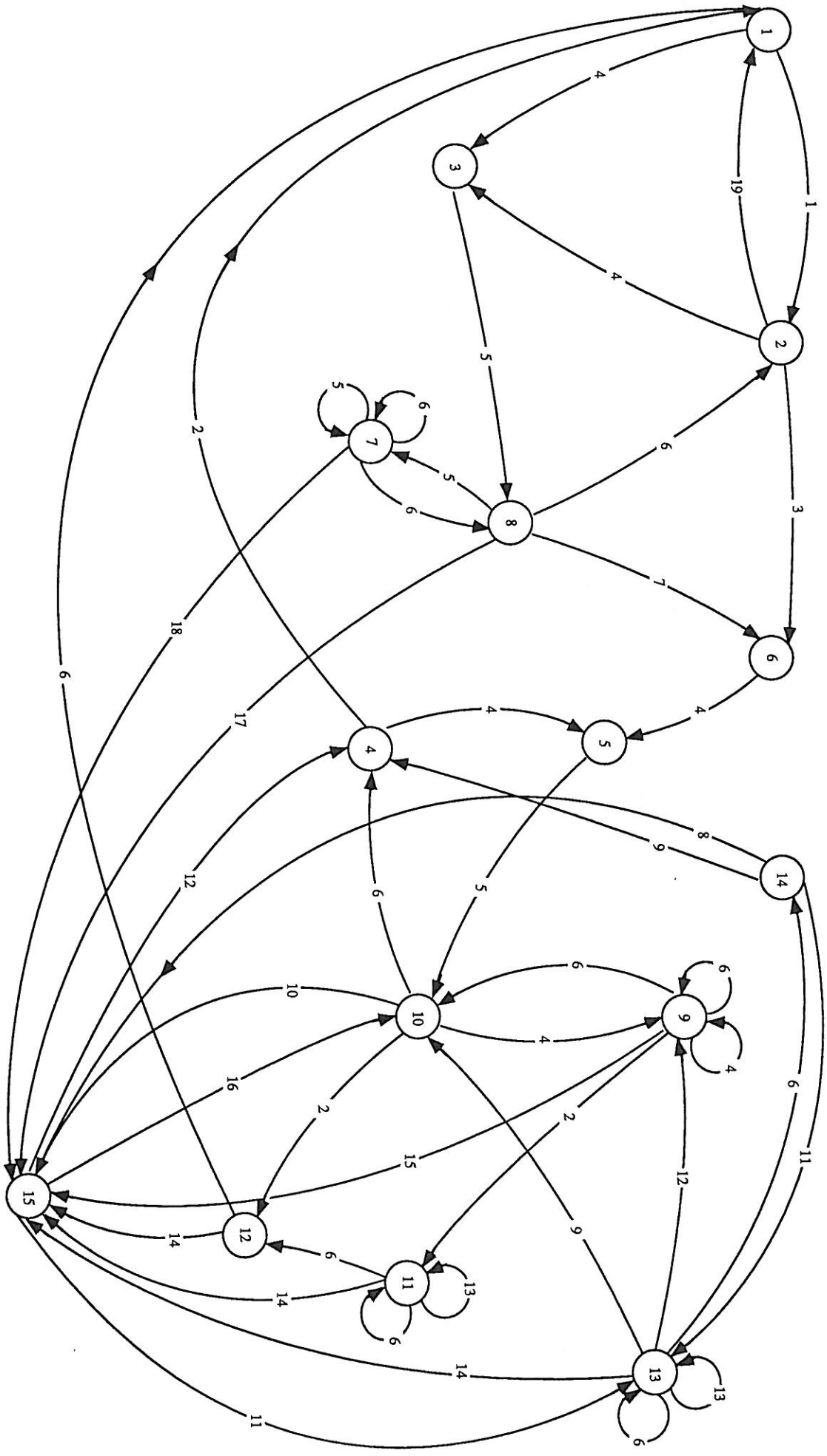
15 In Main Memory of Remote Node RW

The page is mapped into the ATU of a remote node as read/write. In accordance with the coherence strategy, the page does not exist in the main memory of any other node.

A2.2 State Transition Diagram (transitions in terms of events)



A2.3 State Transition Diagram (transitions in terms of messages)



Legend for state transition diagram A2.3, depicting state transitions in terms of messages.

1 Owner page fault interrupt resulting in *new_page_fault* message linked to virtual memory process queue.

2 Owner page discard resulting in a message being linked to the disk process queue with the disk_function field set to *write.*

3 Write fault interrupt at owner node. Handled internally by local virtual memory process.

4 Owner node receives *request_page* message.

5 Owner node provides the page using a *supply_page* message.

6 Owner node receives *page_invalidated* message for the page.

7 Owner node transmits *invalidate_page* message(s) and receives *page_invalidated* message(s).

8 Owner node receives *request_changed_access_rights* message. Owner responds with *access_rights_changed* message.

9 Owner node receives *return_page* message and replies with *page_received* message.

10 Local page discard resulting in a message being linked to the disk process queue with the disk_function field set to *write.* When discard complete, owner responds with *access_rights_changed* message.

11 Owner receives *request_page* message. Transmits *send_page* message to node with page. Node sets access to read-only, and transmits copy of page with *supply_page* message.

12 Owner receives *return_page* message, and replies with *page_received* message.

13 Owner receives *request_page* message. Transmits *send_page* message to node with copy of page. Node transmits copy of page with *supply_page* message.

14 Owner receives *request_changed_access_rights* message. Transmits *invalidate_page* messages to other nodes with copies of the page. When all *page_invalidated* messages received, owner transmits *access_rights_changed* message to requesting node.

15 Owner receives *request_changed_access_rights* message. Owner page discard resulting in a message being linked to the disk process queue with the disk_function field set to *write*. Owner transmits *invalidate_page* messages to other nodes with copies of the page. When all *page_invalidated* messages received, owner transmits *access_rights_changed* message to requesting node.

16 Owner transmits *send_page* message (with itself as the send to node) to node with page. Node sets access to read-only, and transmits copy of page back to owner with *supply_page* message.

17 Owner receives *request_changed_access_rights* message. Owner invalidates local copy of page. Owner then transmits *access_rights_changed* message to requesting node.

18 Owner receives *request_changed_access_rights* message. Owner invalidates local copy of page. Owner transmits *invalidate_page* messages to other nodes with copies of the page. When all *page_invalidated* messages received, owner transmits *access_rights_changed* message to requesting node.

19 Page invalidated at owner. Handled internally by virtual memory process.

Appendix 3 Algorithms for the operation of kernel processes

There are three categories of kernel process. These are

(1) server processes,

(2) synchronous device processes, and

(3) asynchronous device processes.

In this appendix the algorithm for the operation of each of these categories is shown.

A3.1 Algorithm for the Operation of a Server Process

The sole role of a server process is to service requests from other processes. A server process has no associated interrupt, and never directly interacts with hardware devices. As such it is scheduled by semaphores only. The algorithm for the operation of a server process is

repeat

ksemp(queue semaphore);

process next item on queue;

remove the item from the queue;

notify the requesting process of completion

forever;

A3.2 Algorithm for the operation of a Synchronous Device Process

A synchronous device process interacts with other kernel processes and with hardware devices. The interaction with hardware is of a cause and effect nature, meaning that hardware interrupts are totally predictable and synchronous with the operation of the process. A synchronous device process is scheduled using a combination of hardware interrupts and semaphores, and the algorithm for its operation is

repeat

ksemp(queue semaphore);

get next entry from queue;

start transfer on associated hardware device;

ksusp; {waiting for interrupt from h/w device}

check result;

finish job;

remove the corresponding item from the queue;

notify the requesting process of completion

forever;

A3.3 Algorithm for the operation of an Asynchronous Device Process

An asynchronous device process interacts with other kernel processes and with hardware devices. This interaction is of an essentially random nature, and it is not possible to predict whether the next activation of the process will be to service a hardware or software request. An asynchronous device process is scheduled using a combination of hardware interrupts and software interrupts, and the algorithm for its operation is

```
repeat
        ksusp;
        while there is work to be done do
            begin
                check origin of request;
                    {activation indicates whether h/w or s/w}
                if origin is hardware then
                    begin
                        do some work;
                        notify kernel process if appropriate
                    end
                else {origin is software}
                    begin
                        do some work;
                        remove request item from queue;
                        start transfer on associated h/w
                                                    device
                        if appropriate
                    end
            end
forever;
```

Appendix 4 Implementation Details

This appendix describes the various data structures used in implementing the MONADS DSM.

A4.1 Virtual memory request block

The virtual memory request block is used for kernel processes *at the same node* to communicate as necessary to achieve virtual memory management at the node. The fields of a virtual memory request block are

(a) *request_next.* Pointer to the next block in a list of blocks.

(b) *request_type.* Code defining what action is being requested with this block. The actions are listed in figure ??, and their purpose described in the text.

(c) *address_space.* Number of address space containing the page to which the original request refers.

(d) *page_number.* Number of the within address space page to which the original request refers.

(e) *advisory_info.* Advisory information gleaned from the module capability used in an attempt to open a module. This information is copied into the virtual memory request block when it is set up by the page fault interrupt process. The use of this information is described in section 6.2.

(f) *process_number.* The number of the local process waiting on the page, if any.

(g) *disk_page_link.* Link to list of virtual memory request blocks referring to the same disk page.

(h) *process_link.* Link to list of virtual memory request blocks referring to the same virtual page.

(i) *net_page_link.* Link to list of virtual memory request blocks referring to virtual pages currently being transmitted onto the network from this node.

(j) *page_status.* Environment field containing current status of search through the volume address space directory.

(k) *environment_1.* Environment field used for temporary storage of hash table information when searching through the volume address space directory.

(l) *lock_bit.* Flag indicating whether the virtual page is to locked into main memory when mapped in.

(m) *write_fault_bit.* Flag indicating that virtual memory request block pertains to resolution of a write fault on the virtual page.

(n) *disk_function.* Used when virtual memory request block is requesting disk transfer. Possible function codes are *read*, *write*, or *mount.*

(o) *disk_number.* Used when virtual memory request block is requesting disk transfer. Defines the logical volume number for the volume on which the disk transfer is to occur.

(p) *disk_page.* Used when virtual memory request block is requesting disk transfer. Defines the disk page on which the read or write operation is to occur.

(q) *physical_address.* Contains the start address of the main memory page used to store the virtual page to which this request refers.

(r) *disk_status.* Transfer status returned by the disk process. If this flag is non-zero a disk transfer error has occurred.

(s) *request_status.* Transmission status returned by the network process. If this flag is non-zero then a transmission error has occurred.

(t) *address_space_disk.* Contains the address space number of the current disk request. May differ from the address_space field if finding the originally requested page involves page faults on pages containing disk directory information.

(u) *page_disk.* As for the field above, contains the within address space page number of the page which is the subject of the current disk request.

(v) *network_message_destination.* Contains a code defining the destination kernel process for a network message.

(w) *trx_number.* Contains the transaction number for a network request.

(x) *block_source.* Contains the source kernel process for network requests.

(y) *request_from.* Flag indicating whether a request to resolve a page fault is to resolve a local or a remote fault.

(z) *monads_info.* Header information used in creating MONADS messages for network transmission, or copied from a received MONADS network message.

A4.2 Virtual memory request block *request_type* codes

The request_type field defines the purpose of a virtual memory request block. The possible values of this field are

(a) *new_page_fault.* The message block represents a new user process page fault.

(b) *page_on_way_in.* The message block signals the fact that a virtual page has been brought into a main memory page frame and that completion of the page fault resolution may be carried out.

(c) *hash_table.* The volume directory is being searched to find the disk address of the root page of the address space.

(d) *new_page_on_way_in.* The required page is in main memory and simply needs to be mapped into the ATU to complete resolution of the page fault.

(e) *high_page_table_on_way_in.* The page containing the high page table entry for the required page is in main memory and requires mapping into the ATU.

(f) *new_page_table_on_way_in.* A process has accessed a virtual page for the very first time. The page containing the primary page table entry for the page had also never been accessed, so a new disk page was allocated to the page table page and all entries initialised as null. The new page table page needs mapping into the ATU so that the page translation may continue.

(g) *page_cleaned.* The contents of the main memory page frame have been flushed to disk. The frame is now available for other use.

(h) *new_flush_memory.* All memory above the kernel should be flushed to disk. This is probably required because the node is about to be shut down.

(i) *flush_memory.* The main memory page frame indicated in the block has just been flushed to disk. If this frame is not the last in physical memory, the next frame should now be flushed.

(j) *new_flush_disk.* All the pages of the indicated volume must be flushed to disk. This is required because the disk is to be unmounted or as part of a stabilise operation.

(k) *flush_disk.* The main memory page frame indicated in the block has just been flushed to disk. If the frame is not the last in main memory containing a page from the indicated volume, then the next such page should be flushed.

(l) *new_flush_as.* All the pages of the indicated address space must be flushed to disk. This is required because no process has the module stored in the address space open.

(m) *flush_as.* The main memory page frame indicated in the block has just been flushed to disk. If the frame is not the last in main memory containing a page from the indicated address space, then the next such page should be flushed.

(n) *new_address_space_on_way_in.* A new address space root page has been allocated. The page must be initialised as a root page by creating the low address space page table entries and red tape information, and the page must be mapped into the ATU.

(o) *delete_address_space.* The indicated address space must be deleted. This means that all pages from the address space that are currently in main memory must be mapped out, the disk pages allocated to the address space must be added to the disk free list, and that the volume directory for the address space must be removed.

(p) *delete_address_space_next.* The indicated virtual page has been mapped out of main memory (if it was mapped in) and the disk page allocated to the page added to the disk free list. The next page from the address space must now be deleted.

(q) *get_page_for_network*. The network process requires a free main memory page frame. This is required to set up a number of outstanding *receive_requests* to enable receipt of unsolicited messages.

(r) *new_page_for_network*. The virtual memory process is providing the network process with a free main memory page frame.

(s) *here_is_page_for_remote*. The virtual memory process is providing a page for transmission to a remote node.

(t) *receive_request*. The network process is being asked to make the indicated main memory page frame available for receipt of a network message.

(u) *receive_request_reply*. A message has been received and is contained in the virtual memory request block and in the indicated main memory page frame.

(v) *send_request_long*. The network process is requested to transmit a long message. The addressing information, header information, and a pointer to the page of data are included in the virtual memory message block.

(w) *send_request_long_reply*. The network process has transmitted a long message as requested and the status of this transmission is being returned.

(x) *send_request_short*. The network process is requested to transmit a short message. The addressing information and message is included in the virtual memory request block.

(y) *send_request_short_reply*. The network process has transmitted a short message as requested and the status of the transmission is being returned.

(z) *node_on_line*. The network process has been informed that the indicated node has just come on line. The virtual memory process is being informed as required by the stability protocol.

A4.3 Message Passing Data Structures

Data structures maintained at each node are used to implement the MONADS DSM message passing. These data structures, defined in pseudo code, are

```
type logical_node_num = 0 .. 3;
     mm_type_code = (here_i_am, i_am_here_too, request_page,
                     supply_page, request_root_page,
                     supply_root_page, invalid_address_space,
                     req_access_rights, ch_access_rights,
                     return_page, page_received, invalidate_page,
                     page_invalidated, send_page,
                     activate_process, req_volume_location,
                     volume_mounted, module_moved,
                     node_going_down);
     actual_node_num = 0 .. 2**48;
     volume_no_type = 0 .. 2**6
     as_type = 0 .. 2**32;
     page_type = 0 .. 2**16;
     page_no = record
                  page : page_type;
                  address_space : as_type
               end;
     access_rights = (read_only, read_write);
     process_id = integer;
     kernel_process_code = (disk_p, pf_interrupt_p, vm_p, logon_p,
                            timer_p, u_process_sch_p, terminal_p,
                            network_p, idle_p);
     monads_header_type = record
                             destination : logical_node_num;
                             source : logical_node_num;
                             message_type : mm_type_code;
                             source_address : actual_node_num;
                             volume : volume_no_type;
                             requested_as : as_type;
                             supplied_as : as_type;
                             requested_page : page_no
                             access : access_rights;
                             second_node : logical_node_num;
                             {used for mounting and send-to node}
                             reply_to : kernel_process_code;
                             advisory_info : as_type;
                             process_no : process_id
                          end;
     nw_message_type = 0 .. 2**16
     trx_num_type = integer;
     header_ptr = ^message_header_type;
     message_header_type = record
                             destination : actual_node_num;
```

```
                        source : actual_node_num;
                        type : nw_message_type;
                        mess_dest : kernel_process_code;
                        trx_number : trx_num_type;
                        mon_header : monads_header_type;
                        next_header : header_ptr
                    end;
prb_req_type = (new_page_fault, page_on_way_in, hash_table,
                new_page_on_way_in, high_page_table_on_way_ir
                new_page_table_on_way_in, page_cleaned,
                flush_memory, new_flush_disk, flush_disk,
                new_flush_as, flush_as,
                new_address_space_on_way_in,
                delete_address_space, get_page_for_network,
                new_page_for_network, here_is_page_for_remote
                receive_request, receive_request_reply,
                send_request_long, send_request_long_reply,
                send_request_short, send_request_short_reply,
                node_on_line);
page_status_type = integer;
disk_fn_type = (read, write, mount);
disk_num_type = integer;
disk_page_type = integer;
mm_address_type = 0 .. 2**32;
disk_status_type = integer;
rq_status_type = integer;
local_or_remote = (local, remote);
prb_ptr = ^vm_request_block;
vm_message_block = record
                        request_next : prb_ptr;
                        request_type : prb_req_type;
                        address_space : as_type;
                        page_number : page_type;
                        advisory_info : volume_no_type;
                        process_number : process_id;
                        disk_page_link : prb_ptr;
                        process_link : prb_ptr;
                        net_page_link : prb_ptr;
                        page_status : page_status_type;
                        environment_1 : page_status_type;
                        lock_bit : boolean;
                        write_fault_bit : boolean;
                        disk_function : disk_fn_type;
                        disk_number : disk_num_type;
                        disk_page : disk_page_type;
                        physical_address : mm_address_type;
                        disk_status : disk_status_type;
                        request_status : rq_status_type;
                        address_space_disk : as_type;
                        page_disk : page_type;
                        nw_mess_dest : kernel_process_code;
                        trx_number : trx_num_type;
                        block_source : kernel_process_code;
```

```
                    request_from : local_or_remote;
                    monads_info : monads_header_type
                end;
var  vm_message_block_heap : prb_ptr;
     message_header_heap : header_ptr;
     vm_process_queue : prb_ptr;
     network_process_queue : prb_ptr;
     usr_process_scheduler_queue : prb_ptr;
     disk_process_queue : prb_ptr;
```

BIBLIOGRAPHY

1. "PS-algol Reference Manual - fourth edition", University of Glasgow and St Andrews, Persistent Programming Research Report 12/88, 1988.

2. Abramson, D. A. "Hardware Management of a Large Virtual Memory", *Proc. 4th Australian Computer Science Conference*, Brisbane, pp. 1-13, 1981.

3. Abramson, D. A. and Keedy, J. L. "Implementing a Large Virtual Memory in a Distributed Computing System", *Proc. 18th Hawaii Conference on System Sciences*, pp. 515-522, 1985.

4. Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, USENIX, pp. 93-112, 1986.

5. Albano, A., Cardelli, L. and Orsini, R. "Galileo: A Strongly Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, 10(2), pp. 230-260, 1985.

6. Archibald, J. and Baer, J. L. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, 4(4), pp. 273-298, 1986.

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

8. Atkinson, M. and Morrison, R. "Persistent System Architectures", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 73-97, 1989.

9. Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26, 4, Nov., pp. 360-365, 1983.

10. Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), pp. 49-71, 1984.

11. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17(7), pp. 24-31, 1981.

12. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "CMS - A Chunk Management System", *Software Practice and Experience*, 13(3), pp. 259-272, 1983.

13. Bacon, J. M. and Hamilton, K. G. "Distributed Computing with the RPC: the Cambridge Approach", *Distributed Processing*, IFIP, North-Holland, pp. 355-369, 1988.

14. Beloff, B., McIntyre, D. and Drummond, B. "Rekursiv Hardware", Linn Smart Computing Ltd., 1988.

15. Bertis, V., Truxal, C. D. and Ranweiler, J. G. "System/38 Addressing and Authorisation", *I.B.M. System/38 Technical Developments*, pp. 51-54, 1978.

16. Bic, L. and Shaw, A. C. "The Logical Design of Operating Systems", Prentice Hall, ISBN 0-13-540139-9, pp. 55-56, 1988.

17. Birrell, A. D. and Nelson, B. J. "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, 2(1), pp. 39-59, 1984.

18. Black, U. "OSI - A Model for Computer Communications Standards", *International Series*, Prentice Hall, New Jersey, ISBN 0-13-638859-0, pp. 103-156, 1991.

19. Brössler, P., Henskens, F. A., Keedy, J. L. and Rosenberg, J. "Addressing Objects in a Very Large Distributed System", *Proc. IFIP Conference on Distributed Systems*, Amsterdam, pp. 105-116, 1987.

20. Brössler, P. and Rosenberg, J. "Support for Transactions in a Segmented Single Level Store Architecture", *Proceedings of the International Workshop on Computer Architectures to support Security and Persistence of Information*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 319-338, 1990.

21. Brown, A. L. "Persistent Object Stores", Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.

22. Brown, A. L. and Cockshott, W. P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews, PPRR-13, 1985.

23. Brown, A. L., Dearle, A., Morrison, R., Munro, D. and Rosenberg, J. "A Layered Persistent Architecture for Napier88", *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 155-172, 1990.

24. Campbell, R. H., Johnston, G. M. and Russo, V. F. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems", *ACM Operating Systems Review*, 21(3), pp. 9-17, 1987.

25. Challis, M. F. "Database Consistency and Integrity in a Multi-user Environment", *Databases: Improving Useability and Responsiveness*, ed B. Scheiderman, Academic Press, pp. 245-270, 1978.

26. Cheriton, D. R. "VMTP: A Transport Protocol for the Next Generation of Communication Systems", *Proceedings of SIGCOMM 86*, Stowe, Vt., ACM, New York, 1986.

27. Cheriton, D. R. "The V Distributed System", *Communications of the ACM*, 31(3), pp. 314-333, 1988.

28. Cockshott, W. P. "Design of POMP - a Persistent Object Management Processor", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 367-376, 1989.

29. Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), 1984.

30. Comer, D. "Internetworking With TCP/IP - Principles, Protocols and Architecture", Prentice Hall, pp. 49-63, 1988.

31. Connor, R., Brown, A., Carrick, R., Dearle, A. and Morrison, R. "The Persistent Abstract Machine", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 353-366, 1989.

32. Copeland, G., Keller, T., Krishnamurthy, R. and Smith, M. "The Case for Safe RAM", *Proceedings of the 15th International Conference on Very Large Databases*, Amsterdam, pp. 245-270, 1989.

33. Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System", *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.

34. Date, C. J. "An Introduction To Database Systems", *The Systems Programming Series*, vol 1, Addison Wesley, ISBN 0-201-52878-9, 1990.

35. Dearle, A. "On the Construction of Persistent Programming Environments", Ph.D. Thesis, University of St. Andrews, 1988.

36. Delp, G. S. "The Architecture and Implementation of Memnet: a High-Speed Shared-Memory Computer Communication Network", University of Delaware, Udel-EE Technical Report Number 88-05-1, 1988.

37. Denning, P. J. "The Working Set Model for Program Behaviour", *Communications of the ACM*, 11, pp. 323-333, 1968.

38. Denning, P. J. "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, 6(1), pp. 64-84, 1980.

39. Dennis, J. B. and Van Horn, E. C. "Programming Semantics for Multiprogrammed Computations", *Communications of the A.C.M.*, 9(3), pp. 143-145, 1966.

40. Digital Equipment Corporation, Intel Corporation and Xerox Corporation "The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specification", 1980.

41. Dijkstra, E. W. "Co-operating Sequential Processes", *Programming Languages*, ed F. Genuys, Academic Press, London, 1965.

42. Dineen, T. H., Leach, P. J., Mishkin, N. W., Pato, J. N. and Wyant, G. L. "The Network Computing Architecture and System: An Environment for Developing Distributed Applications", *Proceedings, 1987 Summer USENIX Conference*, Phoenix, USENIX Association, Berkeley, California, pp. 385-398, 1987.

43. Edwards, D. B. E., Knowles, A. E. and Woods, J. V. "MU6-G: A New Design to Achieve Mainframe Performance from a Mini-sized Computer", *Computer Architecture News*, 8(3), pp. 161-167, 1980.

44. Evered, M. "LEIBNIZ - A Language to Support Software Engineering", Dr.Ing. Thesis, Faculty of Informatics, Technical University of Darmstadt, 1985.

45. Excelan Inc "EXOS 201 Intelligent Ethernet Controller For Multibus Systems Reference Manual", Publication No. 4200006-00 Rev C, 1986.

46. Excelan Inc "NX 200 Network Executive Reference Manual", Publication No. 4200036-00 Rev A with Release 5.5 Update Notes, 1987.

47. Fabry, R. S. "Capability-Based Addressing", *Communications of the A.C.M.*, 17(7), pp. 403-412, 1974.

48. Farmer, W. D. and Newhall, E. E. "An Experimental Distributed Switching System to Handle Bursty Computer Traffic", *Proceedings of the ACM Symposium on Probabilistic Optimisation of Data Communications Systems*, pp. 1-33, 1969.

49. Fotheringham, J. "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store", *Communications of the ACM*, 4(4), pp. 435-436, 1961.

50. Guy, M. R. "Persistent Store - Successor to Virtual Store", *Proceedings, A Workshop on Persistent Object Systems: their Design, Implementation and Use*, Appin, Scotland, ed R. Carrick and R. Cooper, pp. 266-282, 1987.

51. Hagan, R. A. and Wallace, C. S. "A Virtual Memory System for the Hewlett Packard 2100A", *ACM Computer Architecture News*, 6(5), pp. 5-13, 1979.

52. Halsall, F. "Data Communications, Computer Networks, and OSI", Addison-Wesley, 0-201-18244-0, pp. 447-495, 1988.

53. Harland, D. M. "REKURSIV: Object-oriented Computer Architecture", Ellis-Horwood Limited, 1988.

54. Henskens, F. A., Rosenberg, J. and Hannaford, M. R. "Stability in a Network of MONADS-PC Computers", *Proceedings of the International Workshop on Computer Architectures to support Security and Persistence of Information*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 246-256, 1990.

55. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Fully Transparent Network", University of Newcastle, N.S.W. 2308, Australia, Technical Report 89/7, 1989.

56. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Distributed Shared Memory", *Proceedings of the 14th Australian Computer Science Conference*, Sydney, Australia, pp. 29.1-29.12, 1991.

57. Hsu, M. and Tam, V. "Managing Databases in Distributed Virtual Memory", Harvard University, Technical Report TR-07-88, 1988.

58. ISO "Information Processing Systems - Open Systems Interconnection - Basic Reference Model", ISO 7498, 1984.

59. Johnston, G. M. and Campbell, R. H. "An Object Oriented Implementation of Distributed Virtual Memory", *Proceedings of the USENIX Workshop on Distributed and Multiprocessor Systems*, pp. 39-58, 1989.

60. Keedy, J. L. "An Outline of the ICL2900 Series System Architecture", *Australian Computer Journal*, 9(2), pp. 53-62, 1977.

61. Keedy, J. L. "Paging and Small Segments: A Memory Management Model", *Proc. IFIP-80, 8th World Computer Congress*, Melbourne, Australia, pp. 337-342, 1980.

62. Keedy, J. L. "Support for Software Engineering in the MONADS Computer Architecture", Ph.D. Thesis, Monash University, 1982.

63. Keedy, J. L. "A Memory Architecture for Object-Oriented Systems", *Objekt-orientierte Software und Hardwarearchitekturen*, ed H. Stoyan and H. Wedekind, Teubner-Verlag, Stuttgard, pp. 238-250, 1983.

64. Keedy, J. L. "An Implementation of Capabilities without a Central Mapping Table", *Proc. 17th Hawaii International Conference on System Sciences*, pp. 180-185, 1984.

65. Keedy, J. L., Abramson, D., Rosenberg, J. and Rowe, D. M. "The MONADS Project Stage 2: Hardware Designed to Support Software Engineering Techniques", *Proceedings, 9th Australian Computer Conference*, Hobart, pp. 575-580, 1982.

66. Keedy, J. L., Ramamohanarao, K. and Rosenberg, J. "On Implementing Semaphores with Sets", *The Computer Journal*, 22(2), pp. 146-150, 1979.

67. Keedy, J. L. and Rosenberg, J. "Support for Objects in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, Newcastle, Australia, ed J. Rosenberg and D. M. Koch, Springer-Verlag, 1989.

68. Keedy, J. L. and Rosenberg, J. "Uniform Support for Collections of Objects in a Persistent Environment", *Proceedings of the 22nd Hawaii International Conference on System Sciences*, vol II, ed B. D. Schriver, pp. 26-35, 1989.

69. Keedy, J. L., Rosenberg, J. and Ramamohanarao, K. "On Synchronizing Readers and Writers with Semaphores", *The Computer Journal*, 25(1), pp. 146-150, 1982.

70. Kilburn, T., Edwards, D. B. E., Lanigan, M. J. and Sumner, F. H. "One Level Storage System", *I.R.E. Transactions on Electronic Computation*, EC-11, No. 2, pp. 223-234, 1962.

71. Knapp, V. and Baer, J.-L. "Virtually Addressed Caches for Multiprogramming and Multiprocessor Environments", *Proc., 18th Hawaii International Conference on System Sciences*, pp. 477-486, 1985.

72. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, U.S.A., pp. 99-109, 1990.

73. Levy, H. M. and Lipman, P. H. "Virtual Memory Management in the VAX/VMS Operating System", *Computer*, 15(3), IEEE Computer Society, pp. 35-41, 1982.

74. Li, K. "Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.

75. Linn, C. and Linn, J. "The Carrick-on-Shannon Architecture: A two-level Cache-Coupled Multiprocessor Architecture", *Proc., 18th Hawaii International Conference on System Sciences*, pp. 487-504, 1985.

76. Lorie, R. A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, 2,1, pp. 91-104, 1977.

77. Matthes, F. and Schmidt, J. W. "The Type System of DBPL", *Proceedings of the Second International Workshop on Database Programming Languages*, Portland, Oregan, Morgan Kaufmann, pp. 219-225, 1989.

78. Meijer, A. "Systems Network Architecture", Pitman, London, 1987.

79. Metcalfe, R. M. and Boggs, D. R. "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, 19(7), pp. 395-404, 1976.

80. Morris, R. "Scatter Storage Techniques", *Communications of the ACM*, pp. 38-43, 1968.

81. Morrison, R. and Atkinson, M. P. "Persistent Languages and Architectures", *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and the British Computer Society, pp. 9-28, 1990.

82. Morrison, R., Brown, A. L., Carrick, R., Connor, R., Dearle, A. and Atkinson, M. P. "The Napier Type System", *Persistent Object Systems - Proceedings of the Third International Workshop*, Newcastle, N.S.W., Australia, ed J. Rosenberg and D. Koch, Springer-Verlag, pp. 3-18, 1989.

83. Morrison, R., Brown, A. L., Conner, R. C. H. and Dearle, A. "Napier88 Reference Manual", Universities of Glasgow and St. Andrews, Persistent Programming Research Report PPRR-77-89, 1989.

84. Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R. and van Staveren, H. "Amoeba: A Distributed Operating System for the 1990s", *Computer*, 23(5), IEEE Computer Society, pp. 44-53, 1990.

85. Needham, R. M. and Herbert, A. J. "The Cambridge Distributed Computing System", Addison Wesley, London, 1982.

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

87. Parnas, D. L. "Information Distribution Aspects of Design Methodology", *Proceedings, 5th World Computer Congress*, IFIP, pp. 339-344, 1971.

88. Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12), pp. 1053-1058, 1972.

89. Peterson, J. L. "Myths about the Mutual Exclusion Problem", *Information Processing Letters*, vol. 12, pp. 115-116, 1981.

90. Peterson, J. L. and Silberschatz, A. "Operating System Concepts", Addison Wesley, ISBN 0-201-06198-8, pp. 337-339, 1987.

91. Pierce, J. R. "Networks for Block Switching of Data", *Bell System Technical Journal*, 51, 1972.

92. Popek, J. and Walker, B. "The LOCUS Distributed System Architecture", The MIT Press Series on Computer Systems, pp. 14-17, 40-46, 1985.

93. Randell, B. "A Note on Storage Fragmentation and Program Segmentation", *Communications of the ACM*, 12, 7, pp. 365-369, 1969.

94. Richardson, J. E. and Carey, M. J. "Implementing Persistence in E", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, ed J. Rosenberg and D. M. Koch, Springer-Verlag, pp. 175-199, 1989.

95. Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System", *The Bell System Technical Journal*, 63(6), pp. 1905-1930, 1978.

96. Rosenberg, J. "MONADS-PC Assembler Manual", Department of Computer Science, University of Newcastle, Technical Report 3, 1987.

97. Rosenberg, J. "MONADS-PC Instruction Set", Department of Computer Science, University of Newcastle, Technical Report 1, 1987.

98. Rosenberg, J. "Pascal/M - A Pascal Extension Supporting Orthogonal Persistence", Department of Computer Science, University of Newcastle, Technical Report 89/1, 1989.

99. Rosenberg, J. "The MONADS Architecture - A Layered View", *Proceedings of the 4th International Workshop on Persistent Object Systems*, Martha's Vineyard, U.S.A., Morgan-Kaufmann, 1990.

100. Rosenberg, J. and Abramson, D. A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", *Proc, 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

101. Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", *Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

102. Rosenberg, J. and Keedy, J. L. "Object Management and Addressing in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, Appin, Scotland, 1987.

103. Rosenberg, J., Koch, D. M. and Keedy, J. L. "A Massive Memory Supercomputer", *Proc. 22nd Hawaii International Conference on System Sciences*, vol 1, pp. 338-345, 1989.

104. Ross, D. M. "Virtual Files: A Framework for Experimental Design", University of Edinburgh, CST-26-83, 1983.

105. Schmidt, J. W. "Some High Level Language Constructs for Data of Type Relation", *ACM Transactions on Database Systems*, 2(3), pp. 247-261, 1977.

106. Shapiro, M., Gautron, P. and Mosseri, L. "Persistence and Migration for C++ Objects", *Proceedings, European Conference on Object-Oriented Programming (ECOOP)*, 1989.

107. Smith, A. J. "Bibliography on Paging and Related Topics", *Operating Systems Review*, 12(4), pp. 39-56, 1978.

108. Smith, A. J. "Cache Memories", *ACM Computing Surveys*, 14(3), pp. 437-530, 1982.

109. Stumm, M. and Zhou, S. "Algorithms Implementing Distributed Shared Memory", *Computer*, 23(5), IEEE Computer Society, pp. 54-64, 1990.

110. Sun Microsystems "RPC: Remote Procedure Call Protocol Specification Version 2", *Internet Network Working Group Request for Comments*, No 1057, 1988.

111. Sun Microsystems Inc. "Systems and Networks Administration", Part No: 800-1733-10, Revision A, 1988.

112. Tam, M., Smith, J. M. and Farber, D. J. "A Taxonomy-based Comparison of Several Distributed Shared Memory Systems", *Operating Systems Review*, 24(3), ACM Press, pp. 40-67, 1990.

113. Tanenbaum, A. S. "Operating Systems: Design and Implementation", *International Editions*, Prentice Hall, 0-13-637331-3, pp. 198-226, 1987.

114. Tanenbaum, A. S. "Computer Networks", *International Series*, Prentice Hall, ISBN 0-13-166836-6, pp. 141-148, 1989.

115. Tanenbaum, A. S. "Computer Networks", *International Editions*, Prentice-Hall, ISBN 0-13-166836-6, pp. 454-470, 1989.

116. Tay, B. H. and Ananda, A. L. "A Survey of Remote Procedure Calls", *Operating Systems Review*, 24(3), ACM Press, pp. 68-79, 1990.

117. Thatte, S. M. "Persistent Memory: A Storage Architecture for Object Oriented Database Systems", *Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, pp. 148-159, 1986.

118. Traiger, I. L. "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16(4), pp. 26-48, 1982.

119. Tuke, M. "MONADS-PC Micro-assembler Manual", Department of Computer Science, Monash University, Technical Report 4, 1985.

120. Ungar, D. "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm", *ACM SIGPLAN Notices*, 9(5), pp. 157-167, 1984.

121. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", Department of Computer Science, University of Adelaide, Technical Report PS-1, 1990.

122. Wallace, C. S. "Memory and Addressing Extensions to a HP2100A", *Proceedings, 8th Australian Computer Conference*, Canberra, pp. 1796-1811, 1978.

123. Wang, P. "An Introduction to Berkeley UNIX", Wadsworth Publishing Company, ISBN 0-534-08862-7, 1988.

124. Xerox Corporation "Courier: The Remote Procedure Call Protocol", *Xerox System Integration Standard 038112*, Xerox OPD, 1981.