FORM C

DEAKIN UNIVERSITY

DISPOSITION OF THESIS

I am the author of the thesis entitled Programmer Friendly and Efficient Distributed Shared Memory Integrated into a Distributed Operating System

submitted for the degree of Doctor of Philosophy

and agree to the thesis being made available for such consultation, photocopying or loan as may be approved by the University Librarian, provided that no part of the thesis shall be reproduced without the prior approval of the University Librarian and with the appropriate acknowledgment of the source.

Name J. Silcock
(BLOCK LETTERS)

Signature ..... Signature Redacted by Library .....

Date 9/6/98

DEAKIN UNIVERSITY LIBRARY

TO: ALL USERS OF THIS THESIS

Please sign this form to indicate that you have used this thesis in accordance with the disposition signed by the author of this thesis.

Thank you.

SUE McKNIGHT
University Librarian

Name Signature Date

Programmer Friendly and Efficient Distributed Shared Memory Integrated into a Distributed Operating System

by

Jackie Silcock

BSc, Grad Dip Comp

A thesis submitted in complete
fulfilment of the requirements
for the degree of
DOCTOR OF PHILOSOPHY

May, 1998

School of Computing and Mathematics
Faculty of Science
Deakin University
Waurn Ponds 3217
Australia

FORM B

DEAKIN UNIVERSITY

CANDIDATE'S CERTIFICATE

I certify that the thesis entitled "PROGRAMMER FRIENDLY AND EFFICIENT DISTRIBUTED SHARED MEMORY INTEGRATED INTO A DISTRIBUTED OPERATING SYSTEM" submitted for the degree of DOCTOR OF PHILOSOPHY is the result of my own research and that this thesis in whole or in part has not been submitted for an award including a higher degree to any other university or institution.

Signature Redacted by Library

Jackie Silcock

May 27, 1998

Acknowledgements

I would like to thank my supervisor, Andrzej Goscinski, for his assistance in this research. With patience, encouragement and constant support he has taught me a great deal not only about research but also about dedication.

I would like to thank my colleagues and friends Damien De Paoli and Michael Hobbs for their good humoured assistance. Thank you also to all of the members of the RHODOS group: Robert Dew, Christopher MacAvaney, Justin Rough and Greg Wickham.

Last but by no means least I would like to thank my family Pete, Michael, Caroline and Richard, who have always supported me.

Publications from this Thesis

Refereed Papers:

[Silcock and Goscinski 97] J. Silcock and A. Goscinski, *Invalidation-Based Distributed Shared Memory Integrated into a Distributed Operating System*, Proceedings of IASTED International Conference Parallel and Distributed Systems (Euro-PDS'97), June 1997.

[Silcock and Goscinski 97] J. Silcock and A. Goscinski, *Performance Studies of Distributed Shared Memory Embedded in the RHODOS' Operating System*, Proceedings of The 4th Australasian Conference on Parallel and Real-Time Systems (PART'97), September, 1997.

[Silcock and Goscinski 97] J. Silcock and A. Goscinski, *Update-Based Distributed Shared Memory Integrated into RHODOS' Memory Management*, Proceedings of Third International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'97), December 1997.

[Silcock and Goscinski 98] J. Silcock and A. Goscinski, *The Influence of the Ratio of Processes to Workstations on the Performance of DSM*, Proceedings of The 21st Australasian Computer Science Conference (ACSC'98), February, 1998.

[Goscinski and Silcock 98] A. Goscinski and J. Silcock, *Easy Programming and High Performance of Parallel Applications in the RHODOS DSM System*, Submitted to The Second European IASTED International Conference on Parallel and Distributed Systems (Euro-PDS'98), July 1998.

[Silcock and Goscinski 98] J. Silcock and A. Goscinski, *The RHODOS DSM System*, Submitted to Microprocessors and Microsystems, 1998.

Technical Reports

[Silcock and Goscinski 95] J. Silcock and A. Goscinski, *Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems*, Technical Report, School of Computing and Mathematics, Deakin University. TR C95/20 June 1995.

[Silcock 95] J. Silcock, *Distributed Shared Memory: A Survey*, Technical Report, School of Computing and Mathematics, Deakin University. TR C95/22, June 1995.

[Silcock and Goscinski 96] J. Silcock and A. Goscinski, *Logical design for Distributed Shared Memory on RHODOS*, Technical Report, School of Computing and Mathematics, Deakin University, TR C95/35, October 1995.

[Silcock and Goscinski 96] J. Silcock and A. Goscinski, *Space Based Distributed Shared Memory on RHODOS*, Technical Report, School of Computing

and Mathematics, Deakin University, TR C96/3, February 1996.

[Silcock 96] J. Silcock, *A Consistency Model for Distributed Shared Memory on RHODOS among Shared Memory Consistency Models,* Technical Report, School of Computing and Mathematics, Deakin University, TR C96/6, April 1996.

[Silcock and Goscinski 96] J. Silcock and A. Goscinski, *Invalidation-Based Distributed Shared Memory on RHODOS,* Technical Report, School of Computing and Mathematics, Deakin University, TR C96/13, November 1996.

[Silcock and Goscinski 97] J. Silcock and A. Goscinski, *Update-based Distributed Shared Memory Integrated into RHODOS' Memory Management,* Technical Report, School of Computing and Mathematics, Deakin University. TR C97/03, March 1997.

[Silcock et al. 97] J. Silcock, A. Goscinski and J. Goldberg, *Performance Studies of RHODOS' Operating System Embedded Distributed Shared Memory,* Technical Report, School of Computing and Mathematics, Deakin University. TR C97/08, May 1997.

[Silcock and Goscinski 97] J. Silcock and A. Goscinski, *Programming and Performance Studies of Parallel Applications in a DSM Environment,* Technical Report, School of Computing and Mathematics, Deakin University. TR C97/09, May 1997.

Table of Contents

Chapter 1 Introduction 1
1.1 Research Genesis 1
1.2 Research Aim 7
1.3 Research Methods 9
1.4 The Contents of the Thesis 10

Chapter 2 Basic Concepts and Related Work 13
2.1 Introduction 13
2.2 Distributed Shared Memory 13
2.2.1 Shared Memory vs Message Passing Programming Models 15
2.2.2 Factors Affecting DSM 16
2.3 DSM Research Issues 16
2.3.1 Consistency 17
2.3.2 Granularity 21
2.3.3 Data Location and Access 21
2.3.4 Synchronisation 22
2.4 Current Implementations of DSM 25
2.4.1 IVY 26
2.4.2 Mirage 27
2.4.3 Clouds 28
2.4.4 Munin 28
2.4.5 Midway 29
2.4.6 TreadMarks 30
2.5 Conclusion 31

Chapter 3 Synthesis of a DSM System Integrated into a Distributed Operating System 34
3.1 Introduction 34
3.2 DSM Design Requirements 35
3.3 Parallelism to be supported by the proposed DSM System 37
3.4 The Location of the DSM System 40
3.4.1 Current DSM Architectures 41
3.4.2 Distributed Operating System Based DSM Architecture 42
3.4.3 Memory Management Based DSM System 44
3.5 Services provided by the DSM system 46
3.5.1 Operating system integrated DSM 47
3.5.1.1 Starting the DSM system: General Design 47
3.5.1.2 DSM System's Mechanisms: General Design 50

3.5.1.3 DSM System Functioning: General Design 51
3.5.2 Events and Reactions of the DSM System 53
3.5.2.1 DSM system event protocols 53
3.5.2.2 Events and reactions of the DSM system 54
3.5.3 Initialisation 61
3.5.4 Data and Semaphore Location and Access 65
3.5.5 Synchronisation and Computation Co-ordination 66
3.5.6 Consistency Model 69
3.5.6.1 Write-Invalidate DSM system 70
3.5.6.2 Write-Update DSM System 74
3.6 Conclusions 76

Chapter 4 Distributed Shared Memory integrated into the RHODOS Distributed Operating System 79
4.1 Introduction 79
4.2 RHODOS Distributed Operating System 80
4.2.1 Process Manager 83
4.2.2 Interprocess Communication (IPC) Manager 83
4.2.3 Space Manager 84
4.3 Memory Management in RHODOS 85
4.3.1 RHODOS Spaces 85
4.3.2 The Space Manager 85
4.3.3 Space Manager Primitives used for DSM 86
4.4 Development of Automatic Initialisation System 87
4.4.1 Data Initialisation 87
4.4.2 DSM Parallel Process Initialisation 90
4.4.3 DSM System Initialisation Semantics 91
4.5 DSM in RHODOS 96
4.5.1 A Simple Example of Producer-Consumer Code using RHODOS' DSM 96
4.5.2 The DSM Table 97
4.5.3 Write-Invalidate Based DSM in RHODOS 99
4.5.4 Write-Update Based DSM in RHODOS 103
4.5.5 Semantics of barriers in write-invalidate and write-update based DSM in RHODOS 106
4.6 Physically shared memory in RHODOS' write-update based DSM 110
4.7 Conclusions 113

Chapter 5 Programming and Performance Studies of RHODOS DSM 115
5.1 Introduction 115
5.2 Programming Aspects and Performance Studies 116
5.2.1 Hardware Platform 116
5.2.2 Test Applications 116
5.2.3 Red-Black Successive Over Relaxation (SOR) 118

5.2.4 Matrix Multiplication 122
5.2.5 Jacobi 124
5.2.6 Quicksort 126
5.2.7 Travelling Salesman Problem (TSP) 128

5.3 The Outcomes 129
5.3.1 Experience Gained from Programming of Parallel Applications for RHODOS DSM 130
5.3.2 Comparative Evaluation 132

5.4 The Influence of Physically Shared Memory on the Performance of DSM. 134
5.4.1 The Scope of Study 135
5.4.2 Multiprogramming and Physically Shared Memory 135
5.4.3 Travelling Salesman Problem 136
5.4.4 Matrix Multiplication 137

5.5 Conclusions 140

Chapter 6 Conclusions and Further Work 142
6.1 Conclusions 142
6.2 Further Work 145

Bibliography 147

List of Figures

2.1 Distributed Shared Memory 14
2.2 Range of Consistency Models 20
2.3 Taxonomy of synchronisation 23
3.1 Generic Code using SPMD Model 38
3.2 Generic Code using MPMD Model 39
3.3 Network operating system based architecture 41
3.4 Architecture of distributed operating system which includes DSM system 43
3.5 Architecture of client-server model and microkernel based distributed operating system incorporating DSM system 45
3.6 DSM system parent protocol 55
3.7 DSM system child protocol 56
3.8 Pseudocode for Parent and Child Initialisation 61
3.9 Pseudocode for Parent Process' Parallel Initialisation 62
3.10 Pseudocode for DSM Memory Create 62
3.11 Pseudocode for Child Process' Parallel Execution 63
3.12 Pseudocode for memory_attach 64
3.13 Pseudocode for Page Copy Fault 66
3.14 Pseudocode for Wait 68
3.16 Pseudocode for DSM Barrier Request 68
3.15 Pseudocode for Signal 69
3.17 Pseudocode for DSM Barrier 69
3.18 Pseudocode for Exception Handler 70
3.19 Pseudocode for Page Copy Write Fault 71
3.20 Pseudocode for Page Invalidate Request 72
3.21 Pseudocode for Page Invalidate 72
3.22 Pseudocode for Page Copy Write 73
3.23 Pseudocode for Distribute Diffs 75
3.24 Pseudocode for Update Pages 76
4.1 The Process Layers of RHODOS 81

4.2 RHODOS as used for DSM 84
4.3 The User Process Space of a process using RHODOS DSM 88
4.4 The dsm_semaphore_table Data Structure 89
4.5 The dsm_barrier_table Data Structure 90
4.6 Semantics of Automatic Initialisation on RHODOS 92
4.7 DSM Initialisation code for Parent Process 93
4.8 DSM Initialisation code for Child Process 95
4.9 Producer-Consumer Code segment 97
4.10 The DSM table as used for write-invalidate DSM 98
4.11 The DSM table as used for write-update DSM 98
4.12 Design of write-invalidate based DSM on RHODOS 100
4.13 Logical Design of semaphores and memory consistency in write-update based DSM on RHODOS 103
4.14 Diff Message Structure 105
4.15 Overview of Diff Message generation 105
4.16 Code skeleton using barriers 107
4.17 Logical Design of Barriers in write-invalidate based DSM on RHODOS 108
4.18 Logical Design of barriers in write-update based DSM on RHODOS 109
4.19 Semantics of Physically Shared Memory on RHODOS 111
5.1 Pseudocode of the SOR algorithm 119
5.2 Data decomposition algorithm 120
5.3 Speedup for SOR using RHODOS' write-invalidate DSM on a matrix of 64 × 2048 elements 120
5.4 Speedup for SOR using RHODOS' write-update based DSM on matrix of 128 × 128 elements 121
5.5 Pseudocode for Matrix Multiplication 122
5.6 Speedup for Matrix Multiplication using write-invalidate and write-update based DSM on a matrix of 256 × 256 elements 123
5.7 Pseudocode for the Jacobi algorithm 125
5.8 Speedup for Jacobi on matrices of 64 × 2048 (INVALIDATE) and 60 × 1024 (SMALL-INVALIDATE and UPDATE) elements 126
5.9 Pseudocode for Quicksort 127

5.10 Speedup for Quicksort using invalidation and update-based DSM on an array of 256K elements 128
5.11 Pseudocode for TSP 130
5.12 Speedup for TSP using invalidation and update-based DSM for an 18-city tour 131
5.13 Speedup vs number of processes for ratio of processes to workstations is 1:1 and greater than 1:1 for the Travelling Salesman Problem 137
5.14 Speedup vs number of workstations for ratio of processes to workstations is 1:1 and greater than 1:1 for the Travelling Salesman Problem 138
5.15 Speedup vs number of processes for ratio of processes to workstations is 1:1 and greater than 1:1 for the Matrix Multiplication 138
5.16 Speedup vs number of workstations for ratio of processes to workstations is 1:1 and greater than 1:1 for the Matrix Multiplication 139

Abstract

Distributed Shared Memory (DSM) provides programmers with a shared memory environment in systems where memory is not physically shared. Clusters of Workstations (COWs), an often untapped source of computing power, are characterised by a very low cost/performance ratio. The combination of Clusters of Workstations (COWs) with DSM provides an environment in which the programmer can use the well known approaches and methods of programming for physically shared memory systems and parallel processing can be carried out to make full use of the computing power and cost advantages of the COW.

The aim of this research is to synthesise and develop a distributed shared memory system as an integral part of an operating system in order to provide application programmers with a convenient environment in which the development and execution of parallel applications can be done easily and efficiently, and which does this in a transparent manner. Furthermore, in order to satisfy our challenging design requirements we want to demonstrate that the operating system into which the DSM system is integrated should be a distributed operating system.

In this thesis a study into the synthesis of a DSM system within a microkernel and client-server based distributed operating system which uses both strict and weak consistency models, with a write-invalidate and write-update based approach for consistency maintenance is reported. Furthermore a unique automatic initialisation system which allows the programmer to start the parallel execution of a group of processes with a single library call is reported. The number and location of these processes are determined by the operating system based on system load information.

The DSM system proposed has a novel approach in that it provides programmers with a complete programming environment in which they are easily able to develop and run their code or indeed run existing shared memory code. A set of demanding DSM system design requirements are presented and the incentives for the placement of the DSM system within a distributed operating system and in particular in the memory management server have been reported. The new DSM system concentrates on an event-driven set of cooperating and distributed entities, and a detailed description of the events and reactions to these events that make up the

operation of the DSM system is then presented. This is followed by a pseudocode form of the detailed design of the main modules and activities of the primitives used in the proposed DSM system.

Quantitative results of performance tests and qualitative results showing the ease of programming and use of the RHODOS DSM system are reported. A study of five different applications is given and the results of tests carried out on these applications together with a discussion of the results are given. A discussion of how RHODOS' DSM allows programmers to write shared memory code in an easy to use and familiar environment and a comparative evaluation of RHODOS DSM with other DSM systems is presented. In particular, the ease of use and transparency of the DSM system have been demonstrated through the description of the ease with which a moderately inexperienced undergraduate programmer was able to convert, write and run applications for the testing of the DSM system. Furthermore, the description of the tests performed using physically shared memory shows that the latter is indistinguishable from distributed shared memory; this is further evidence that the DSM system is fully transparent. This study clearly demonstrates that the aim of the research has been achieved; it is possible to develop a programmer friendly and efficient DSM system fully integrated within a distributed operating system.

It is clear from this research that client-server and microkernel based distributed operating system integrated DSM makes shared memory operations transparent and almost completely removes the involvement of the programmer beyond classical activities needed to deal with shared memory. The conclusion can be drawn that DSM, when implemented within a client-server and microkernel based distributed operating system, is one of the most encouraging approaches to parallel processing since it guarantees performance improvements with minimal programmer involvement.

FORM F

DEAKIN UNIVERSITY

Plain Language summary of thesis submitted for the degree of Doctor of Philosophy

Title: Programmer Friendly and Efficient Distributed Shared Memory Integrated into a Distributed Operating System.

Distributed Shared Memory (DSM) provides programmers with a shared memory environment in systems where memory is not physically shared. Clusters of Workstations (COWs), an often untapped source of computing power, are characterised by a very low cost/performance ratio. The combination of COWs with DSM provides an environment in which the programmer can use the well known approaches and methods of programming for physically shared memory systems and parallel processing can be carried out to make full use of the computing power and cost advantages of the COW.

The DSM system proposed and built in this research, to satisfy a set of demanding design requirements, has a novel approach in that it provides programmers with a complete programming environment in which they are easily able to develop and run their code or indeed run existing shared memory code.

Quantitative results of performance tests and qualitative results showing the ease of programming and use of the RHODOS DSM system are reported. It is clear from this research that distributed operating system integrated DSM makes shared memory operations transparent and almost completely removes the involvement of the programmer beyond classical activities needed to deal with shared memory.

The conclusion can be drawn that DSM, when implemented within a distributed operating system, is one of the most encouraging approaches to parallel processing since it guarantees performance improvements with minimal programmer involvement.

Signed:..... **Name**: Jackie Silcock

Supervisor: Professor Andrzej Goscinski

Date: May 27, 1998

Chapter 1 Introduction

Parallel processing on supercomputers and massive parallel processor systems suffers from the lack of parallel software and high cost/performance ratio. Distributed Shared Memory (DSM) provides programmers with a shared memory environment in systems where memory is not physically shared. Clusters of Workstations (COWs), an often untapped source of computing power, are characterised by a very low cost/ performance ratio. The combination of Clusters of Workstations (COWs) with DSM provides an environment in which the programmer can use well known approaches and methods of programming for centralised systems and parallel processing can be carried out to make full use of the computing power and cost advantages of the COW.

1.1 Research Genesis

Parallel processing involves dividing a task into sub-tasks which are executed on separate processors. Tasks can be divided at the instruction level or at the function or program level. Tasks divided at the instruction level can be grouped into four basic models based on the combination of the data and instruction stream used in the computation. These models are: single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD) and multiple instruction multiple data (MIMD).

The SIMD and MIMD models are the basis for parallel computer systems which have been embodied in four hardware platforms: Symmetric Multiprocessors, Massively Parallel Processors, Distributed Shared Memory Systems and Clusters of Workstations (COWs). If the instruction stream is viewed as coarse grained i.e., the task is divided at the function or program level instead of the instruction level, two new computational models can be defined. These new models are single program multiple data (SPMD) and multiple program multiple data (MPMD). COWs are ideally suited to support the coarse grained parallelism of SPMD and MPMD models [Goscinski 97].

COWs, which are made up of off-the-shelf hardware, are relatively inexpensive compared with supercomputers and massive parallel processors. Thus, COWs can be

used as a less costly means of gaining the power of a supercomputer. COWs are a source of huge but very often underutilised computing power because many of the workstations are frequently lightly loaded or even idle. Parallel processing can be improved through the use of COWs by spreading the workload for a single application over a number of workstations. In fact, some applications can be executed efficiently on COWs that previously required the use of a supercomputer. However, in order to tap this often unused resource programmers need to be able easily and successfully to write the applications which will run in parallel. They need to be able to concentrate on algorithm development without being concerned about the type of system on which their code will run and parallelism management.

One of the following programming approaches can be used to support parallel processing on a COW [Goscinski 97]:

- Parallel or distributed programming languages, which require the programmer to identify the units of parallelism explicitly; the language then provides primitives to ensure synchronisation and interprocess communication;
- Parallel programming tools which allow the programmer to indicate the unit they require to be executed in parallel. The tools employ library routines which call the operating system to invoke synchronisation and communication functions;
- Parallelising compilers which identify units of parallelism in a piece of sequential code. The compiler then inserts communication and synchronisation points; and
- Sequential programming language plus software of a DSM system which allows programmers to write code as though the COW had a single global physically shared memory.

DSM, compared to the other approaches given above, offers the best environment for parallel processing because programmers can use the easy and familiar shared memory programming model. Programmers using DSM view the memory as though it were a single globally shared memory although the memory is, in fact, physically distributed.

DSM is not only an abstraction for shared memory, but also an alternative, higher abstraction form of interprocess communication to message passing and remote procedure calls (RPC). Message passing is the basis of interprocess communication in most distributed systems. It is at the lowest level of abstraction and requires the application programmer to be able to identify the destination process, the message, the source process and the data types expected from these processes. Message passing leaves the programmer with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) relieves this burden by increasing the level of abstraction and providing semantics similar to local procedure calls.

For the shared memory model the assumption is made that there is a single shared memory, thus any operation performed on the memory is immediately visible to all other processes. Processes do not need to communicate explicitly with one another. They communicate implicitly through shared variables. A programmer writing code for this type of model needs to use synchronisation primitives to prevent competing accesses to the same memory address. Application programmers have traditionally considered the shared memory programming model easier to use than the message passing model because it allows them to write their code without being concerned about this explicit movement of data from one process to another.

In COWs the memory is physically distributed and all interprocess communication must take place through message passing. This places an additional burden on programmers and can divert them from the most important task of development of the algorithm. DSM is a higher abstraction of interprocess communication which exploits shared memory. Thus, DSM allows programmers to use the concept of shared memory when writing programs. DSM is a paradigm where memory, although distributed over a network of autonomous computers, gives the appearance of being centralised. The memory is accessed through virtual addresses, thus processes are able to communicate by reading and modifying data which are directly addressable. Programmers are able to access complex data structures. Thus, DSM relieves programmers of the many concerns by abstracting away from the complexities and drawbacks of message passing and RPC. In summary, DSM offers the two advantages of being able to write programs using the preferred shared memory

model on a COW which is cheaper and scales to larger numbers than shared memory machines.

Despite all of its advantages, DSM provides significant challenges for system designers because, if it is to be considered a viable option for application programmers, the performance of the system must not degrade significantly when DSM is being used. Thus, the important issue in the design of a DSM system is to balance the dual needs of improving the performance of the system with making it programmer friendly and transparent. However, there are several problems in achieving this goal.

Different levels of transparency can be provided by DSM implemented in systems where the memory is physically shared or distributed. User level transparency allows the application programmer to write code for and use the DSM system without any knowledge of its implementation. Binary transparency can be achieved between a system where the memory is physically shared and one where the memory is distributed as in Shasta [Scales and Gharachorloo 97].

The simplest form of DSM involves having a single copy of the shared data which is migrated from workstation to workstation whenever a read or write operation is performed. However, it is more convenient to replicate the shared data on all workstations. The shared data remains consistent if multiple processes perform read operations on the replicated copies but when write operations are performed a mechanism is required to keep the replicated copies consistent. The consistency model governs how and when this replicated data is made consistent. Thus, the most important decision when designing a DSM system is the consistency model to be employed. Consistency models can be broadly classified as strict and weak models. The strict models attempt to emulate the behaviour of memory in a uniprocessor system. In these models changes to memory are propagated to all other workstations immediately after they are carried out. On the other hand, in the weak models the propagation of memory updates is delayed until the updated memory is required by another process.

In order to achieve consistency some mechanism is required to ensure the serialisation of write operations such that any subsequent operations on the same

memory location access the updated data. There are two approaches used to serialise writes in parallel processes, write-invalidate and write-update. In the write-invalidate model multiple readable pages can exist in the system at any time. An attempt to perform a write operation on one of the readable copies of the page causes all copies of the page to be invalidated. In the write-update model multiple readable and writable copies of a page can exist at any time. Changes made to a page are placed in a message and propagated to all workstations. These consistency related messages are the major overhead in DSM systems. In write-invalidate systems messages are required to invalidate all copies of a page prior to a write operation and later a message is needed to retrieve a new copy of the invalidated page. Write-update systems require a large number of messages to maintain the consistency of the data; in these systems a message containing updates must be sent to each workstation in the COW which has a copy of the shared data. However, in write-update systems it is possible, depending on the consistency model used, to delay the propagation of these update messages until the updated data is required.

Much of the effort in DSM research has concentrated on improving the performance of the DSM system, particularly in the area of more relaxed consistency models rather than researching a more integrated DSM design [Iftode and Singh 97].

Treadmarks [Keleher 95] uses a new consistency model, lazy release consistency, which delays updating the shared memory until entry to a critical region in which that data will be used. Lazy release consistency appears to improve the performance of applications. However, the implementation requires specialised hardware since a large amount of memory is required on each workstation to store unapplied updates. In addition, a presumably time-consuming garbage collection function must run at intervals in order to clear the memory of a backlog of updates. The frequency of the garbage collection runs is related to the size of the local memory, the smaller the memory the more often this function must run. The performance improvements for Treadmarks have been achieved using a very fast network making us unsure whether these improvements would be exhibited in a COW with a normal ethernet connection and normal sized memory on each workstation.

In order to extract the best performance from the DSM system many

implementations require programmers to go beyond their normal shared memory based practices. In particular, Midway [Bershad et al. 93] requires programmers to label all DSM variables and to explicitly associate each of them with a different synchronisation variable, while Munin [Carter et al. 95b] requires programmers to decide which consistency protocol should be used for each shared variable according to its access patterns. These approaches clearly require the programmer to gain additional skills and to have a thorough insight into the implementation of the DSM system they are using in order to make successful use of the DSM system. In fact, programmers require a deeper than usual insight into the application's data sharing patterns than would normally be expected in shared memory programming.

All DSM systems are currently implemented as a separate piece of software on top of or as an addition to an existing operating system. The existing operating system design have influenced some of the design decisions for these DSM systems. IVY, for example, uses eventcounts for synchronisation because they were already implemented in the Aegis operating system on top of which IVY was prototyped [Lamport 79]. This means that the design of the DSM may be compromised by the design of the operating system. In addition, because the DSM is placed on top of the operating system it may not be able to use all of the low-level operating system functions in its implementation.

Current DSM systems require the user to know in advance of commencing execution the number of workstations that will be executing the application and the location of the workstations. This requires the application programmer to have knowledge of the network size, its load and the distribution of the load. In a modern distributed operating system the load information is used by the operating system to determine the scheduling of the processes. It follows that the operating system, not the application programmer, should handle the placement of the DSM processes.

Many modern operating systems tend to pay more attention to performance than to ease of use for the programmer. Furthermore, many operating systems which run in a distributed environment require application programmers to use message passing for interprocess communication as such they force application programmers to use a difficult, unpopular programming model when writing parallel application code. These operating systems do not provide programmers with a convenient execution

environment for parallel processing. For instance, the V-System, on top of which Munin was implemented, offers message passing and remote procedure calls as interprocess communication mechanisms available for programmers. Treadmarks is built on top of the UNIX operating system. In UNIX programmers must use sockets for interprocess communication between processes located on workstations which are remote from one another. Midway was built on top of the Mach operating system, which provides message passing for interprocess communication. In summary, programmers wishing to use DSM in any of these operating system environments must explicitly choose to run a separate piece of software on top of the operating system in order to gain the benefits of an easy parallel programming environment. These benefits should be provided by the operating system.

1.2 Research Aim

Parallel processing is a means of improving the execution time of applications. COWs are a source of underutilised computing resources. The combination of these two is potentially a very powerful means of carrying out diverse computations that previously required supercomputers. DSM is a very promising programming approach which can be used to support parallel processing on COWs. However, several problems need to be overcome if this potential is to be fully realised. Since the major overhead of DSM systems is communication, it is important that this overhead be minimised in order to provide an efficient system. Furthermore, programmers writing code for a DSM system should not be forced to have additional input over and above the normal requirements in a system which has physically shared memory. The user of the DSM system should not be aware that the memory they are using is not local; in other words the system should provide transparency. Both of these problems would be solved by integrating the DSM system into the operating system. Efficiency would be improved because the DSM system would be able to use the low level operating system functions and transparency achieved because the DSM software would be hidden completely within the operating system.

Since one of the aims of an operating system is user convenience, simply providing an efficient, transparent DSM is not enough, a whole parallel processing environment is required. Thus, the operating system into which the DSM is integrated

should provide application programmers with an environment in which it is easy to write and run their parallel applications whether they write new or re-use existing shared memory code. COWs do not have any physically shared memory. Therefore, the operating system should itself provide support for a transparent and efficient distributed shared memory to enable programmers to take the maximum advantage of the computing power, availability, flexibility and cost of COWs. Furthermore, the operating system itself needs to be able to support remote process creation, efficient remote interprocess communication and transparency. These factors point to the fact that a distributed operating system is the proper choice of operating system. Essentially a new class of distributed operating systems is required, one that provides its users with an environment in which they can easily write and run parallel programs.

The aim of this research is to synthesise and develop a distributed shared memory system as an integral part of an operating system to provide a transparent and efficient environment for the development and execution of parallel applications. Furthermore, the operating system into which the DSM system is integrated should be a distributed operating system.

It is important to resolve whether the operating system based approach is fully feasible and whether it is able to provide results comparable with or even better than DSM systems which have been developed as a separate piece of software which runs on top of an operating system. The choice of the type of operating system is very important. Current development trends are toward microkernel and client-server based distributed operating systems which provide the flexibility and configurability necessary for the integration of an efficient DSM system. In addition, the research must show that an operating system integrated DSM system is transparent at user level, while maintaining the highest efficiency and scalability possible. This feature will guarantee that operating system integrated DSM allows programmers to write shared memory code in a programmer friendly and familiar environment. In order to achieve this aim it is important to show that the implementation of DSM within the operating system takes distributed operating system design to a new level; a level in which programmers can concentrate on algorithm development free of the concerns of message passing or knowledge of the implementation of the DSM system which they

use.

To achieve this aim it is necessary to carry out the following tasks:

1. Synthesis of a new and original DSM system within a microkernel and client-server based distributed operating system. The DSM system synthesised will:
   - use both strict and weak consistency models, with a write-invalidate and write-update based approach for consistency maintenance;
   - automatically initialise the parallel applications from a single parent process by starting processes on remote workstations. The placement and number of remote processes will be based on the system load information; and
   - provide user level transparency (binary level transparency is beyond the scope of this study).
2. Implementation and testing of the DSM system within the RHODOS distributed operating system in order to demonstrate the feasibility of the approach developed in Task 1.
3. Qualitative assessment of the easiness of programming and use of the RHODOS DSM system and performance studies on the DSM system using applications which use both the SPMD and MPMD computational models in order to demonstrate that the design requirements have been achieved.

1.3 Research Methods

The research aim stated in Section 1.2 will be achieved by employing the principles of experimental computer science [Snyder 95]. Distributed shared memory will be implemented within the RHODOS distributed operating system as a "proof-of-concept" to demonstrate the feasibility of providing an operating system environment which support the proposed DSM system. Furthermore, the proposed implementation will demonstrate the feasibility of implementing DSM within this operating system environment. Further, as "proof-of-performance" the proposed implementation will be used to test the notion that, having built the appropriate environment, i.e. the operating system embedded DSM, the performance levels will be at least as good as those for

other DSM systems which are built on top of operating systems.

The experimental applications proposed for use in the performance tests will be applications employed by other researchers to test their implementations of DSM.

1.4 The Contents of the Thesis

In order to demonstrate that the research aim have been achieved this thesis is structured as follows.

In Chapter 2 the general principles of DSM and related work are discussed. A general introduction to DSM is given followed by an analysis of the major issues in DSM research; granularity, data location and access, computation co-ordination, and consistency models. This leads into a description of the DSM systems which have been developed: IVY, Mirage, Clouds, Munin, Midway and TreadMarks. These systems are discussed and analysed to identify the research areas which have been overlooked.

Chapter 3 contains a study into the synthesis of a DSM system within a microkernel and client-server based distributed operating system which uses both strict and weak consistency models, employing a write-invalidate and write-update coherence protocols, respectively. The DSM system proposed in this chapter has a novel approach in that it provides programmers with a complete programming environment in which they are easily able to develop and run their code or indeed run existing shared memory code. Furthermore, an automatic initialisation system is proposed which allows programmers to initialise the whole system with a single library call. This initialisation system relieves the application programmer of the job of having to decide the number of processes required to execute an application and the workstations on which these processes should execute. The operating system decides the number and location of remote processes that will execute the application based on the system load and then starts their execution. The chapter starts with the introduction of a set of demanding DSM system design requirements followed by a discussion of the two models of parallelism to be supported by the DSM system. Next, the incentives for the placement of the DSM system within a distributed operating system and in particular in the memory management server have been reported. This is followed by a description at a high level of the functioning of the DSM system within a client-server

and microkernel based distributed operating system. The synthesis of a new DSM system, which concentrates on an event-driven set of cooperating and distributed entities, and a detailed description of the events and reactions to these events that make up the operation of the DSM system is then presented. Finally, a pseudocode form of the detailed design of the main modules and activities of the primitives used in the proposed DSM system is presented.

In Chapter 4 a description of the RHODOS operating system and its components, paying particular attention to those components which are directly involved in and support the DSM system, are described. This is followed by a discussion of the design and semantics of the automatic initialisation in which the system load information is used to decide the number and placement of the remote processes, in fact the system actually starts the child processes on the remote workstations. Programmers can insert a simple library call to initialise parallel execution on a number of workstations. A discussion of the design and semantics of the write-invalidate (used to implement the sequential consistency model) and write-update (used to implement the release consistency model) based DSM systems and their relationship with the semaphore-type synchronisation designed follows. The semantics of barriers in RHODOS are presented next. Finally, a discussion of the manner in which RHODOS' operating system integrated DSM deals with physically shared memory, highlighting the fact that distributed and physically shared memory are employed by users in precisely the same manner, is given. This chapter demonstrates the feasibility of the DSM system integrated within a distributed operating system proposed in Chapter 3.

In Chapter 5 the quantitative results of the performance tests and qualitative results showing the ease of programming and use of the RHODOS DSM system are presented. Initially, a study of five different applications is given and the results of tests carried out on these applications together with a discussion of the results for these tests are discussed. This is followed by a discussion of how RHODOS' DSM allows programmers to write shared memory code in an easy to use and familiar environment and a comparative evaluation of RHODOS DSM with other DSM systems. In particular, the qualitative tests carried out and lessons learned by allocating a

moderately inexperienced undergraduate programmer the task of writing code for some of these applications are presented. Furthermore, the results are given for experiments carried out using both the DSM system and the system where the processes physically share the memory. These results are then discussed. This chapter clearly demonstrates that the aim of the research has been achieved; it is possible to develop a programmer friendly and efficient DSM system fully integrated within a distributed operating system.

Chapter 6 contains the conclusions of this research and the future work to be carried out.

Chapter 2 Basic Concepts and Related Work

2.1 Introduction

As stated in Chapter 1 the goal of this research is to synthesise and develop a Distributed Shared Memory system as an integral part of a distributed operating system in order to provide application programmers with a convenient and high performance environment which allows them to develop and execute parallel applications. Furthermore, the DSM must function in a transparent and efficient manner. In order to justify this goal it is necessary to look at the issues in DSM system research and at current DSM implementations to highlight the areas of research that have been neglected by other researchers. Moreover, it is essential to place this research into perspective with respect to the work of other researchers.

In this chapter the general principles of DSM are examined. Consequently, the chapter starts with a general introduction to DSM. This is followed by an analysis of the issues in DSM research which can be broadly divided into consistency models, granularity, data location and access and synchronisation.

Having discussed the research issues, the most representative DSM systems which have been developed are presented and analysed in an attempt to identify the research areas which have been overlooked. To achieve this the advantages and disadvantages of other DSM implementations are examined. The systems discussed are IVY, Mirage, Clouds, Munin, Midway and TreadMarks.

2.2 Distributed Shared Memory

DSM has been an active area of research since the mid 1980s. Some of the first work in this area was carried out by Libes [Libes 85], who implemented a shared variable scheme at user level on a Unix system. However the first true software implementation of DSM appeared in 1986 when IVY was developed [Li 86].

DSM is an abstraction for shared memory in a system with no physically shared memory. The logical view is one of several machines sharing a centralised memory.

The physical situation, however, is quite different — the distributed machines have their own local memory and are connected to one another through the interconnection network. With the addition of appropriate hardware or software these individual workstations are able to directly address a memory which they view as global. This is illustrated in Figure 2.1.

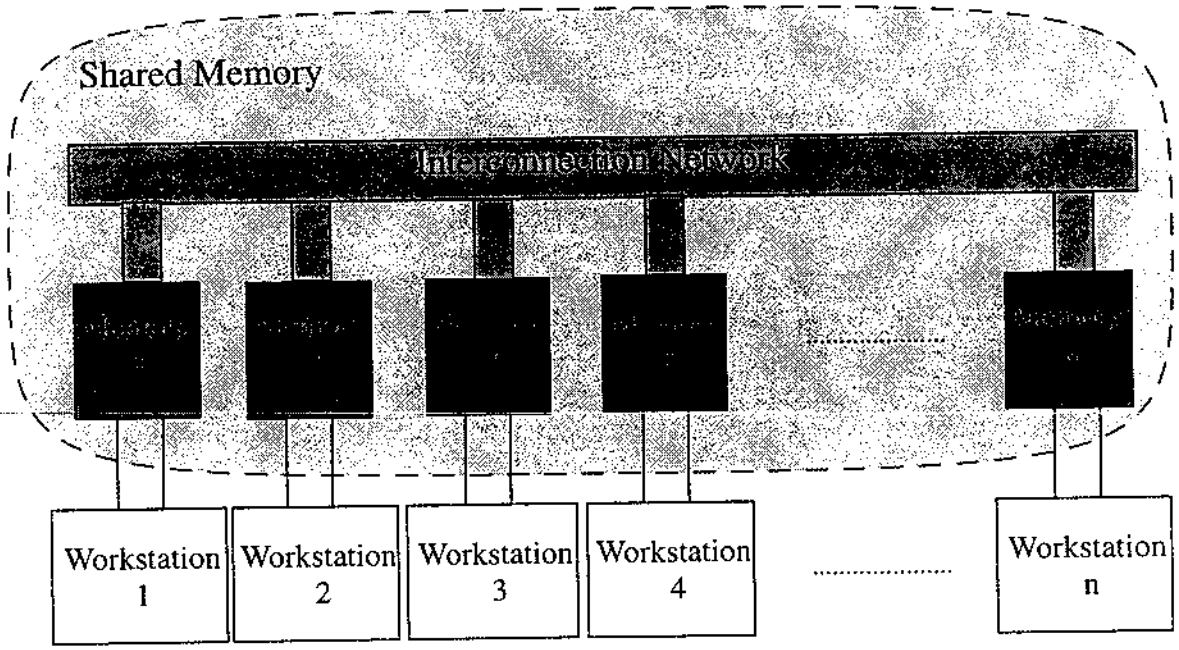

Figure 2.1 Distributed Shared Memory

The advantages of DSM [Coulouris et al. 93], [Nitzberg and Lo 94], [Levelt et al. 92], [Tanenbaum 95] are that it:

- Increases the ease of programming by sparing programmers the concerns of message passing;
- Allows the use of algorithms and software written for shared memory multiprocessors on distributed systems;
- Distributed machine scales to much larger numbers than multiprocessors resulting from the absence of hardware bottlenecks; and
- Enables the use of low cost of distributed memory machines.

Furthermore, one advantage not mentioned in the literature is that concurrent

software written for uniprocessors can be used directly on a COW. The most important advantage of DSM is that it allows the programmer to use the shared memory programming model instead of the message passing model to write their programs. These two models will be discussed and compared in the next section.

2.2.1 Shared Memory vs Message Passing Programming Models

The two basic models for parallel programming, the shared memory model and the message passing model are discussed and compared in [Carter et al. 95b] and [Lu et al. 95]. The shared memory model is traditionally thought of as easier to program than the message passing model.

The shared memory model assumes that there is a single shared memory, thus any operation performed on the memory is immediately visible to all other processes. Processes do not need to communicate explicitly with one another. They rather communicate implicitly through shared variables using ordinary read and write operations.

The distributed memory model, on the other hand, assumes that there is no shared memory and all interprocess communication must take place through message passing. The message passing paradigm requires that application programmers partition data and explicitly control the movement of that data between processes. Programmers must know which data to send, the location to which they are sending the data and the timing of the data send. Thus, using message passing is difficult particularly if the application being written uses complex data structures. Furthermore, in [Lu et al. 97], a paper in which the authors quantify the difference between DSM using TreadMarks and message passing using PVM, they state that based on their study of some applications, ".....for programs with complicated communications patterns,....., it takes more effort to write a correct and efficient message passing program". Message passing is not only tedious for the programmer but also causes difficult to solve synchronisation problems such as deadlock, transparency of memory accesses and consistency [Keane et al. 95]. This additional burden on programmers can distract them from the more important task of algorithm development.

2.2.2 Factors Affecting DSM

The main motivation behind the initial implementation of DSM was to emulate the well understood and convenient shared memory programming paradigm. However, although researchers viewed DSM as a promising tool for parallel applications, it was felt that its ultimate success would depend upon the efficiency of its implementation [Coulouris et al. 93].

In order to improve the efficiency of DSM, much work has been done to reduce the access time for non-local memory accesses to as close as possible to that for local memory accesses. Since any operation on memory which is not local involves some form of remote interprocess communication (IPC) and remote IPC is the major overhead of DSM, the reduction of these remote communications should improve the efficiency of the system. The underlying method of implementation of DSM is through the use of message passing and remote procedure calls because the shared memory must be moved or copied, or messages must be sent to a centralised memory server in order to access non-local memory. One method of reducing the number of remote communications is through the weakening of the consistency model used for the shared memory. Thus, much of the early research effort has been put into consistency models and their implementations [Iftode and Singh 97].

Because consistency models are an important issue in DSM, many of the issues and research goals of DSM are similar to those of multiprocessor caches and networked file systems [Tam et al. 90]. The shared memory is, in some developments, replicated over the network in a form of caching which is often used to improve performance in DSM systems. However, the maintenance of the consistency of this replicated memory in a transparent and efficient way is an important research objective because computer network latency is typically much larger than that of a shared bus [Nitzberg and Lo 94], [Tam et al. 90].

The research issues related to DSM fall into four broad areas which will be discussed in the next section.

2.3 DSM Research Issues

The following issues have been identified in the current literature as the major

issues in research into DSM:

- Consistency;
- Granularity;
- Data Location and Access; and
- Synchronisation.

2.3.1 Consistency

Consistency is an important issue in the design of DSM systems because the majority of messages sent during the execution of an application using DSM are consistency related. Communication is the largest overhead in DSM systems. It is possible, through the use of weaker consistency models, to dramatically reduce the number of consistency related messages being sent between workstations [Tanenbaum 95], [Nitzberg and Lo 94]. Consequently, much of the research effort has been concentrated in this area in an attempt to improve the efficiency of DSM systems [Iftode and Singh 97].

Since the memory regions are replicated across the system some mechanism must be in place to maintain the consistency of the replicated memory. Consistency models determine the order in which the shared memory updates of one process will be observed by another process. The strongest form of consistency is strict/sequential consistency which requires the serialisation of all data accesses, i.e. both read and write accesses must occur in strict order. Since this implies that all data accesses are performed in sequential order its use is precluded in parallel applications. The replication of data and removal of the strict consistency requirement allows reads of the same data to be executed in parallel. However, any write operations performed in parallel would result in inconsistent data. Thus, for more relaxed consistency models some mechanism is required to ensure the serialisation of write operations and that any subsequent reads or writes of the same memory location access the updated data. This mechanism is the coherence protocol used in the DSM system.

No distinction is made in much of the literature between the terms coherence and consistency except [Nitzberg and Lo 94] where coherence is used as a general term for the semantics of memory operations and consistency for a specific type of

coherence. In this thesis the term coherence protocol will be used to refer to the mechanism used to make the memory consistent while the term consistency will be used for specific consistency models which determine the timing of the execution of these mechanisms.

There are two coherence protocols used to serialise write accesses in parallel processes:

- *write-invalidate* — many copies of read only data are allowed but only one writable copy. All other data is invalidated before a write access can proceed.
- *write-update* — when data is written all other copies of the data are updated before any further accesses to the data are allowed.

When a write access is performed upon replicated data the copies must either be updated or invalidated.

The majority of multiprocessor systems use invalidation based protocols since the overhead of updating all copies can be high and can affect the efficiency of the system [Nitzberg and Lo 94]. In the *write-invalidate* coherence protocol each workstation has a local shared memory region which may contain writable, readable and invalid (offsite) pages. An attempt by a process to access a page for which it does not have a copy results in a page fault which will be handled by the DSM software. The DSM software will then retrieve the missing page. If the access that caused the page fault was a write access then all other copies of the page must be invalidated. Much has been written about the inefficiency of the *write-invalidate* protocol, largely because of false sharing [Hellwagner 90], [Coulouris et al. 93], [Tanenbaum 95], [Nitzberg and Lo 94]. False sharing occurs when processes on different workstations write to different memory addresses on the same page. Before the write access may proceed all other copies of the page must be invalidated. Thus, false sharing can cause the page to "thrash" across the network from one workstation to another.

In the *write-update* coherence protocol each process has a local shared memory region which contains both writable and readable pages. The processes on different workstations may read from and write to the pages, however, it is the responsibility of

the DSM software to ensure that all the shared memory regions in the system are consistent. This can be done by generating a message containing the changes made to a page and distributing it to all workstations containing the same shared memory region at the time that the write is performed. DSM systems using weak or relaxed consistency models and the *write-update* protocol commonly delay the distribution of updates until they are required by another workstation.

The consistency model chosen determines the time at which the replicated data should be invalidated/updated [Keleher 95]. Furthermore, the consistency model chosen can either incorporate synchronisation or not. The consistency models can also be classified based upon the access category for the memory. Thus, Mosberger [Mosberger 94] divided consistency models into *uniform* and *hybrid* models. Uniform models are those which do not distinguish between the memory access categories. Hybrid models make a distinction between the ordering constraints they apply depending upon the category of the access. Thus, in hybrid models, memory is made consistent at synchronisation points and the order of operations performed upon memory between these synchronisation points is not important. When using a hybrid model, the memory system will give the appearance of being sequentially consistent as long as the system adheres to the synchronisation model.

The range of consistency models, taking into consideration the Mosberger division, is shown in Figure 2.2.

The uniform consistency models, often called strict models, are:

- *atomic*, where any read operation returns the value stored by the most recent write operation;
- *sequential*, where invalidation is performed before the write is executed;
- *causal*, where causally related events should be seen immediately; and
- *processor consistency*, where updates are pipelined so that they are seen on all processors in the same order.

The hybrid consistency models, often called weak models, are:

- *weak*, where all previous writes of non-shared variables are made consistent with any access to a synchronisation variable;

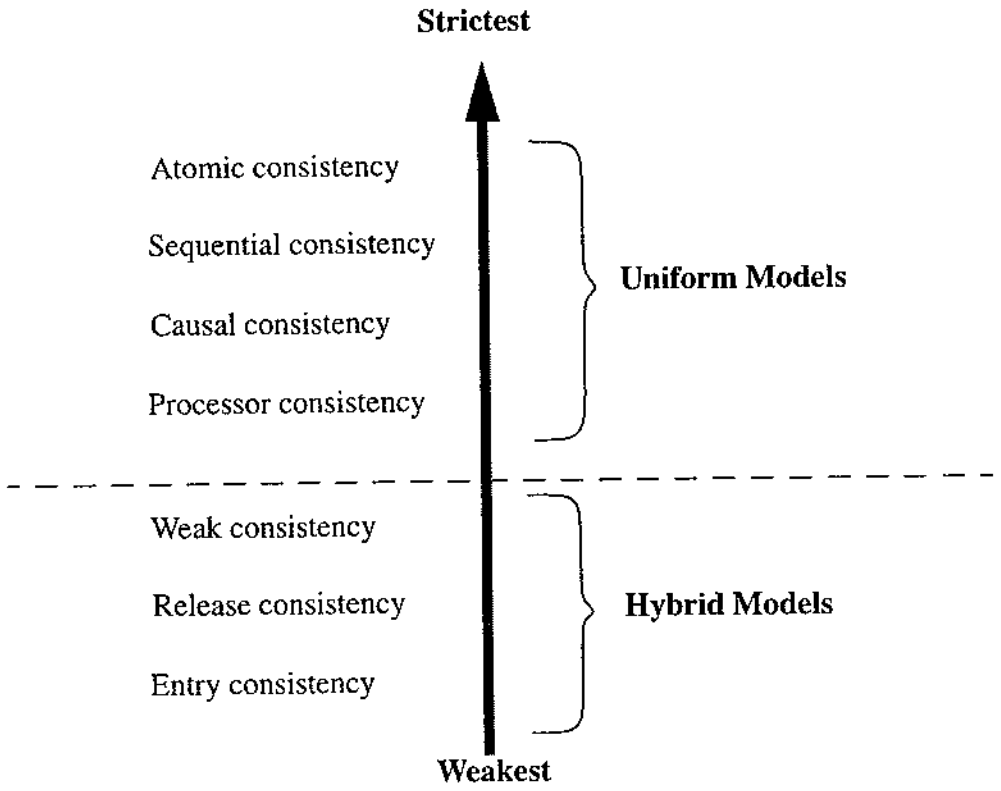

Figure 2.2 Range of Consistency Models

- *release*, where all previous writes of non-shared variables are made consistent when a lock on a synchronisation variable is released; and
- *entry consistency*, where all previous writes of non-shared variables are made consistent when a lock on a synchronisation variable is obtained.

In an attempt to make DSM more efficient various hybrids of the consistency models which incorporate synchronisation have been developed. These increase the efficiency of the system by relying on the fact that the data need only be consistent when it is required to be read thus avoiding unnecessary updates. Some of these hybrid models are [Tanenbaum 95]:

- *eager consistency models* where a data block may be made invalid immediately after a read; and
- *lazy consistency models* where any data block is made invalid when a read operation is requested subsequent to a write operation e.g. lazy release consistency.

2.3.2 Granularity

The granularity of the data is an important issue related to the size of the unit of sharing [Nitzberg and Lo 94]. The granularity can be one of the following:

- fixed sized blocks of data (IVY [Lamport 79], TreadMarks [Keleher 96]);
- variable sized blocks of data (Midway [Bershad et al. 93], Munin [Carter et al. 95b]); or
- user defined structures (Orca [Bal et al, 92], Linda [Carriero and Gelernter 86], Clouds [Mohindra 93]).

When deciding on the granularity there are three aspects to be considered; minimising the communication overhead; false sharing; and the existing units of memory managed by the operating system. Since in all message passing systems there is a large, constant communication latency, the time taken to send a large message is very little more than to send a small one. The temptation is to use large grained memory units. However, with large units of memory false sharing becomes an issue. The third aspect to be considered is the unit of memory managed by the operating system. The DSM system integrates well with the operating system's virtual memory management if the granularity of sharing in the DSM system is the same or a multiple of the memory units managed by the operating system.

2.3.3 Data Location and Access

The accessing of non-local data can affect the efficiency and scalability of the system. An inefficient data location algorithm can slow the execution down and a centralised approach can result in a bottleneck [Nitzberg and Lo 94].

In an invalidation-based DSM system an attempt, by a user process, to access non-local data should result in an access fault. The DSM system should provide some mechanism to locate the missing data. Extensive work on page allocation strategies was carried out by Li and Hudak [Li and Hudak 89]. These strategies can be broadly divided into centralised and distributed strategies.

In the centralised strategy some form of centralised manager is maintained which controls all access to the data. The manager holds all information regarding replicated copies of all pages and their owners as the pages do not have fixed owners.

With respect to write-invalidate, the owner of a page is the processor which most recently had write access to the page. A centralised strategy can cause a bottleneck since all data accesses must be preceded by a message to the single centralised manager [Nitzberg and Lo 94]. This can affect the performance and scalability of the distributed system.

Distributed strategies can further be sub-divided into those where the page management is either:

- *fixed,* where the page manager remains fixed throughout execution; the page managers are not located on a single node but are distributed throughout the system; or
- *dynamic,* where the page management moves with ownership changes.

The fixed approach can result in a bottleneck since all data access requests have to pass through the manager, but makes accessing the owner easier since the requesting process knows where to locate the page manager which in turn has ownership information.

When a dynamic management strategy is used a mechanism is required to locate the current owner of the page. The ownership information can be centralised or distributed. A centralised method means that the information regarding the ownership of each page is kept on a single machine and that machine is informed of any changes of ownership. This too can result in a bottleneck. If the ownership information is distributed, there is a problem of maintaining the consistency of this data. In some implementations broadcast messages are used to update ownership information on each machine. This broadcast message is either sent at each ownership change, which can be expensive, or periodically. If periodical updating is used each machine maintains a *probowner* table which contains the last known owner of the data; this gives a hint of the location of the current owner [Li and Hudak 89]. This is referred to as the Dynamic Distributed Management strategy.

2.3.4 Synchronisation

Before discussing synchronisation it is important to clarify the terms used in

this thesis. For this purpose a synchronisation taxonomy is proposed here shown in Figure 2.3. Following this taxonomy, synchronisation which controls access to a single

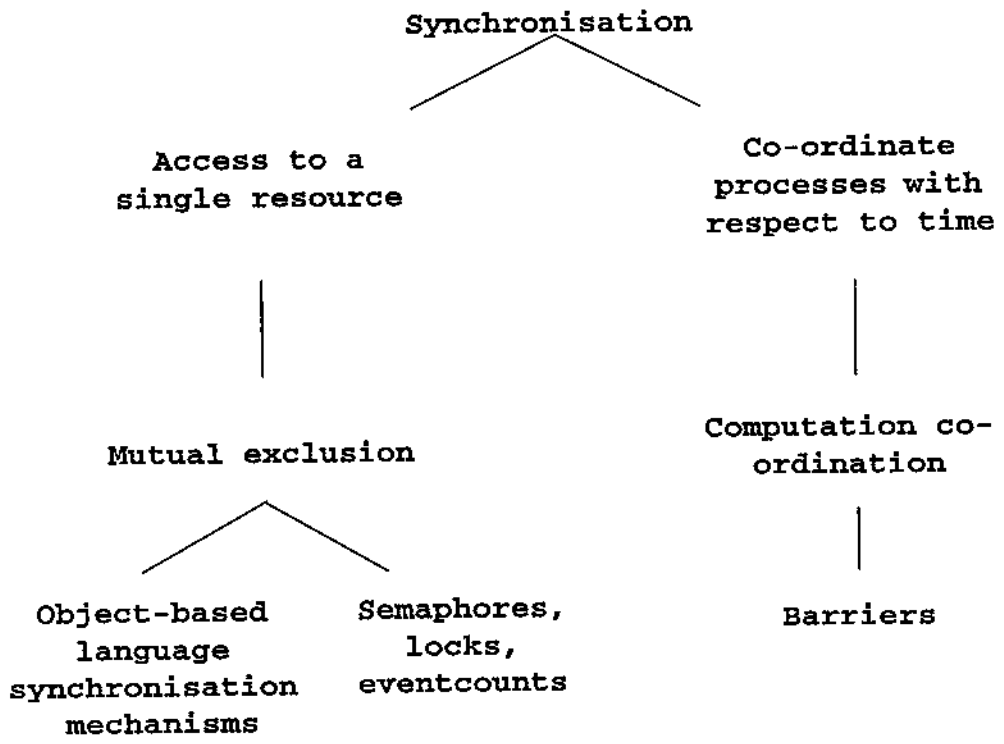

Figure 2.3 Taxonomy of synchronisation

resource and synchronisation to co-ordinate the execution of more than one process. The former is achieved by enforcing mutual exclusion while the latter by the co-ordination of parallel computations such that no process passes a particular point in the code until all processes have reached that point.

Mutual exclusion can either be made the responsibility of the application programmer, using explicit synchronisation primitives, or it can be made the responsibility of the system developer, as in object based implementations, with synchronisation being implicit at application level. The latter involves the use of a specialised language.

The implementation of these two forms of synchronisation can either be achieved using two separate or a single mechanism. In a non-object-based system the shared memory in a DSM system must be accessed within a critical section as in any shared memory system. Synchronisation for mutual exclusion is achieved through the

use of semaphores, eventcounts, monitors or locks.

When synchronisation is being implemented in a DSM system two issues must be considered, firstly the type of synchronisation mechanism to be used and secondly the implementation of that mechanism. A survey of synchronisation mechanisms for shared memory multiprocessors is given in [Dinning 89]. This survey indicates that semaphores are the most common form of synchronisation in shared memory systems. These mechanisms are commonly based on atomic access to shared variables. Since, in DSM there is no physically shared memory, a decision must be made on a manner to implement the semaphores.

In DSM systems the semaphores can be implemented as shared variables in the distributed shared memory itself or in the portion of the memory which is not shared [Ramachandran and Singhal 95]. If the former implementation is used the semaphores will be treated like any shared variables and will be kept consistent using the DSM mechanisms, i.e. write-update or invalidate mechanisms. In [Ramachandran and Singhal 95] this is referred to as the integrated method, where the synchronisation mechanism is integrated into the DSM systems's mechanisms. This may lead to unnecessary thrashing of the pages on which these variables are kept in invalidation-based systems. The authors of this paper suggest a mechanism to avoid this by placing the semaphores on a separate *semaphore page*. Alternatively, a non-integrated mechanism may be used where the synchronisation mechanism is quite separate from the DSM systems's mechanisms, i.e. the semaphores may be placed in memory which is not shared, e.g. as part of the system's memory. This type of implementation requires that the DSM system have read and write access to system memory.

The control of the synchronisation itself can either be centralised or distributed. According to a centralised approach the control of all synchronisation variables is maintained by a single synchronisation manager. This approach can cause a bottleneck. Distributed management approaches are along the same lines as those for distributed page management, where the node which last held the synchronisation variable is nominated as the manager.

An important aspect of synchronisation in DSM systems, if the DSM system is

to genuinely allow application programmers to use the shared memory programming model, is that the synchronisation should not differ from that used in tightly coupled systems and concurrent systems executing on a uniprocessor. In other words, the synchronisation mechanism should be one that is familiar to programmers and the implementation should be transparent at application level.

Computation co-ordination is another element of synchronisation. One implementation of computation co-ordination is barriers [Ramachandran 95]. Barriers provide a mechanism whereby multiple processes can synchronise at certain points during program execution. Processes synchronise at barriers by stopping their execution until all parallel processes executing the same application have reached the barrier after which all processes will continue execution.

The implementation of barriers can be through [Grunwald and Vajracharya 94]:

- A centralised algorithm with a single workstation being nominated the barrier manager and all processes communicating with the manager when they reach their barrier. This approach is simple but not highly scalable;
- Software combining trees where the processes are divided into groups. When a process reaches a barrier it communicates with the rest of the group. The process whose message arrives last at the rendezvous point sends a message to the next group;
- Dissemination algorithm where the sequence of send and receives are statically determined for the network. Thus each process knows when and where to send its barrier message. When the last process in the network receives its message this indicates that all processes have reached their barrier. Each host then signals its partner process in reverse allowing it to continue execution.

2.4 Current Implementations of DSM

In this section the current implementations of DSM are discussed and their advantages and disadvantaged emphasised in order to identify those features which should and should not be incorporated into the DSM system proposed in Chapter 3. Particular attention is paid to the programming environment provided by the DSM

itself and the environment in which the DSM has been implemented.

2.4.1 IVY

IVY was the first implementation of a DSM system [Li 88], [Li and Hudak 89]. It was prototyped on Apollo workstations which were connected by a token-ring network. The operating system on top of which it was implemented was a modified Aegis operating system. IVY used the sequential consistency model using the write-invalidate coherence protocol. The unit of granularity used was a 1K page.

IVY not only showed that software DSM was viable but also introduced the dynamic distributed page management strategy. In this strategy ownership of pages follows the write permission for that page. The owner is then responsible for handling requests for the page and keeping the copyset or list of readable copies of the page until a write request for that page arrives. The owner then invalidates all copies of the page including its own and sends a copy of the page to the workstation on which the request originated.

Unfortunately because of the strict consistency model, the performance was not good. In addition, false sharing was quickly identified as a substantial problem for page-based DSM systems. In extreme cases of false sharing, neither of the user processes ever gets a chance to perform an operation upon the page as it is requested and removed before the user process has performed its operation. False sharing could be reduced by careful placement of the shared data structures but this placed an unacceptable load on the application programmer.

The synchronisation mechanism used in IVY was eventcounts. The decision to use them was driven by the fact that they had already been implemented in the Aegis operating system. The eventcounts were stored in local memory of the processors and synchronisation communication was achieved using remote procedure calls. In a later implementation the eventcounts were placed on shared pages which were migrated to the processors on which they were required. The latter mechanism is said to be simpler and more efficient than the RPC implementation. Ramachandran [Ramachandran 95] doubts that better performance would be achieved if multiple eventcounts which shared a page were accessed by multiple remote processes.

The experiences gained from IVY were:

- DSM is feasible and can result in significant speedups over sequential execution;
- Invalidation based implementations with large page sizes experience false sharing;
- Dynamic distributed page location strategy is very effective.

2.4.2 Mirage

Mirage was developed at UCLA [Fleisch and Popek 89]. It was developed as a prototype as an additional service of a modified version of the Locus operating system running on top of 3 VAX 11/750s. Mirage was a write-invalidate based implementation which used a 512-byte page as the unit of granularity, and exploited the sequential consistency model.

The major improvement made by Mirage over IVY was that it reduced the effects of false sharing by locking a page on a processor for a certain period of time. This meant that the requesting process was able to make some progress in its execution because it could perform all or part of its access to that page before another process' request for the page was handled. Unfortunately, invalidation messages to replicated pages were sent sequentially rather than broadcast or multicast and there was no facility to queue invalidations received when the page was locked. These invalidations had to be resent [Protic et al. 97].

Mirage+ [Fleisch et al. 94] was later developed to run on a network of 12 personal computers with a modified version of AIX 1.2.

The synchronisation mechanism used in Mirage was semaphores, implemented in Locus. The semaphores were grouped into sets, managed by a set manager.

The experiences gained from Mirage were:

- False sharing can be reduced by locking a page on a particular processor for a period of time;
- Invalidation messages to replicated pages should be broadcast or multicast rather than being sent sequentially; and

- Invalidations received when the page is locked should be queued rather than being resent.

2.4.3 Clouds

Clouds [Chen and Dasgupta 91] was developed at Georgia Institute of Technology. It was implemented on a network of SUN 3 workstations connected through a 10Mbps ethernet at the low, address level in Ra, the nucleus of the Clouds operating system. In [Mohindra 93] a study was performed on DSM integrated into the virtual memory system of Ra's virtual memory management.

Clouds was the first object-oriented development to use a DSM approach to support object relocation [Protic et al. 97]. The resources in the system were all viewed as objects which were composed of program-defined segments. The consistency model used in Clouds was sequential consistency with the program-defined segments as the unit of granularity. Programmers were able to lock and unlock segments on particular workstations. However, this required additional knowledge from the application programmer.

A Distributed Shared Memory Controller (DSMC) at each node in the network maintained any segments created at that node. This DSMC maintained consistency through the use of a coherence protocol which was lock-based and managed to combine the transportation of data with synchronisation.

To support synchronisation independent of the RPC mechanism semaphores were supported in the kernel of the operating system. These semaphores were grouped together into semaphore segments on the processor on which they were created. Requests were handled on this processor and requests for a semaphore which was already held by another process were queued on this processor.

The experiences gained from Clouds were:

- The performance of applications using this DSM is poor possibly because the operating system was object oriented;

2.4.4 Munin

Munin was developed at Rice University [Carter 93]. It was implemented on a

network of SUN 3/60 workstations on top of the V System. The granularity of the shared memory was a variable sized object.

Munin was the first DSM to use a weak consistency model. In fact, it introduced the concept of multiple consistency protocols. Munin used nine shared data-types with their own consistency model. The consistency model used was object specific and depended upon the access pattern of the object. The use of weak models improved the performance of applications but, unfortunately, the labelling of the objects was left to the application programmer. This meant that the programmer was required to have a thorough knowledge of not only the data access patterns for the application but also the DSM implementation itself.

Synchronisation was in the form of locks which were implemented as one of the shared data types. In addition, barriers and condition variables were provided. The locks used the Dynamic Distributed Management strategy for the lock manager.

Carter debated whether to use a distributed barrier mechanism resembling the barriers designed for scalable multiprocessor systems, but decided to use a centralised scheme based on the size of the prototype implementation. Furthermore, the barriers used in their test applications were used to mutually synchronise all user threads in the program. Hence, a centralised scheme required fewer messages than the distributed algorithm.

The experiences gained from Munin were:

- Weak consistency models are feasible;
- Weak consistency models reduce the execution time for applications over sequential consistency models;
- The labelling of programming objects by the application programmer to use the multiple consistency protocols is tedious for the programmer and may lead to reduced performance if the programmer's labelling of the objects is less than optimal.

2.4.5 Midway

Midway was developed at Carnegie Mellon University [Bershad et al. 91],

[Bershad et al. 93]. It was implemented on a cluster of MIPS R3000-based DEC stations on top of Mach 3.0.

Midway introduced a new consistency model — entry consistency. In addition, it allowed the use of multiple consistency models. When the entry consistency model is used the memory is made consistent upon entry to a critical section. This meant that shared data had to be associated explicitly with the synchronisation variables which were to protect the critical sections in which it were to be accessed, i.e. variables had to be declared as *shared* and at runtime these variables are associated with a particular synchronisation variable. The speedups shown by Midway were very promising but unfortunately the complexity added to the application programmer's task proved prohibitive for complex problems.

The experiences gained from Midway were:

- Entry consistency is feasible;
- Entry consistency improves performance over sequential consistency;
- The labelling of variables and the explicit association of each shared variable with a synchronisation object is tedious for the application programmer and may lead to reduced performance if the programmer labels the variables incorrectly.

2.4.6 TreadMarks

TreadMarks was developed at Rice University [Keleher 95], [Keleher 96]. It was implemented at user level on top of Ultrix V4.3. The hardware platform used for the implementation and testing of TreadMarks was a network of eight DECstation-5000/240's connected by a 100-Mbps switch-based ATM LAN. Each machine had a Fore ATM interface connected to a Fore ATM switch.

The granularity of the sharing used in TreadMarks was a 4K page. TreadMarks was the first DSM system to use Lazy Release Consistency. This consistency model was a hybrid of the Release Consistency Model in which the propagation of changes made to the shared memory were delayed until a process exits the critical section. In the lazy hybrid the propagation was delayed further until another process attempted to enter a critical section. At this stage the memory was updated with changes made to it

during the last critical section. Only the memory of the process attempting to enter the critical section was updated. Unfortunately, this meant that a large amount of memory was required to store the unapplied updates and that at some stage a garbage collection function had to run to apply the updates to all copies of the shared memory in order to free this memory.

Synchronisation in TreadMarks was carried out by locks and barriers. The lock managers were fixed and were assigned at initialisation time. Because lazy release consistency meant that not all versions of the shared memory were updated when the release operation was carried out, all locks had to be timestamped. The timestamps were then used to determine which updates needed to be forwarded upon the release operation.

Barriers were implemented using a centralised barrier manager. The manager collected messages indicating that a process had arrived at the barrier and distributed messages to the processes allowing them to continue execution after the barrier. A round robin ordering was used to statically assign managers to barriers.

The experiences gained from TreadMarks were:

- Lazy release consistency is feasible;
- The algorithm for lazy release consistency is highly complex;
- Delaying updates requires a large amount of memory;
- Delaying updates requires a careful management and a time consuming garbage collection function.

2.5 Conclusion

In this chapter the issues in DSM research and current DSM implementations have been examined in order to highlight the research areas that have been neglected. The lessons learned from this general review of DSM are that it is a promising area of research because: it relieves programmers of the task of writing message passing code; allows the use of new and reuse of existing shared memory code; and allows shared memory code to be run on scalable and cheap COWs. The major areas that need to be considered in DSM system design are the consistency model to be used, the granularity of the shared data, the method of locating shared data and the synchronisation

mechanism to be used and its implementation.

The existing DSM systems discussed in this chapter were IVY, Mirage, Clouds, Munin, Midway and TreadMarks.

The IVY implementation showed the general feasibility of software DSM and that a level of improvement is possible over sequential execution time. The implementation of the dynamic distributed page location algorithm showed that this algorithm is effective in locating objects in a distributed system. The most significant problem of the IVY implementation was false sharing.

Mirage's contribution to DSM knowledge was that false sharing can be reduced by locking a page or data block on a particular processor for a period of time. Furthermore, the distribution of invalidation messages should use a broadcast or multicast facility and that queuing of request messages rather than resending them should be used whenever the request cannot be handled immediately.

Mohindra's work on implementing DSM in Clouds' kernel [Mohindra 93] demonstrated that it is feasible to implement a DSM system at the very low address level within the kernel of an operating system. Clouds is an object-oriented system. The poor performance of this DSM system indicates that the choice operating system into which DSM is integrated operating is a very important factor in DSM design.

Munin extended the knowledge of DSM by demonstrating that weak consistency models are feasible and reduce application's execution time compared to sequential consistency models. However, the use of multiple consistency protocols requires the labelling of programming objects by the application programmer. Thus, Munin was the first DSM system which required significant additional knowledge and input from the application programmer. Although the effect of this is not easily measured it is thought by researchers [Iftode and Singh 97], [Ramachandran 95] to place a tedious burden on the application programmer.

Midway's contribution to DSM research was that the use of the weakest consistency model, entry consistency, was feasible and indeed improved the performance of applications. However, the extensive labelling of variables and the explicit association of each shared variable with a synchronisation object requires

unacceptable additional input from the application programmer.

TreadMarks introduced lazy release consistency, a highly complex algorithm which requires a large amount of memory. Updates to memory are delayed until a process attempts to enter a critical section and then only that process' memory is updated. The shared memory must be periodically brought up to date by a garbage collection function in order to release the memory.

Overall the major lessons learned from this review is that the desirable attributes of a DSM system are that it should:

- be integrated into an operating system;
- use a weak consistency model;
- require little input from application programmers additional to that required for shared memory programming;
- enable the reuse of shared memory programs;
- be without specialised hardware requirements, i.e. is able to be run on off the shelf hardware.

In addition, the work discussed in [Ramachandran 95] and [Dinning 89] gives a good insight into the synchronisation requirements for DSM. It is clear that programmers are most comfortable with semaphore-type synchronisation and that this type of synchronisation mechanism should be handled with the use of dynamic distributed management to decrease the chance of causing a bottleneck.

Chapter 3 Synthesis of a DSM System Integrated into a Distributed Operating System

3.1 Introduction

The advantages of using DSM over using message passing are that it allows:

- the programmer to employ the familiar and easy to use shared memory programming model,
- the direct use of existing shared memory software, and
- both of the above to be achieved in a cluster of workstations, COW, (distributed system) which scales to larger numbers of machines than tightly coupled systems, and demonstrates an excellent ratio of performance to cost.

The analysis in the previous chapter of the DSM systems developed by other researchers has highlighted that ease of use for application programmers is sacrificed for the sake of performance. The DSM system proposed in this chapter places more emphasis on ease of programming and use, while still not sacrificing performance, than developing an efficient, but sometimes difficult to use, DSM system which the average programmer simply will not adopt as their execution environment.

The goal of this chapter is to report on the study into the synthesis of a DSM system within a microkernel and client-server based distributed operating system. The DSM system proposed uses both strict and weak consistency models, with a write-invalidate and write-update based approach for consistency maintenance. This proposed DSM system provides programmers with a complete programming environment in which they are easily able to develop and run their code or indeed run existing shared memory code. At the same time users are able to gain the performance advantages of running applications in parallel.

In order to demonstrate that this research aim and chapter goal have been achieved this chapter is structured as follows. First of all, appropriate DSM system design requirements have been elaborated. Secondly, two models of parallelism to be supported by the DSM system are presented. Thirdly, the reasons for and design of the placement of the proposed DSM system within a distributed operating system and in particular in the memory management server have been reported. Fourthly, a high level description of the functioning of the DSM system within a client-server and microkernel based distributed operating system has been presented. This is followed by the synthesis of a new DSM system, which concentrates on an event-driven set of co-operating and distributed entities, and a detailed description of the events and reactions to these events that make up the operation of the DSM system. Fifthly, the detailed design of the main modules and activities in the form of the pseudocode of the primitives used in the proposed DSM system are presented.

3.2 DSM Design Requirements

As stated in [Hellwagner 90] many of the current designs of write-invalidation based DSM have taken the basic IVY implementation and attempted to improve one aspect of the design. These improvements come at times at the expense of the others. These aspects of DSM have been studied and as a result weaker consistency models than those implemented using the write-invalidate coherence protocol have been introduced in an attempt to improve the performance of DSM systems. These aspects have been discussed in the previous chapter. However, ease of programming and even the attempt to have a general, application independent system have largely been neglected.

The aim of this research, as stated in Chapter 1, is to synthesise and develop a DSM as an integral part of a distributed operating system in order to provide application programmers with a convenient environment in which to develop and execute parallel applications, and which does this in a transparent and efficient manner. In order to meet this aim and on the basis of the study of the advantages and disadvantages identified for other DSM developments a cohesive design for a DSM system is presented.

As a first step to describing the proposed DSM system the following design requirements have been identified as being desirable for a DSM system:

- *Ease of Programming.* The DSM should provide programmers with an easy to use environment in which they are comfortable writing their shared memory code. The programmer should not be forced to go beyond the concepts of sequential shared memory-based programming, supported by such constructs as semaphores, locks or barriers, with which they are well familiar. This will allow easy development of new and porting of existing sequential programs to the proposed execution environment.
- *Ease of Use.* The DSM should be easy to use; it should provide a convenient environment in which to run parallel, shared memory programs. Programmers should be able to start a single parent process and the system should not only create all parallel child processes but should also place them transparently on workstations in the system.
- *Transparency.* The users should be unaware that the memory they are using is not physically shared and they should not be expected to have any DSM-related input to the program other than barriers which are easy to use and understand.
- *Efficiency.* The access time of non-local memory should be as close to the access time of local memory as possible. This is also related to transparency in that the programmer should see no discernable difference between local and non-local memory accesses.
- *Scalability.* One of the benefits of DSM systems mentioned in much of the literature [Nitzberg and Lo 94], [Tanenbaum 95] is that they scale better than many tightly-coupled shared-memory multiprocessors. Scalability is limited by physical bottlenecks. Thus, there should be no potential bottlenecks to limit the scalability of the system.

Application programmers are still required to carry out data decomposition for problems which use the SPMD computational model. However, since the performance of this operation is also a requirement for parallel programming for physically shared memory environments this is not regarded as DSM specific input.

In the following sections the proposed DSM system is described and how this design meets these very demanding requirements described in this section.

3.3 Parallelism to be supported by the proposed DSM System

The basic design requirement of the proposed DSM system is "ease of programming". By "ease of programming" it is understood that the programmer should use programming techniques used in sequential programming; the only simple extension is the co-ordination of the parallel processes.

The two coarse grained parallelism models, the single program multiple data (SPMD) and multiple program multiple data (MPMD) computational models, have been discussed in Chapter 1. These models use a higher level of abstraction than the SISD, SIMD, MISD and MIMD models which use the instruction stream as their level of granularity. In contrast, the SPMD and MPMD models use a sequence of actions such as a program as their level of granularity [Goscinski 97]. The SPMD computational model exploits data parallelism while the MPMD computational model exploits functional parallelism.

Data parallel processing is the manipulation of "chunks" of data in parallel; thus an important element of this type of processing is domain decomposition or data partitioning — the act of dividing the data set into portions to be manipulated independently of one another by the same program. Thus, in applications using SPMD model the data is partitioned into "chunks" before execution commences. Data partitioning is the responsibility of the application programmer and is not part of the function of the DSM system. The data is partitioned into a number of equal sized "chunks" according to the number of processes being used to execute the application. When using DSM each user process has the full data set in its shared memory. Thus, each user process uses its process number to identify the "chunk" of data assigned to it for computation.

Functional parallelism involves the division of the computation task itself into two or more computation sub-tasks which execute in parallel also on "chunks" of data. The group of processes exhibiting functional parallelism require mutually exclusive

access to shared data but otherwise execute independently of one another toward a common goal.

The code given in Figures 3.1 and 3.2 shows the programming aspects of DSM

```
main(){
      initialisation;
      Data_i = ith partition of Data, i = 1..n
      co-ordinate processes;    /*Synchronise the*/
                                 /*start of execution*/
      program(Data_i);
      {
            ......              /*program statements*/
      }
      co-ordinate processes;    /*Synchronise the*/
                                /* end of execution*/
}
```

Figure 3.1 Generic Code using SPMD Model

based parallel execution using the SPMD and MPMD computation models, respectively.

When either model is used all processes must synchronise (co-ordinate) at certain points during the execution of the application in particular at the commencement and conclusion of the parallel code. When all processes have reached a particular co-ordination point they may continue execution. This form of co-ordination is used to ensure that all processes have reached a particular point in the execution before any of the processes may continue beyond that point.

The co-ordination point at the start of execution is required because, as in the case of a tightly coupled system where the memory is physically shared, none of the processes should begin computation until the shared memory initialisation has been completed. Likewise, the final barrier ensures that the processes exit only after all processes have completed the execution of the application.

```
main(){
      initialisation;
      co-ordinate processes;    /*Synchronise the*/
                                /* start of execution*/
      program_i(Data)           /*Where program_i is a */
                                /*functional unit of the*/
                                /*whole program*/
      {
      ......                    /*program_i statements*/
      critical section entry;   /*Entry to the critical*/
                                /*section*/
      ......                    /*program_i statements*/
      critical section exit;    /*Exit from the critical*/
                                /*section*/
      ......                    /*program_i statements*/
      critical section entry;   /*Entry to the critical*/
                                /*section*/
      ......                    /*program_i statements*/
      critical section exit;    /*Exit from the critical*/
                                /*section*/
      ......                    /*program_i statements*/

      }
      co-ordinate processes;    /*Synchronise the*/
                                /*end of execution*/
}
```

Figure 3.2 Generic Code using MPMD Model

In the MPMD model the processes share program variables. In order to maintain the consistency of these variables access to them must be controlled. Thus, additional synchronisation is required to that for SPMD computations, i.e. some mechanism to ensure processes have mutually exclusive access to shared variables. Commonly, shared accesses take place within a critical section where only one process may be in the critical section at any time. Strictly, the SPMD computational model

does not require the use of critical sections as there is no data sharing. However, many of the test applications which use the SPMD model such as the solution for Partial Differential Equations (PDEs) have a small amount of sharing of boundary values, access to which must be synchronised.

An analysis of the existing DSM systems carried out in Chapter 2 shows that many of them exploit the shared memory in their implementations of synchronisation. Largely their synchronisation takes the form of locks which are implemented as variables within the shared memory region. In an implementation using the write-invalidate model the placement of the lock variables in the shared memory must be very carefully considered. If the lock variables are placed on the same page as program variables they can become involved in false sharing problems. Furthermore, an analysis of programming languages and programs developed for physical memory shared system [Dinning 89], [Ramachandran and Singhal 95] shows that the most commonly used method of defining critical sections is through the use of semaphores. For these reasons the use of semaphore-type synchronisation is proposed with the semaphore variable being a data structure in the memory server.

During the development of the proposed DSM system the practicality of using a distributed barrier mechanism like those used for multiprocessor systems was considered. It was, however, decided that a simple centralised scheme would be sufficient. Like the barriers used in Munin [Carter 93], all of the barriers used in the test applications synchronise all of the user processes in the application, rather than a subset of the processes. Hence, the centralised scheme requires a smaller number of messages than the more complex distributed algorithm.

3.4 The Location of the DSM System

The environment for a DSM system, i.e. the operating system that manages resources and services of a COW, is an important element influencing the design of a DSM system. The aim of this section is to propose an architecture which will support a DSM system to meet the design requirements listed in Section 3.2. For this purpose, firstly, the architectures in which the DSM system sits on top of an operating system are discussed. This is followed by the design of the architecture best suited to the

proposed DSM system. Having defined the best environment for the DSM system, the location of the DSM within that environment is proposed and elaborated.

3.4.1 Current DSM Architectures

An examination of the environments in which current DSM systems are executed has shown that they run as a separate piece of software or as a set of library functions on top of an existing network operating system. This architecture is shown in Figure 3.3.This solution suffers, as stated in Chapter 2, from a lack of transparency,

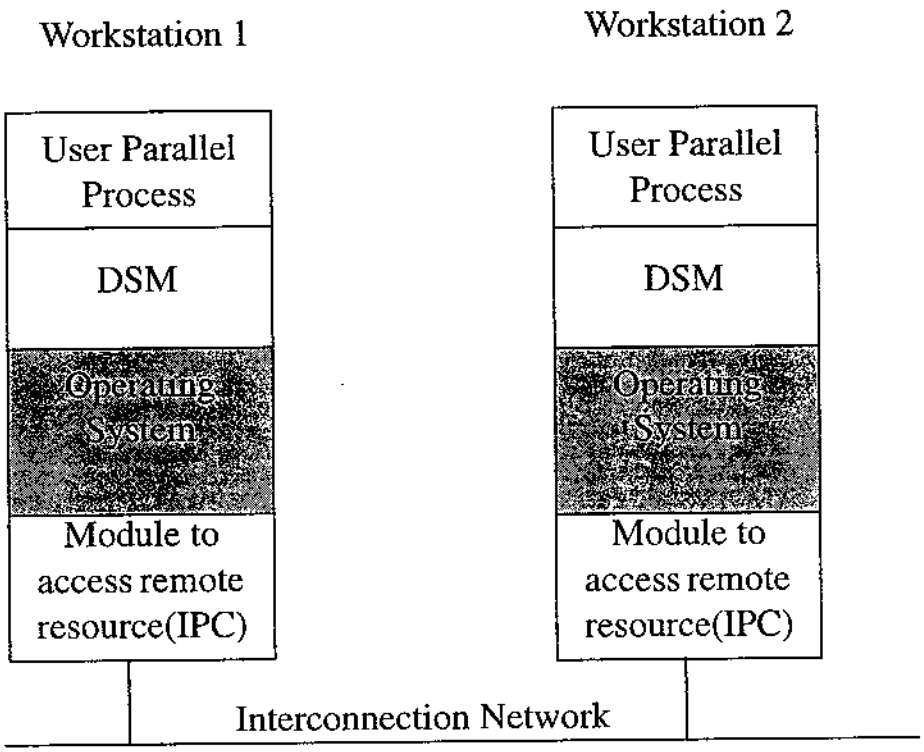

Figure 3.3 Network operating system based architecture

efficiency and ease of use because the DSM cannot use low level operating system functions and must often be explicitly invoked by the user.

Furthermore, some DSM systems have allowed the design of the operating system on top of which they are to run to influence the design decisions made for the DSM system itself. IVY [Li and Hudak 89] uses eventcounts for synchronisation, not necessarily because the system developers thought it was the best form of synchronisation, but because eventcounts already existed in the Aegis operating system on top of which IVY was implemented. From this example it can be seen that there is a

risk that DSM design decisions may be compromised if the DSM is placed on top of an established existing operating system which is fixed rather than being developed in a flexible operating system environment. For these reasons it is proposed that the DSM system should be integrated into a modifiable operating system which can be adapted to accommodate the DSM design decisions and not vice versa.

3.4.2 Distributed Operating System Based DSM Architecture

The deficiencies identified in other DSM systems and discussed in Chapter 2 are that these systems lack:

— Ease of use for application programmers;

— Ease of programming for application programmers; and

— Transparency.

Placing the DSM system in a flexible operating system environment will not only help to overcome some of the deficiencies identified in other DSM developments but it will help to achieve some DSM requirements (presented in Section 3.2) by exploiting the generic objectives of operating systems. These requirements are:

— Ease of Programming;

— Ease of Use;

— Transparency;

— Efficiency; and

— Scalability.

Operating system objectives include abstracting away from the hardware and providing programmer convenience through a sophisticated programming environment and efficient resource management. Since programmer convenience can be achieved, to some extent, by providing an environment in which programmers can develop and run their code with ease, an operating system integrated DSM should help to meet the requirements for the DSM system. Thus, it is proposed that the DSM system, which provides a programming environment which supports parallel processing on a COW, should be integrated into the operating system itself rather than being run as a separate piece of software on top of the operating system. The proposed architecture is shown in

Figure 3.4.

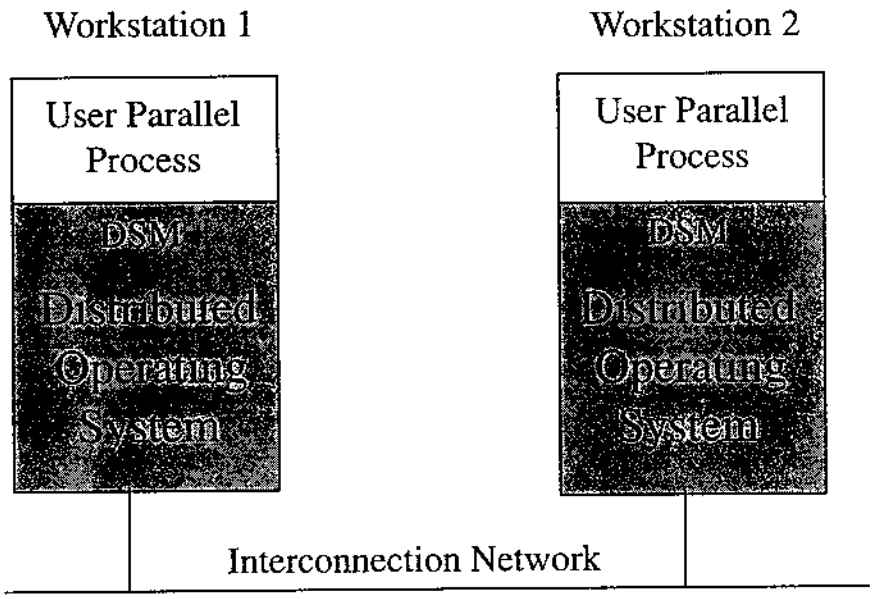

Figure 3.4 Architecture of distributed operating system which includes DSM system

Since distributed operating systems are designed to transparently manage COWs [Goscinski 91], it follows that the most logical choice of an operating system to contain the proposed DSM system is a distributed operating system. Distributed operating systems can be broadly classified into monolithic kernel and microkernel based distributed operating systems.

Monolithic kernel based operating system architectures result in systems that provide the complete operating system functionality within a large kernel. A large monolithic kernel provides services more efficiently than a microkernel. However, monolithic kernel based operating systems have lost favour because they offer very little flexibility and configurability [Goscinski 91], [Tanenbaum 95]. Hence, this type of operating system would be unable to provide the flexibility required to meet the DSM systems' design requirements. Consequently, a monolithic kernel based distributed operating system was rejected as the environment for the proposed DSM system. Current development trends are towards the more flexible microkernel and client-server based distributed operating systems.

In a microkernel based architecture the microkernel supplies the minimum functionality and services, with the remainder of the services being provided by a set of servers. This enables designers to separate the "policy" and "mechanism" and thus design a flexible, modular operating system. The ability to separate policy and mechanism and the modularity of the client-server model are clear advantages of this model, while the microkernel provides a compact, well designed and adaptable footing upon which the whole operating system can be built. Consequently, the decision was made to place the proposed DSM in a microkernel and client-server based distributed operating system since it would provide the required flexibility and is the distributed operating system of the future. Furthermore, integrating the DSM system into such an operating system should help the DSM system to meet the transparency and efficiency requirements proposed in Section 3.2. The next decision, which is discussed in the following section, is the placement of the DSM system within the operating system.

3.4.3 Memory Management Based DSM System

The main resources of any computer system, including COWs, are processes and processors, memory and messages. In a microkernel and client-server based operating system the system resources are managed by a set of servers such as a process server, memory server, and interprocess communication server, respectively. Shared memory can itself be viewed as a resource which requires management. The options for placing the DSM system within the operating system are either to build it as a separate server or incorporate to it within one of the existing servers. The first option of placing the DSM system in the operating system as a separate server was rejected because there would have been a conflict with two servers (the memory server and the DSM server) both managing the same object type, i.e. memory. Synchronised access to the memory in order to maintain its consistency would become a serious issue. Since DSM is essentially a memory management function the memory server is the server into which the DSM system must be integrated. Therefore the decision was made to integrate the DSM into the functionality of the memory server of the distributed operating system, as shown in Figure 3.5.

In order to support memory sharing in a COW, which employs message passing to allow processes to communicate, the DSM system must be supported by the

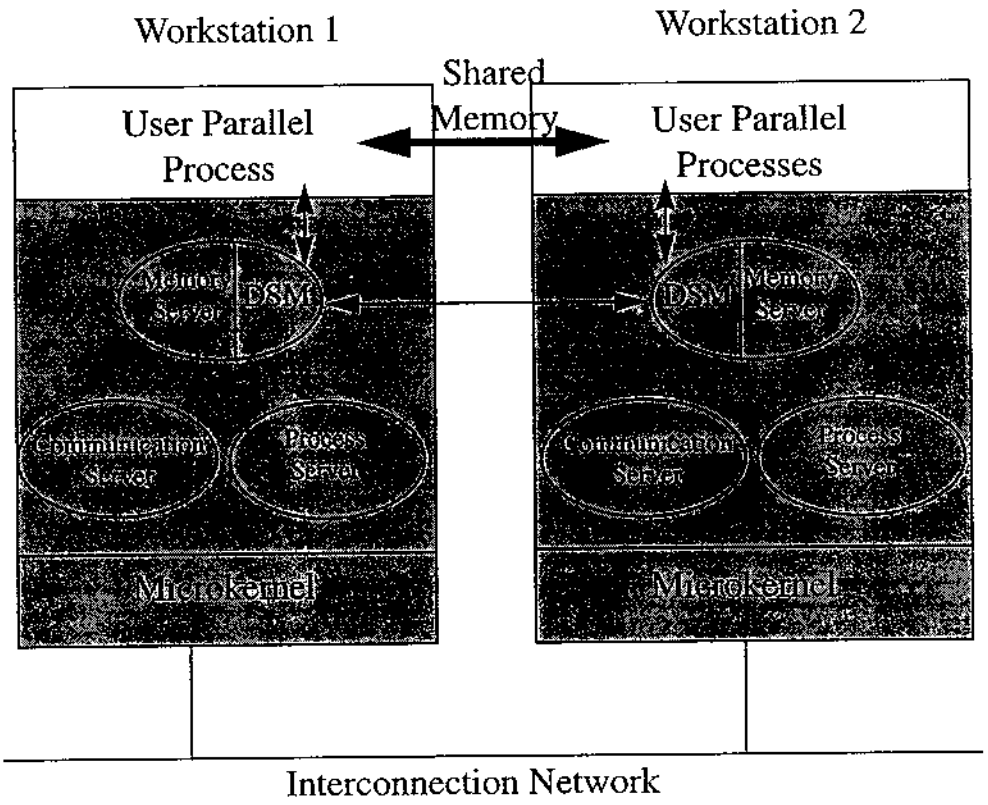

Figure 3.5 Architecture of client-server model and microkernel based distributed operating system incorporating DSM system

interprocess communication server. However, the support provided to the DSM system by this server is invisible to programmers of the user processes. Furthermore, because the DSM based user parallel processes must be properly managed, including their creation, synchronisation when sharing a memory object, and co-ordination of their execution, the process server must be involved in DSM system activities.

The granularity of the shared memory object is an important issue in the design of a DSM system. The proposed DSM system is placed within the memory server of a microkernel and client-server based distributed operating system. The granularity of a memory unit of modern operating systems is usually a page. Therefore, it follows that the most appropriate object of sharing for the DSM system is the operating system's memory unit, and consequently a page.

A single memory unit, page, rather than multiple ones are to be used as the unit

of granularity in the proposed system. The reason is that, although multiple memory units would reduce the effect of the communication overhead, it would increase the incidence of false sharing and thus reduce the efficiency of the DSM system.

The placement of the DSM system in the memory manager of a client-server and microkernel based distributed operating system should help the DSM system to achieve several design requirements. Firstly, because the DSM system is integrated into the memory management's functionality the programmer is able to use the shared memory as though it were physically shared, hence, ease of programming and transparency are achieved. Secondly, because the DSM system is in the operating system itself and is able to use the low level operating system functions efficiency is achieved.

3.5 Services provided by the DSM system

In this section the design of the proposed DSM system and the services provided by this system are presented and discussed. The discussion relates to the requirements for this proposed DSM system and to the problems of other developments as identified in Chapter 2. This section is organised in the following way. The first subsection contains a description of a general design of the proposed DSM system, including the automatic initialisation process, synchronisation method and operation of write-invalidate and -update based DSM systems. Since this system is event driven, in the next subsection, a precise protocol of the synthesised DSM system and a description of the events that occur and the reaction to these events which drive the system are given. This is followed by a detailed description of the working of the system and pseudocode of the primitives (reactions to the identified events) which are invoked by the events.

An application using the DSM system presented in this thesis is made up of a group of user processes executing on a group of remote workstations. The first or parent process initialises the DSM memory and starts the execution of a set of child processes through a library call to the operating system. In response to the library call the operating system places a child process on each of a group of workstations identified as being idle or lightly loaded. The operating system then creates and

initialises an identical DSM memory to that of the parent process and initialises the synchronisation primitives for all processes (children and parent). The application then commences execution. The consistency model used by the DSM system for this application is either sequential or release, (the programmer is able to choose), the default consistency model is release consistency.

The sequential consistency model is implemented using the write-invalidate protocol, where all copies of a page are invalidated whenever that page is altered. The ownership of a page moves with write access to that page. A Dynamic Distributed Page Ownership strategy is used with a chain of probable owners used to locate the current owner of a page. The release consistency model is implemented using the write-update coherence protocol. All pages written to while a process is in a critical section are duplicated. When the process exits the critical section the changes are found and propagated to all workstations running DSM where they are used to update the local copy of the memory.

Synchronisation is in the form of semaphores and barriers. Semaphore ownership uses a similar Dynamic Distributed strategy to that used for page ownership. Semaphore ownership moves to the workstation on which the process resides that last entered the critical region. The semaphore location algorithm like the page location algorithm uses a chain of probable owners to locate the required semaphore's owner.

3.5.1 Operating system integrated DSM

In this subsection an overview is given of how the proposed DSM system integrated into a client-server and microkernel based distributed operating system works. This presentation forms the background for the detailed design of the DSM system.

3.5.1.1 Starting the DSM system: General Design

A parallel application using distributed shared memory is made up of two or more processes running on separate workstations. The problem is to initialise these processes on all workstations. Furthermore, these processes may share memory. This implies a need to create sharable objects and associate them with these parallel processes. The latter operation is influenced by the consistency model being used. The

synchronisation approach selected, as stated earlier, is semaphore-based mutual exclusion. Thus, semaphores must be initialised in order to correctly access sharable objects. Parallel processes must be coordinated to achieve correct results. Barriers which are used as the coordination mechanism, must also be initialised.

These basic initialisation operations could be carried out either manually by the programmer or automatically by an initialisation system. The latter has been advocated as a solution which satisfies the ease of programming and ease of use design requirements of this proposed DSM system.

In order to satisfy these requirements, the DSM system should be initialised in the following way. When an application using DSM starts to execute, the first process initialises the DSM system. This first process, known as the parent, requests initialisation with a single primitive which initially requests the memory server to allocate a block of memory for the globally shared memory. An investigation of existing DSM implementations [Carter et al. 95b], [Bershad et al. 93], [Keleher 95], presented in Chapter 2, has shown that weaker consistency models improve the performance of applications using DSM compared with strict models. However, this issue has not been studied within one single environment where both sequential and release consistency are implemented within the operating system. Thus, it is important to test this within a single environment and on a range of test applications. For this reason, both sequential and release consistency models are employed in the proposed DSM system. Programmers are able to nominate one of two consistency models to be used in the DSM system. However, the default is release consistency. When the memory block has been created the memory server attaches it (the memory block) to the parent process.

The next operation carried out by the initialisation primitive is a request for the operating system to initialise parallel execution, i.e. start the remote (child) processes. The operating system uses the services of the scheduler and process server to begin parallel execution. The scheduler uses system load information to find the number of idle or lightly loaded workstations which are available to take part in the parallel execution of the application. The operating system then creates a single child process on each of the remote workstations nominated by the scheduler as eligible to take part

in the parallel execution.

When all of the child processes have been created successfully the processes all block waiting for the unique identifier for the memory block from the parent. The child process executes a primitive which attempts to attach this memory block to itself. The attach request causes control to be passed to the local memory server which detects that the memory block does not exist locally. The local memory server then requests the information regarding the memory block from the parent process's memory server such as size, location and consistency model. When this information is received an identical memory block is created locally and attached to the child process. After completion of this operation a message is sent to the parent's memory server. The parent and child processes' memory servers then initialise the semaphores and barriers and return the address of the shared memory to the parent and child processes.

When the memory servers return the memory address of the shared memory to the parent and child processes they also return the number of processes involved in the parallel execution. This number is then used by the application programmer for data decomposition if the computation being carried out uses the SPMD computational model. The DSM processes then synchronise at the end of this initialisation phase before starting execution of the application.

The method of creating physically shared memory is the same as that for distributed memory sharing. If for any reason more than one process needs to be created on a single workstation the mechanism is the same except the processes share the same physical memory. When a child process using the release consistency model is started on a workstation on which a DSM process performing the same application already exists, the new child process requests the memory server to attach the shared memory block to itself. The memory server searches for and finds the memory block which exists locally and attaches the child to the existing memory block. Semaphores and barriers function in the same way as they do when the processes are remote from one another. Any updates carried out on the shared memory are immediately visible to both processes.

3.5.1.2 DSM System's Mechanisms: General Design

Certain mechanisms need to be in place to implement the policies for the proposed DSM system. In this section these mechanisms are discussed.

When a user process using a write-invalidate based DSM system attempts to access a page which is not on the local workstation a page fault results. Page ownership migrates with write permission for each page. Each workstation maintains a field containing the last known owner (often called probable owner) of the page. This gives a hint of the location of the current owner. The trail of probable owners will eventually reach the current owner.

The DSM system uses two synchronisation mechanisms: semaphores for mutual exclusion; and barriers for the coordination of the parallel computation. The location of the semaphore ownership is through a chain of probable owners. A request for entry to a critical section is forwarded along a probable owner chain until the current owner is located. If a process is already in the critical section, the request is queued in the request queue to be handled later. If no process is in the critical section, the request is acknowledged immediately and the requesting memory server will become the new owner of the semaphore. If, when a process exits the critical section, there are requests in the request queue, the request at the head of the queue is acknowledged and the remainder of the queue is forwarded to the requesting memory server along with the acknowledgement.

In order to co-ordinate parallel DSM processes each process will halt at a barrier until all participating processes have reached the same barrier in their own code. Barrier management is centralised. When a user process reaches a barrier it will block. Control is then passed to the memory server which checks whether release consistency is being used and if so updates must be distributed. When all updates have been distributed and replies received the memory server sends a message to the central barrier manager. The central barrier replies when it has received a message from all processes indicating that they have reached the barrier. When a user process receives the latter reply it will unblock and continue execution.

3.5.1.3 DSM System Functioning: General Design

The general operation of the DSM system will be introduced in this section showing how the system functions when the two consistency models are used.

When the parallel user processes start execution they perform read and write operations on the shared memory. The DSM system is responsible for ensuring that whenever processes access shared memory they have a consistent view of that memory. This is achieved by making certain that all copies of the shared memory are updated after one of the processes has altered (written to) its copy of the memory. In the proposed DSM system, there are two mechanisms for making the memory coherent; the write-invalidate and write-update coherence protocols. The timing of this updating process is determined by the consistency model.

As stated earlier, both sequential and release consistency models are employed in the proposed DSM. The write-invalidate protocol is used to implement the sequential consistency. In the write-invalidate protocol a single read/writable copy or multiple read-only copies of any page can exist at one time.

When the write-invalidate coherence protocol is being used an attempted read access on a page which is offsite, i.e. does not exist in the memory of the local workstation, causes a page fault which in turn causes the memory server to send a message to the current owner of the page along the probable owner chain. When the current owner of the page is located a copy of the requested page being sent to the requesting workstation and the requesting workstation is added to the copyset. The copyset is a list (maintained by the memory server which owns the page), of the locations of all readable copies of that page. When received at the requesting workstation the page is mapped in to the shared memory of the requesting process and the permissions are set to read-only.

An attempted write access to an offsite page also results in a page fault; the memory server sends a request message along the probable owner chain to locate the current owner of the page. This current owner sends a message to all members of the copyset requesting them to invalidate their copy of the page. The members of the copyset reply to the owner when they have invalidated the page. When all of the replies

have been received the owner sends the page and any requests queued for handling to the requesting workstation before invalidating the local copy of the page. When the page reaches the requesting workstation the page is mapped into the shared memory and the page is given read-write permissions. The page is then locked for a period so that the user process has exclusive access to the page to enable it to complete its write operation. Any requests for the page while it is locked are queued to be handled later.

An attempt to write to a read-only page also results in a page fault. A request for write permission is sent to the memory server which owns the page along the chain of probable owners. This request is accompanied by the version number for the page. As in the case of the write access fault on an offsite page, the owner sends a request to all members of the copyset to invalidate their local version of the page. When all of the replies have been received from the members of the copyset confirming that they have invalidated their pages, the owner compares the version number of the local page with the version number in the request message. If the version number which accompanied the request is out of date, i.e. smaller than the current version, the page itself is sent to the requesting workstation along with the message granting write access. When this message reaches the requesting workstation the page, if it accompanied the message, is mapped in to the memory and the page is given read-write permissions and locked for its time-slice.

In this proposed design the write-update protocol is used to implement release consistency. The write-update protocol means that at any time there may be multiple read-write copies of any page in the system. Changes made to a page are placed in a message and sent to all workstations, the pages on these workstations are then updated. Release consistency means that the propagation of updates is delayed until the process exits from a critical section or departs from a barrier.

Initially each page in the shared memory is made read-only. Thus any attempt to write to a page in the shared memory causes a write fault. If the memory is using release consistency, the memory server makes a copy of the page (or twin) and the status of the page is changed to read-write before the write can proceed. Any further write accesses to that page will not cause a write fault because the protections of the page have been changed to read-write. When a user process attempts to exit a critical

section all of the pages that have been twinned are compared word by word with their twins to identify any updates made to the shared memory. These updates are then placed in a message by the memory server and sent to all workstations which are sharing the memory. When this update message is received by another workstation the memory server on that workstation applies the updates to the shared memory and replies to the originating workstation. When the originating workstation has received all replies the user process may exit the critical section.

The functioning of the release consistency model is the same when the memory is physically shared. In other words, it is transparent to the user if more than one processes physically share memory. An attempt to perform a write access on a page in the shared memory region on the workstation which has physically shared memory results in a page fault because the page has read-only protection. As before, the memory server twins the page and changes the protection for that page to read-write so that further accesses will not trigger a fault. The write operation can then proceed.

When the child process on the workstation which has physically shared memory attempts to exit the critical section the memory server generates and sends the updates to the all remote workstations which are sharing the memory. Since the local child is physically sharing the memory with the server, the updates are already visible to this process.

3.5.2 Events and Reactions of the DSM System

The results of the general design presented in Section 3.5.1 specify the functioning and basic mechanisms which form a background for the synthesis of the DSM system integrated into a client-server and microkernel based distributed operating system. Following Section 3.5.1, the proposed DSM system has been synthesised as an event driven set of co-operating and distributed modules. The new DSM system is presented in two parts. The first contains the event-based protocols of a DSM parent and DSM child. The second shows a description of the events that occur and the reaction to these events which drives the system.

3.5.2.1 DSM system event protocols

The DSM system is an event driven set of modules described by two basic

protocols: the DSM parent protocol and DSM child protocol. They are shown in Figures 3.6 and 3.7, respectively. These two protocols differ only in the initialisation events.

3.5.2.2 Events and reactions of the DSM system

The following events are required to provide all the necessary services to the distributed shared memory parallel processes:

Create_shared_memory — A request is sent by the child or parent to the memory server to create a memory block. The format of the primitive is as follows:

- *dsm_create(memory_size, consistency_model, process_id)* → (*memory_id*) — This primitive is performed implicitly during the execution of the *start_dsm*() in the parent's initialisation phase and from the *memory_attach*() primitive which is executed from *dsm_parstart*() in the child's initialisation phase. It creates a memory of *memory_size* size, labels it as *consistency_model* type and attaches it to *process_id* process. The primitive returns the identifier of the memory that has been created (*memory_id*).

Attach_memory — A request is sent by a child process to the local memory server to attach a memory block to itself. The format of the primitive is as follows:

- *memory_attach(process_id, memory_id)* → (*memory_id*) — This primitive attempts to attach the *memory_id* memory block to the *process_id* process. If the memory is not local and it is a DSM shared memory a message is sent to the parent's memory server requesting the dimensions and position of the memory. When received an identical memory block is created and the *memory_id* for this memory is returned. If the memory to which the process is attempting is local, i.e. there is already a DSM process on this workstation, the attach proceeds and the memory is physically shared.

Automatic_initialise_parent — Whenever a parent process has created its

```
DSM_Parent
     Automatic_initialise_parent:
          start_dsm(memory_id, memory_size,
                         opt_num_procs, child_name,
                         process_id, num_sems,
                         num_barriers);
Event Types {Create_shared_memory, Attach_memory,
        Enter_critical_section, Exit_critical_section,
        Reach_barrier, Receive_barrier_message,
        Page_fault, Read_on_non_resident_page,
        Write_on_non_resident_page,
        Reply_to_page_fault_request,
        Receipt_of_page_write_request, Update_memory,
        Receipt_of_diff_message}
repeat{
     case of (event)
          Create_shared_memory:
               dsm_create(memory_size,
                         consistency_model, process_id);
          Attach_memory:
               memory_attach(process_id,memory_id);
          Enter_critical_section:
               wait(sem);
          Exit_critical_section:
               signal(sem);
          Reach_barrier:
               dsm_barrier_req(barrier);
          Receive_barrier_message
               dsm_barrier(barrier);
          Page_fault:
               exception_handler(fault_address,
                         type_of_fault);
          Read_on_non_resident_page:
               page_copy_req(fault_address);
          Write_on_non_resident_page:
               page_copy_write_req(fault_address);
          Reply_to_page_fault_request
               page_copy_write(fault_address);
          Receipt_of_page_write_request:
               page_invalidate_req(fault_address);
          Update_memory
               distribute_diffs(memory_id);
          Receipt_of_diff_message
               update_pages(memory_id);
      }until forever;
```

Figure 3.6 DSM system parent protocol

```
DSM_Child
     Initialise_child:
          dsm_parstart (process_id, num_sems,
                        num_barriers);
Event Types (Create_shared_memory, Attach_memory,
        Enter_critical_section, Exit_critical_section,
        Reach_barrier, Receive_barrier_message,
        Page_fault, Read_on_non_resident_page,
        Write_on_non_resident_page,
        Reply_to_page_fault_request,
        Receipt_of_page_write_request, Update_memory,
        Receipt_of_diff_message}
repeat{
     case of (event)
          Create_shared_memory:
               dsm_create(memory_size,
                          consistency_model, process_id);
          Attach_memory:
               memory_attach(process_id,memory_id);
          Enter_critical_section:
               wait(sem);
          Exit_critical_section:
               signal(sem);
          Reach_barrier:
               dsm_barrier_req(barrier);
          Receive_barrier_message
               dsm_barrier(barrier);
          Page_fault:
               exception_handler(fault_address,
                          type_of_fault);
          Read_on_non_resident_page:
               page_copy_req(fault_address);
          Write_on_non_resident_page:
               page_copy_write_req(fault_address);
          Reply_to_page_fault_request
               page_copy_write(fault_address);
          Receipt_of_page_write_request:
               page_invalidate_req(fault_address);
          Update_memory
               distribute_diffs(memory_id);
          Receipt_of_diff_message
               update_pages(memory_id);
     }until forever;
```

Figure 3.7 DSM system child protocol

shared memory it is ready to start parallel execution with this primitive. The format of the primitive is as follows:

- *start_dsm(memory_id, memory_size, opt_num_procs, child_name, process_id, num_sems, num_barriers)* → *(memory_address)* — The primitive involves *process_ncreate*() which uses information from the system to find the system load and starts processes from the *child_name* executable code on disk. When these have been created it initialises *num_sems* semaphores and *num_barriers* barriers. The primitive returns the address of the shared memory block to the parent process.

Initialise_child — Whenever a child process starts execution this primitive is the first one executed. The format of the primitive is as follows:

- *dsm_parstart (process_id, num_sems, num_barriers)* → *(memory_address)* — This primitive causes the process to block until the information about the memory (*memory_id* and *num_procs*) arrives from the parent's workstation. It then involves *memory_attach*() to attempt to attach to the memory. If the memory is not local *memory_attach*() creates a shared memory block identical to the parent's shared memory block. A message is then sent to the parent's memory server. The primitive then initialises *num_sems* semaphores and *num_barriers* barriers. The primitive returns the address of the shared memory block to the child process.

Enter_critical_section — This event occurs whenever either a parent or child process requests entry to a critical section. The format of the primitive is as follows:

- *wait(sem)* — The memory server uses a probable owner "trail" to locate the actual owner of the semaphore and then either grants access to the critical section or queues the request to be handled later if the semaphore is already being accessed by another process.

Exit_critical_section — This event occurs whenever a parent or child process requests to exit a critical section. The format of the primitive is as follows:

- *signal(sem)* — If the write-update protocol is used, a message containing updates is generated and sent to all workstations sharing the same memory. The user process waits until all workstations have acknowledged the update message before it goes on to release the semaphore. The latter is then either changed to released status or access is granted to the head request in the request queue, if one exists.

Reach_barrier — This event occurs whenever any user process reaches a barrier in their code. A message is sent to the memory server which triggers the execution of this primitive. The format of the primitive is as follows:

- *dsm_barrier_req(barrier)* — This primitive causes the memory server to check whether the write-update protocol is being used, if so a message containing updates is generated and sent to all workstations sharing the same memory. When all acknowledgements are received it sends a message to the barrier manager.

Receive_barrier_message — This event occurs whenever a barrier message, sent during the execution of the *dsm_barrier_req*() primitive, is received by the central barrier manager. The format of the primitive is as follows:

- *dsm_barrier(barrier)* — This primitive is executed by the central barrier manager which is in the memory server for the parent process. The *barrier_cnt* variable is incremented. If *barrier_cnt* equals the number of participating processes, i.e., all participating processes have reached their barrier, a multicast message is sent allowing all of the participating processes to unblock and continue execution.

Page_fault — This event occurs whenever a page fault happens which triggers a hardware trap, this results in a message being sent to the memory server on the workstation on which the trap occurred. The format of the primitive is as follows:

- *exception_handler(fault_address, type_of_fault)* — This primitive is executed by the memory server. If the write-invalidate protocol is being

used either the *page_copy_req*() or the *page_copy_write_req*() primitive is executed. If the write-update protocol is being used the page is twinned and the page permissions are changed to read_write.

Read_on_non_resident_page — This event occurs whenever a message is received from a memory server. The message is sent from either the *exception_handler*() primitive or forwarded along the probable owner chain from another *page_copy_req*() primitive. The format of the primitive is as follows:

- *page_copy_req(fault_address)* — This primitive is executed by the memory server when the write-invalidate protocol is used. It is invoked initially from the *exception_handler*(). The request may be forwarded to the probable owner when the primitive invocation would be triggered by the receipt of the request message. Otherwise a copy of the page is sent to the requesting workstation and the local copy is made read_only.

Write_on_non_resident_page — This event occurs whenever a message is received from a remote memory server requesting a writable copy of a page. The message is sent from either the *exception_handler*() primitive or forwarded along the probable owner chain from another *page_copy_write_req*() primitive. The format of the primitive is as follows:

- *page_copy_write_req(fault_address)* — This primitive is executed by the memory server when the write-invalidate protocol is used. It is called initially from the *exception_handler*(). The request may be forwarded to the probable owner when the primitive invocation would be triggered by the receipt of the request message. Otherwise a copy of the page is sent to the requesting workstation and the local copy is made invalid after all other copies in the system have been invalidated.

Reply_to_page_fault_request — This event occurs whenever a message is received in reply to a request for a page or write permission to a page. The message is sent either from a *page_copy_write_req*() or *page_copy_req*() primitive being executed. The message contains either a copy of a page or write

permission to an existing page. The format of the primitive is as follows:

- *page_copy_write(fault_address)* — If a page was requested it is mapped in to the local memory and the page given the appropriate access, read-write or read-only.

Receipt_of_page_write_request — This event occurs whenever a write request has been received and the local memory server is the owner of the requested page. The write request will have been sent during the execution of a *page_copy_write_req*() primitive. The format of the primitive is as follows:

- *page_invalidate_req(fault_address)* — This primitive is executed by the memory server on the workstation of the owner of the requested page. This primitive causes a message to be sent to all workstations which have copies of the requested page indicating that the page on which *fault_address* exists should be invalidated.

Receipt_of_page_invalidate_replies — This event occurs whenever a message is received from a remote workstation indicating the remote copy of the page has been invalidated. The message that triggers this event is sent during the execution of a *page_invalidate_req*() primitive. The format of the primitive is as follows:

- *page_invalidate(fault_address)* — When all of the pages in the copyset have been invalidated a copy of the page is sent to the requesting workstation if required and the local version of the page on which *fault_address* exists is invalidated.

Update_memory — This event occurs whenever a DSM process attempts to exit from a critical section and at the start of a *dsm_barrier_req*(), if the release consistency model is being used. The format of the primitive is as follows:

- *distribute_diffs(memory_id)* — This primitive is executed by the memory server. It causes all twinned pages in the memory block labelled *memory_id* to be compared with their originals to identify changes which are bundled into a message which is sent to all workstation sharing the memory.

Receipt_of_diff_message — This event occurs whenever an update message is received and the release consistency model is being used. The format of the primitive is as follows:

- *update_pages(memory_id)* — This primitive is executed by the memory server. It causes the updates contained in the diff message to be applied to the memory block labelled *memory_id* on the workstation at which the message has been received.

3.5.3 Initialisation

The following sections contain the presentation of the functioning of the DSM system using individual primitives. The proposed DSM system, which is made up of a set of event-driven co-operating and distributed entities, is described in detail with the primitives for the main modules and activities shown in the form of pseudocode.

In an ideal DSM system, programmers are able to write shared memory code or use previously written shared memory code and run the code on a COW with no knowledge of the DSM system running within the operating system. When an application using DSM starts to execute the first process, known as the parent, it initialises the DSM system as shown in Figure 3.8. The parent requests parallel

```
parent initialisation
     memory_address= start_dsm(memory_id,
        consistency_model, memory_size, opt_num_procs,
        child_name, process_id,num_procs, num_sems,
        num_barriers)

child initialisation
     memory_address= dsm_parstart (process_id,
        num_procs,num_sems, num_barriers)
```

Figure 3.8 Pseudocode for Parent and Child Initialisation

execution using the pseudocode shown in Figure 3.9 with the *start_dsm*() primitive. The *start_dsm*() primitive requests the memory server to allocate a block of memory

```
start_dsm(opt_num_procs, child_name, process_id,
        num_sems, num_barriers)
{
      memory_id = dsm_create (memory_size,
         consistency_model, process_id)
      num_procs = process_ncreate(child_name,
         opt_num_procs, new_process_ids)
      initialise semaphores
      initialise barriers
      return memory address of new shared memory region
}
```

Figure 3.9 Pseudocode for Parent Process' Parallel Initialisation

for the globally shared memory using a *dsm_create*() primitive. The *dsm_create*() primitive, shown in Figure 3.10, causes the memory server to create the memory block

```
dsm_create(memory_size, consistency_model, process_id)
{
      Create the memory and place identifier in memory_id
      make the memory permissions read_only
      if the consistency_model is SEQUENTIAL
           label the memory INVALIDATE
      else                                    /*Default*/
           label the memory UPDATE
      attach the memory to process (process_id)
      return the memory_id to the calling process
}
```

Figure 3.10 Pseudocode for DSM Memory Create

of *memory_size* and labels it *consistency_model* and attaches it to the parent process which has process identifier *process_id*.

The coherence protocol is nominated by the application programmer through the *consistency_model* parameter (the default is the release consistency model). This can either be RELEASE or SEQUENTIAL. Based on the latter *consistency_model*

used the memory is either labelled UPDATE or INVALIDATE, respectively, since these are the coherence protocols that will be used to implement the consistency models. The memory server then returns the identifier for the memory region (*memory_id*) to the *start_dsm*() primitive. To execute DSM processes in parallel, the operating system then creates a single child process on each of the remote workstations nominated by the scheduler as eligible to take part in the parallel execution using the *process_ncreate*() primitive. This primitive requires, as parameters, the name of the executable file for the child, *child_name*, the optimal number of processes to execute the application, *opt_num_procs*, and a field in which the identifiers for the newly created processes, *new_process_ids*, will be placed. The remote processes, known as the child processes, are created remotely from an executable image on disk. Each child process begins its initialisation by executing a *dsm_parstart*() primitive, Figure 3.11.

```
dsm_parstart (process_id, num_sems, num_barriers,
        num_procs)
{
     when memory_id and num_procs received from parent
               process
     populate num_procs parameter
     memory_id is returned by memory_attach(process_id,
               memory_id)
     send message to parent that memory has been created
     initialise semaphores
     initialise barriers
     return memory address of shared memory region
}
```

Figure 3.11 Pseudocode for Child Process' Parallel Execution

When all of the child processes have been created successfully they block waiting for the *memory_id* of the shared memory block from the parent. The child process then attempts to attach the memory block to itself using the *memory_attach*() primitive, shown in Figure 3.12. The *memory_attach*() detects that this memory block is not local and sends a message to the memory server on the parent's workstation

```
memory_attach(process_id,memory_id)
{
      If memory_id is local memory
            attach memory to process_id
      else If memory_id is DSM memory
      {
            send request to memory_id's owner for memory's
                  dimensions
            when dimensions returned
            call dsm_create(memory_size,
                  consistency_model, process_id)
      }
      return memory_id of memory region
}
```

Figure 3.12 Pseudocode for memory_attach

requesting the dimensions and position of the memory block. When this information is returned an identical memory block is created using the *dsm_create*() primitive (Figure 3.10). The *dsm_create*() primitive returns the *memory_id* identifier for the memory to the *memory_attach*() primitive which in turn returns the identifier to the *dsm_parstart*() primitive. This identifier is unique for the shared memories on each workstation. When the memory block has been created a message is sent to the parent's memory server. The child's memory server then initialises the semaphores and barriers and returns the memory address of the shared memory to the child process. The primitive also populates the *num_procs* field with the number of participating processes, which is used to decompose the data into "chunks" for SPMD parallel execution as described in Section 3.3.

Although the application programmers should be able to specify the preferred number of processes to execute the application, *opt_num_procs*, it is not appropriate that they should be involved in the final decision regarding the number of workstations on which to execute the processes in parallel. This decision should clearly be the function of the distributed operating system in a COW. Thus *num_procs* may not be the

same as *opt_num_procs*.

The memory identifier of the parent's shared memory, *memory_id*, which was sent by the parent process is used as a unique identifier for the group of parallel processes participating in the execution of this particular task. This identifier is used by the operating system to tag messages sent between the members of this group. The DSM processes then synchronise at the end of this initialisation phase before starting execution of the application.

Physical memory sharing in the proposed DSM system is designed to provide full transparency. This means that there is no difference between invoking and reacting to results as actions of the DSM system performed on remote and local workstations. Thus, if more than one DSM process is created on a single workstation the mechanism is the same except the processes share the same physical memory. When a child process using the release consistency model is started on a workstation on which a DSM process performing the same application already exists, the new child process invokes the *dsm_parstart*() primitive as normal. This primitive invokes the *memory_attach*() primitive. The memory server searches for and finds that the memory to which it is attempting to attach already exists locally and the new child process is simply attached to the existing shared memory, i.e., the memory is physically shared. Furthermore, the design is such that semaphores and barriers function in the same way as they do when the processes are located on different workstations. However, any updates carried out on the shared memory are immediately visible to both processes.

3.5.4 Data and Semaphore Location and Access

When a user process using a write-invalidate based DSM system attempts to access a page which is not on the local workstation an access fault should result. The DSM system must provide some mechanism to locate the missing data. The data location algorithm can affect the efficiency and scalability of the system, hence, it is important to use an efficient data location algorithm. By studying other implementations of DSM it has been possible to identify that page and semaphore management are best implemented using a dynamic distributed algorithm, as centralised management can result in a bottleneck and hence limit the scalability of the

system.

The Dynamic Distributed Management Algorithm [Li and Hudak 89] is used in the proposed DSM system to locate the current owner of the page. In this algorithm the page ownership migrates with write permission for that page. Each memory server maintains a field (frequently called the probable owner or *probowner*) which contains the last known owner of the data. This gives a hint of the location of the current owner. The trail of probable owners will eventually reach the current owner. The maximum number of messages to locate the owner is n - 1, where n is the number of workstations in the COW which are using DSM. In addition, an update message is sent to all workstations at specified intervals to update the probable owner on all workstations. Similarly, the proposed algorithm for location of the semaphore ownership is through a chain of probable owners, as shown in Figure 3.13.

```
page_copy_req(fault_address)
{
     if the page_status is REMOTE
          forward to probowner
     else if the page_status is LOCKED
          add_request to request queue
     else if the page_status is OWNER
     {
          make local page read_only
          add requesting memory server to copyset
          send page_copy message
     }
}
```

Figure 3.13 Pseudocode for Page Copy Fault

3.5.5 Synchronisation and Computation Co-ordination

Two synchronisation mechanisms have been selected (in Section 3.3) for the proposed DSM system: semaphores and barriers. Because one of the design requirements for the proposed system is to provide a programming environment which gives ease of programming, the synchronisation provided by the system should take a

generic form and be as close as possible to the synchronisation provided in existing shared memory code. Since semaphore type synchronisation is used for programming in most centralised memory systems [Dinning 89], [Ramachandran and Singhal 95], it is more familiar to users than the lock type synchronisation used by many DSM implementations [Carter et al. 95b], [Keleher 95]. Thus, semaphore type synchronisation is used in the proposed DSM system to provide mutually exclusive access to shared variables. Synchronisation points are used as points at which memory is updated in weak consistency models. A weak consistency model is used for the proposed DSM system; for weak consistency models synchronisation operations must be visible to the memory server. Hence, the semaphores are implemented completely within the operating system as variables in the memory server.

As stated earlier, semaphore ownership is decentralised to avoid the obvious bottleneck and fault tolerance problems of a centralised ownership system. A request for entry to a critical section *wait*() is forwarded along a probable owner chain until the current owner is located. If a process is already in the critical section, the request is queued in the request queue to be handled later. If no process is in the critical section, the request is acknowledged immediately and the requesting memory server will become the new owner of the semaphore, as shown in Figure 3.14. If there are requests in the request queue, when a process executes a *signal*() (exits a critical section), the request at the head of the queue is handled. The remainder of the queue is forwarded to the requesting memory server along with the acknowledgement to the request, as shown in Figure 3.15.

Barriers are used to coordinate executing processes. Processes block at a barrier until all processes have reached it; the processes then all continue execution. As stated earlier, barrier management is centralised, the memory server on one of the workstations in the COW is designated the central barrier manager. As such it handles the control and management of the barriers. When a user process executes the *dsm_barrier_req*() primitive the process will block. Control is then passed to the memory server which checks the *consistency_model* for the memory associated with the process. If the *consistency_model* is *RELEASE* the *distribute_diffs*() primitive is called, as shown in Figure 3.16. When all *diffs* have been distributed and replies

```
wait(semaphore)
{
     if semaphore is REMOTE
          forward request to probowner
     else if semaphore is LOCKED
          add request to semaphore request queue
     else if semaphore is RELEASED
     {
          if requesting process is remote
          {
               set semaphore to REMOTE
          }
          else
          {
               set semaphore to LOCKED
          }
          reply to requesting memory server
          set probowner to new owner
     }
}
```

Figure 3.14 Pseudocode for Wait

```
dsm_barrier_req(barrier)
{
     if consistency_model is RELEASE
     {
          execute distribute_diffs()
          wait for all replies
     }
     send message to barrier manager
     wait for reply
}
```

Figure 3.16 Pseudocode for DSM Barrier Request

received the memory server sends a message to the central barrier manager. When the central barrier receives this message it executes the *dsm_barrier*() primitive which is

```
signal(semaphore)
{
     if consistency_model is RELEASE
          execute distribute_diffs()
     if request_queue is empty
          set semaphore to RELEASED
     else
     {
          set semaphore to REMOTE
          remove first request in queue
          prepare reply message for this request
          if request queue contains requests
               attach queue to message
          send message
     }
}
```

Figure 3.15 Pseudocode for Signal

shown in Figure 3.17. When messages from all processes indicating that they have

```
dsm_barrier(barrier)
{
     increment barrier_cnt for barrier
     if the barrier_cnt equals num_procs
          send messages to all dsm processes to continue
}
```

Figure 3.17 Pseudocode for DSM Barrier

reached the barrier have been received, the central barrier manager replies to all of the DSM parallel processes which then unblock and continue their execution.

3.5.6 Consistency Model

As stated in Section 3.5.1.1 the sequential and release consistency models have been selected for the proposed DSM system. Their design is elaborated in the following subsections.

3.5.6.1 Write-Invalidate DSM system

In the write-invalidate protocol a single read/writable copy or multiple read-only copies of any page can exist at one time.

Following the design presented in Section 3.5.2 the system should react to the following three types of DSM page faults (events):

- page_copy —a read fault on an offsite page;
- page_write —a write fault on a read-only page; and
- page_copy_write— a write fault on an offsite page.

All three page faults are initially handled by the *exception_handler*() primitive of the memory server on the workstation on which the fault occurred. The pseudocode

```
exception_handler(fault_address, type_of_fault)
{
     if consistency_model is SEQUENTIAL
     {
          if type_of_fault is read fault
               page_copy_req(fault_address)
          else if type_of_fault is write fault
          {
               if the page is offsite
                    set page_version to zero
                page_copy_write_req(fault_address)
          }
     }
     if consistency_model is RELEASE
     {
          twin the page
          make page_permissions read_write
     }
}
```

Figure 3.18 Pseudocode for Exception Handler

of this primitive is shown in Figure 3.18. The primitive forwards the request to the probable owner's memory server if the page ownership is not local. The chain of

probable owners is followed until the current owner of the page is located. The current owner completes the handling of the fault.

Page_copy and page_copy_write faults result from a read and write access respectively being attempted on a page which is offsite, i.e. does not exist in the memory of the local workstation. The pseudocode of two primitives proposed to handle these faults are shown in Figures 3.13 and 3.19. Both page_copy and

```
page_copy_write_req(fault_address)
{
     if the page_status is REMOTE then
          forward request to probowner
     else if the page_status is LOCKED then
          add request to request queue
     else if the page_status is OWNER then
     {
          set the page_status to LOCKED
          if the page_version number equals the message ver-
            sion number then
               page not required
          else if page_version number > message version
            number then
               page required
          if copyset empty
               invalidate the local page
          else
               send page_invalidate_req to copyset
     }
}
```

Figure 3.19 Pseudocode for Page Copy Write Fault

page_copy_write faults result in a copy of the requested page being sent to the requesting workstation. The *page_version* number (discussed later in this section) is set to zero to indicate that the page is required.

If write access is required the *page_invalidate_req*() primitive, shown in Figure

3.20, is invoked which causes a message to be sent to all members of the copyset,

```
page_invalidate_req(fault_address)
{
     send page_invalidate message to all workstations in
        the copyset
}
```

Figure 3.20 Pseudocode for Page Invalidate Request

requesting them to invalidate their local copy of the page. As each member of the copyset is invalidated a reply is sent to the page owner. When all of these replies have been received by the owner the *page_invalidate*() primitive, shown in Figure 3.21, is

```
page_invalidate(fault_address)
{
     When all acknowledgements received
     set page_status to REMOTE
     prepare page_copy_write message
     if page required
          attach page to page_copy_write message
     if request queue is not empty
          attach request queue to page_copy_write message
     send page_copy_write message
     update probowner to new owner
     if the requesting memory server is not local
          invalidate the local copy of the page
}
```

Figure 3.21 Pseudocode for Page Invalidate

executed which sends the page (and any requests queued for handling) to the requesting workstation before invalidating its local copy of the page. When the page reaches the requesting workstation the *page_copy_write*() primitive is executed and the page is mapped into the memory, as shown in Figure 3.22. The page is given read permissions if read access was required and read/write permissions if write access was required. The page is then locked for an interval so that the user process has a "time-

```
page_copy_write(fault_address)
{
     if page is accompanying message
     {
          map page into local memory
          update version number
     }
     send message to Process Server to restart process
     if required access is read-only
          make page read-only
     else if required access is read-write
     {
          increment page_version number
          make page read-write
          set page_status to LOCKED
          set timer for page_lock
     }
}
```

Figure 3.22 Pseudocode for Page Copy Write

slice" in which to complete its write operation. The length of the interval must be determined experimentally because it will vary according to the hardware used. Any requests for the page while it is locked are queued to be handled later.

An attempt to write to a read-only page results in a page_write fault which also triggers the execution of the *page_copy_write_req* primitive, shown in Figure 3.19. A request for write permission is sent to the memory server which owns that page. As in the case of the page_copy_write fault, a message which triggers a *page_invalidate_req*() primitive, shown in Figure 3.20, which is sent to each workstation in the copyset. When all of the replies have been received the *page_invalidate*() (Figure 3.21) primitive is executed. It sends the page and any requests queued for handling to the requesting workstation before invalidating its local copy of the page. When the page reaches the requesting workstation the *page_copy_write*() primitive (Figure 3.22) is executed. The page is given read/write permissions and locked for its time-slice.

A *page_version* number is a number associated with each page. The *page_version* number is incremented each time a write operation is performed on the page. If the version number in the message requesting write access to the page is smaller than the version number on the owner's page this indicates that the page has been updated and the page itself must accompany the write permission message.

There are two circumstances in which this *page_version* number is used:

- When a page_write fault occurs where a write access is attempted on a page which is offsite. As stated earlier the *page_version* number is set to zero for this type of page fault. This will force the reply message to append the page itself to the message granting write permission.
- When a page_write fault occurs where a write access is attempted on a page which has read-only permissions. Because any page being written to is locked for a time-slice, requests can be queued in the DSM system to be handled later. Thus it is possible that a page may have been made invalid before its write request is processed. In this case, write permission could be granted to a non-existent page. If the owner of the page detects that version number in the request for the page is smaller that the version number on the local page then, when the write request is processed, the page itself is sent to the requesting memory server. If the numbers are the same, only a message granting write permission is sent.

3.5.6.2 Write-Update DSM System

The write-update protocol is used in the proposed DSM system to implement release consistency. In the release consistency model the shared memory is made consistent when the process exits from a critical section and before departure from a barrier.

Attached to each DSM process is an identical block of shared memory, the contents of which are in a consistent state. Initially each page in the shared memory is set to read-only. Thus any attempt to write to a page in the shared memory causes a write fault which triggers the *exception_handler*() primitive (Figure 3.18). If the memory is labelled as using the update protocol, the page is then twinned and the status

of the page is changed to read-write by the memory server before the write can proceed. Any further write accesses to that page will not cause a write fault because the protections of the page have been changed to read-write. Pages are twinned when a barrier is been reached and passed, and when a critical section is exited. Before a user process may exit a critical section the *distribute_diffs*() primitive is executed. In this primitive all of the pages that have been twinned since the latter two events last occurred are compared word by word with their twins to identify updates made to the shared memory. These updates are then bundled together into a message by the operating system and sent to all workstations in the system which share the same memory. The pseudocode for the *distribute_diffs*() primitive is shown in Figure 3.23.

```
distribute_diffs(memory_id)
{
      while pages in memory not examined
            if page has been twinned
            {
                  prepare diff message
                  compare original page with twin
                  append any differences to message
            }
      send diff message to all workstations sharing memory
}
```

Figure 3.23 Pseudocode for Distribute Diffs

The receipt of this message triggers an *update_pages*() (Figure 3.24) primitive in which the updates are applied to the shared memory on all the latter workstations and replies are then sent to the originating workstation. When the originating workstation has received all replies the user process may exit the critical section.

The functioning of the above consistency models is the same when the memory is physically shared. In other words, it is transparent to the user if more than one DSM processes physically share memory. The protections on the shared memory region which is also physically shared are read-only. An attempt to perform a write access on a page in the shared memory region results in a write fault. The page is twinned and the

```
update_pages(memory_id)
{
      while diff message remains
      {
            Locate page number in message
            Locate equivalent page in shared memory
            Locate address at which update occurred
            Replace memory address with updated data
      }
      reply to sender indicating all diffs have been
         applied
}
```

Figure 3.24 Pseudocode for Update Pages

protection for that page is changed to read-write so that further accesses will not trigger a write fault. The write operation can then proceed.

When the user process attempts to exit the critical section the memory server generates and sends the update message containing all changes made to the memory to all remote workstations. Since the local user process physically shares the memory with the process which generated the updates through write operations, these updates are already visible to this process.

3.6 Conclusions

This chapter reported on the outcome of the research into the synthesis of a new DSM system integrated into a microkernel and client-server based distributed operating system. The reasons for choosing this type of operating system and for the location of the DSM in the memory server were that shared memory is a resource which requires management and since DSM is a memory management function the memory server is the server into which the DSM system must be integrated.

In this chapter the major achievements are: the new location of a DSM system within the memory server of a microkernel and client-server based distributed operating system; and the synthesis of a novel DSM system which is an event driven set of modules described by two basic protocols, the DSM parent protocol and DSM

child protocol. The proposed DSM system is made up of an event-driven set of co-operating and distributed entities. Furthermore, the proposal of an automatic initialisation algorithm for the parallel DSM processes provides a novel way of using the scheduling ability of the operating system to make the programmer's task easier. The inclusion of two consistency models in the proposed design makes it possible for application programmers with a knowledge of DSM to choose the model they think is the most appropriate for their application.

The design requirements for the proposed DSM have been identified as ease of programming, ease of use, transparency, efficiency and scalability. The proposed design achieves all of these requirements. In particular,

- *Ease of Programming* has been achieved through the use of the shared memory programming model with the use of the familiar semaphore type synchronisation, with only the addition of barriers which are simple to use and understand.
- *Ease of Use* has been achieved through the use of automatic initialisation. The operating system handles the decisions about the number of child processes to start and where to execute them, in fact the system actually starts the child processes on the remote workstations.
- *Transparency* has been achieved through the sharing of fixed sized memory blocks which has allowed the integration of the DSM into the memory management of the operating system. In addition, by requiring little or no additional input from the programmer the DSM has met the transparency requirement. Moreover, there is no apparent difference from the user's perspective between physically shared and distributed shared memory.
- *Efficiency* could be achieved through the placement of the DSM in the operating system allowing the DSM to use the low-level operating system functions. The DSM uses a weak consistency model to reduce the number of consistency related messages. In addition to this, distributed page and semaphore ownership improve the efficiency by reducing the potential bottlenecks of centralised managers.
- *Scalability* could be achieved through the use of distributed techniques for

data location and synchronisation.

Thus the DSM proposed in this chapter has met all of the design requirements as shown above. In addition the services provided by the proposed DSM system are such that this DSM system places more emphasis on programmer convenience than previous designs while not forsaking efficiency. Thus, this allows the achievement of the research goal of providing users with an environment in which they can easily develop and run their programs.

Chapter 4 Distributed Shared Memory integrated into the RHODOS Distributed Operating System

4.1 Introduction

In the previous chapter the synthesis of a new DSM system which is integrated into a microkernel and client server based distributed operating system was detailed. This followed the claim that the use of the latter class of operating system and the placement of the DSM system within the memory server has two advantages:

- it allows the development of a DSM system integrated into an operating system which results in a new class of distributed operating systems able to efficiently and transparently support parallel processing on COWs; and
- it forms an execution environment which attends to all of the user's needs, in particular ease of programming.

The goal of this chapter is to demonstrate the feasibility of the proposed DSM system synthesised in Chapter 3. For this purpose, this chapter describes the development of the DSM system within the memory management of the RHODOS distributed operating system. Furthermore, the DSM system described here validates the claims of the previous chapter that a DSM system integrated into an operating system is not only transparent but improves the ease of use and programming for users. Results of performance tests presented in the Chapter 5 furthermore show that this approach leads to an efficient DSM system.

The operating system selected for this research is the RHODOS (ResearcH Oriented Distributed Operating System) Operating System. RHODOS is a microkernel and client-server based distributed operating system which was considered the best environment for the synthesised DSM system based on its flexibility and configurability.

This chapter is organised in the following manner. The first section of the

chapter contains a description of the RHODOS distributed operating system and its components, paying particular attention to those components which are directly involved in and support the DSM system. As described in Chapter 3, in order to provide users with an easy-to-use DSM environment, automatic initialisation of the DSM processes and shared objects have been designed. The semantics of this development are presented. It was decided that RHODOS DSM should support both write-invalidate and write-update based coherence protocols, in order to provide an appropriate consistency model and to compare them within one environment. Thus, the design and semantics of the write-invalidate and write-update based DSM systems and their relationship with the semaphore-type synchronisation approach are discussed and presented. The write-invalidate and write-update based coherence protocols are used to implement the sequential and release consistency models, respectively. Included in these descriptions is an account of the memory management of the operating system into which the DSM system is integrated. Following this, the semantics of barriers in RHODOS are presented. Furthermore, the design and semantics of physically shared memory in RHODOS are reported, highlighting the fact that distributed and physically shared memory are employed by users in precisely the same manner.

4.2 RHODOS Distributed Operating System

RHODOS is a microkernel based distributed operating system which uses the client-server approach. The basic communication paradigm used by RHODOS is message passing. RHODOS is a modular, scalable and portable operating system and as such it can be used as a testbed for various research topics. It is composed of the Microkernel (the Nucleus) and several cooperating System and Kernel Servers. Interface between processes and the Microkernel is through system calls [De Paoli et al. 95].

RHODOS supports three levels of processes: Kernel Servers; System Servers; and User Processes, which execute in user mode and are controlled by the Microkernel as shown in Figure 4.1. User Processes have the lowest priority. They have no special privileges and can only access services and resources through calls to the Microkernel and System Servers.

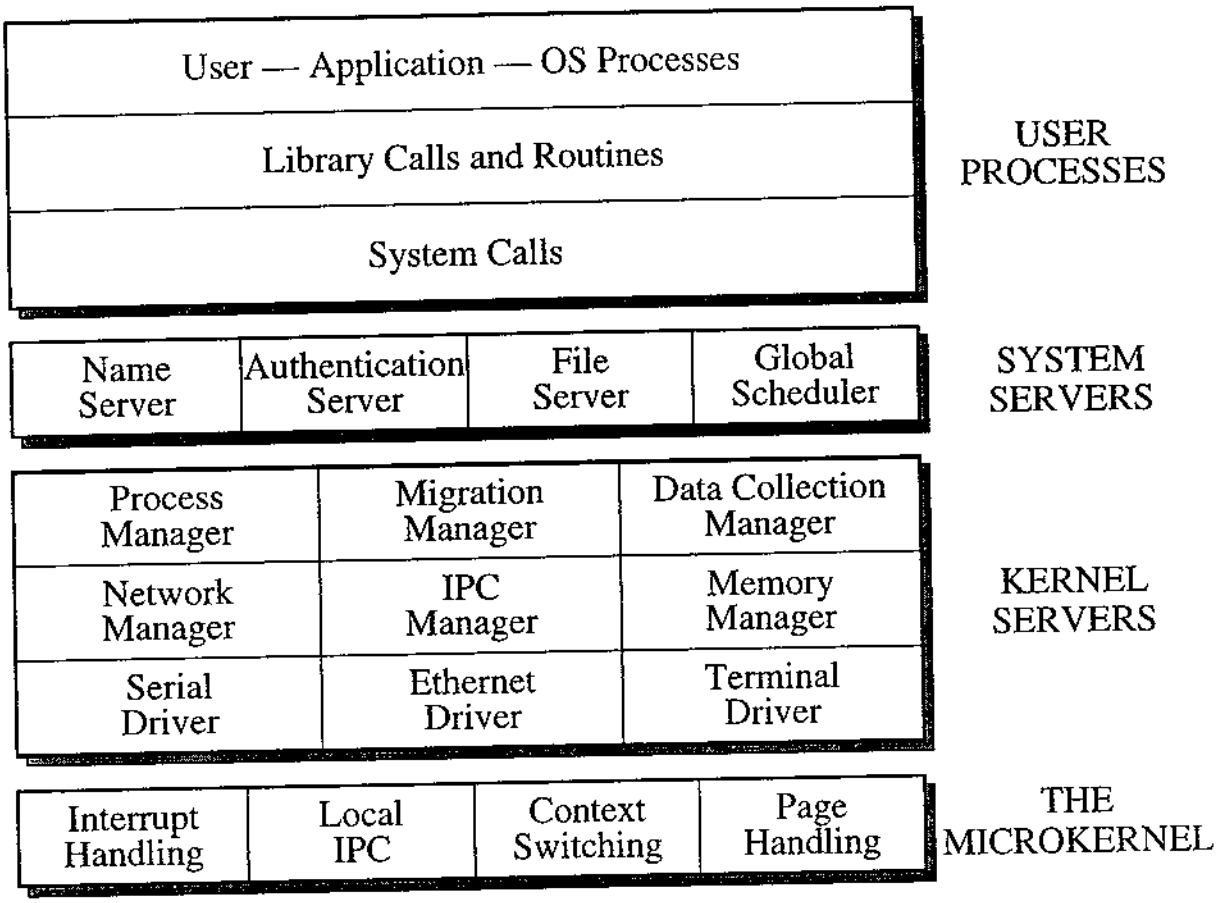

Figure 4.1 The Process Layers of RHODOS

System Servers are next in the process hierarchy after User Processes. They have some privileges and can communicate directly with the Kernel Servers. System Servers form a virtual computational environment and provide to users such services as file management, name management and global scheduling. However, like User Processes, System Servers must use standard system calls to access resources. The Kernel Servers are at the lowest logical level in the process hierarchy. They are responsible for the management of system resources such as processors, processes, memory, network and drivers. Kernel Servers are able to alter Microkernel data and can access privileged system calls.

The Microkernel in RHODOS, which is multithreaded, is an interface between the hardware and the rest of the operating system. All machine dependent code is in the Microkernel. Consequently, the Microkernel abstracts away from the hardware and provides a system that can easily be ported to different hardware platforms. The

Microkernel is responsible for a small set of services: local interprocess communication, basic memory management, context switching and interrupt handling. The remainder of the functions and services necessary for the operating system to function are, thus, provided by the Kernel and System Servers [De Paoli et al. 95].

The Kernel Servers, which are also called Managers, perform functions which would normally be carried out by a Monolithic Kernel, thus they are trusted entities. The eight Kernel Servers in RHODOS are [De Paoli et al. 95]:

- *Process Manager* — Manages all processes. This manager will be discussed further in the next section.
- *Space Manager* — The Space Manager manages the spaces in the system. This manager will also be discussed further in the next section.
- *Interprocess Communication (IPC) Manager* — The IPC Manager manages interprocess communication between processes on remote workstations. This manager will be discussed further in the next section.
- *Network Manager* — The Network Manager provides the interface with the communication network by supporting the lightweight transport, network and data link layer functions, following the ISO/OSI RM. It does this by supporting the IPC Manager in the delivery of messages to remote workstations.
- *Migration Manager* — The Migration Manager controls the migration of running processes between workstations. The Migration Manager provides a mechanism for dynamic load balancing which is a component of the Global Scheduler. The Global Scheduler decides on which process should be migrated where and when.
- *Remote Execution (REX) Manager* — The REX Manager coordinates all user processes in the system. On creation, twinning or exiting of a process the user process must contact the REX Manager which has knowledge of all processes and their resources. Process creation can either be performed on the local or a remote workstation. As a result of this, the REX Manager provides a mechanism for the static allocation component of the Global Scheduler. The Global Scheduler lets the REX Manager know whether a

process create request should be performed locally or remotely.

- *Data Collection Manager (DCM)* — The DCM collects information (computational load and communication patterns) from the running system and generates data regarding individual processes, their communication and workstation use (available resources). This information is used by the Global Scheduler and by developers to assess software performance.
- *Device Manager* — The Device Manager provides an interface, through the device drivers, between physical devices and processes.

The DSM system manages shared memory objects of competing/cooperating parallel processes. Thus, the Kernel Servers most relevant to DSM, the Process Manager, Interprocess Communication Manager and the Space Manager, are shown in Figure 4.2. These three servers will be discussed further.

4.2.1 Process Manager

The Process Manager manages all RHODOS processes. To do this it controls the process queues and cooperates with other kernel servers (Remote Execution Manager and Migration Manager) and the Microkernel. It performs several operations on processes; make a process wait for its child to exit, exit a process, kill a running process, put a process on the frozen queue prior to migration and handle process exceptions.

The design and implementation of the RHODOS DSM system is based on the proposed DSM system, presented in Chapter 3. However, it reflects also the design features of the RHODOS system. One of them is the implementation of a process. The RHODOS process from the resource point of view, consists of at least the following three spaces, text, data and stack spaces.

4.2.2 Interprocess Communication (IPC) Manager

The IPC Manager manages interprocess communication between processes on remote workstations and supports group communication. It handles: address resolution; group membership; message ordering between communicating processes; and communications during process migration for the Migration Manager.

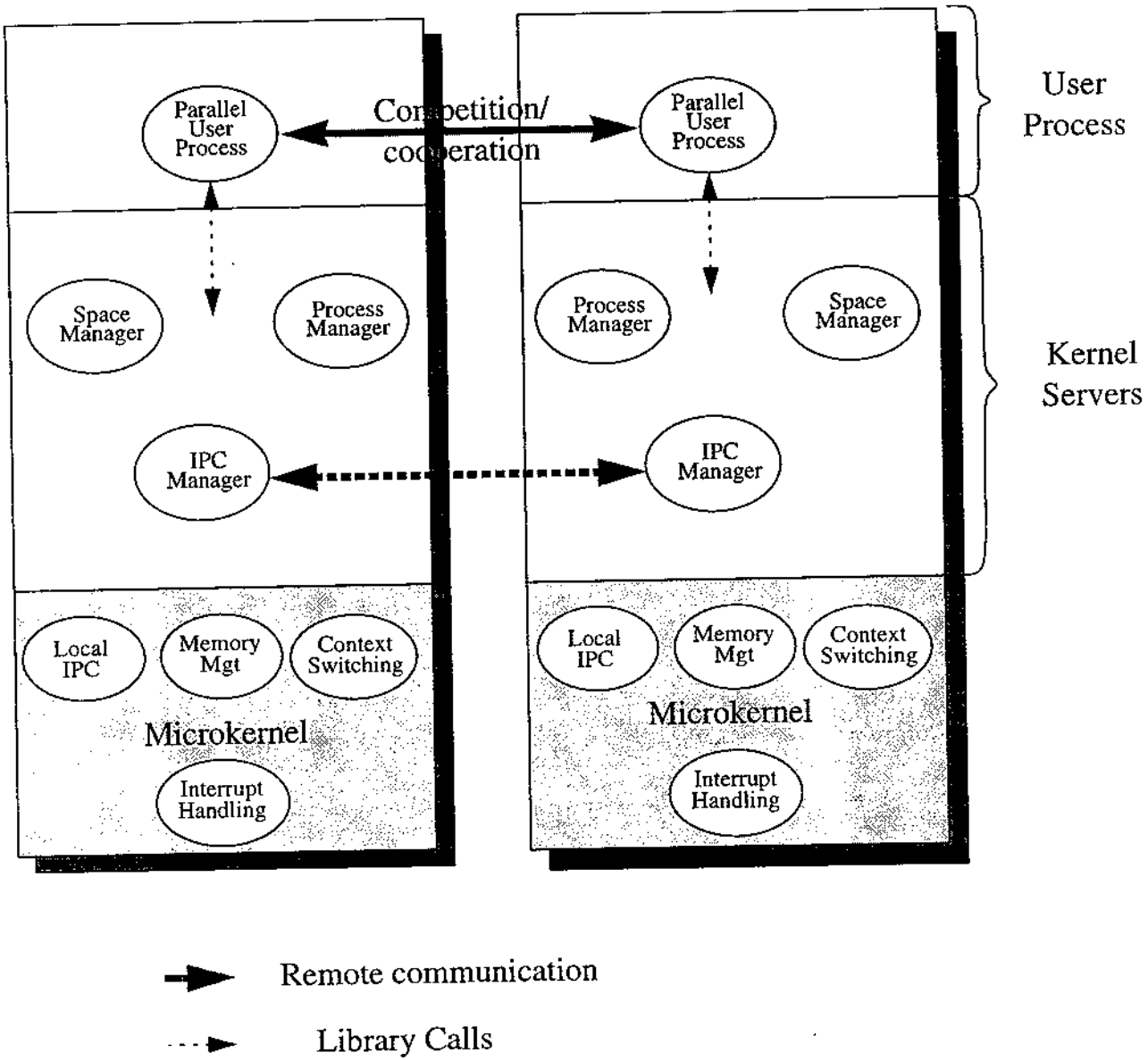

Figure 4.2 RHODOS as used for DSM

4.2.3 Space Manager

The Space Manager manages the spaces in the system. Spaces are logical blocks of data which are mapped to the physical memory. The functions of the Space Manager are to create new spaces and copy, migrate, alter and delete existing spaces. The Space Manager also provides flexible paging functions which assist process migration and DSM functions [Hobbs et al. 95]. In Chapter 3 the synthesis of a new DSM system is presented. According to this approach, the DSM system is integrated

into the Space Management's functionality. This requires the memory management of RHODOS to be covered in more depth (Section 4.3).

4.3 Memory Management in RHODOS

Memory management in RHODOS is divided into the hardware dependent portion which resides in the Microkernel and the hardware independent Space Manager which operates on RHODOS' logical memory units, spaces.

4.3.1 RHODOS Spaces

The fundamental element of memory in RHODOS is a space. Spaces are logical "chunks" of data either in memory or on disk or "a contiguous segment of virtual memory having a known start address and a known length" [De Paoli et al. 95]. RHODOS' spaces are uniquely identified through their system names or SNames [Hobbs et al. 95]. The logical space grouping is, in fact, made up of physical pages. Space management provides a mapping for these physical pages to a region of virtual memory.

4.3.2 The Space Manager

The memory management primitives in RHODOS are provided by the Space Manager in conjunction with the Microkernel. The Microkernel operates on memory at the lowest level by directly manipulating the hardware. The Space Manager provides a level of abstraction between the Microkernel's operations upon the hardware and the rest of the system, providing a better interface for the execution of memory management functions. A set of space structures containing space information resides in the Microkernel data area. When a process is scheduled out the hardware memory mappings (in the Sun 3/50 these are provided by the MMU (Memory Management Unit)) are saved in these space structures. These mappings are reloaded from the space structures when the process is rescheduled. The Space Manager is one of the Kernel Servers and as such is capable of changing Microkernel data; it manipulates memory through a set of primitives which act upon the space data structures together with lists of free and used pages.

The Space Manager currently supports two types of page operations:

- *Copy on write* — Write operations performed on copy-on-write pages are trapped by the Microkernel which then makes a copy of the page before the write can proceed. This allows twin processes to share pages while they are reading them but makes separate copies when either process attempts to write to the page.
- *Copy on reference* — Copy-on-reference pages are used in process migration. Pages are only migrated to a new workstation when they are referenced. Thus a read or write operation on a copy-on-reference page results in a trap which is caught by the Microkernel and passed to the Space Manager. The Space Manager then transfers the page from the old to new workstation by sending a copy of the page and deleting the original page before the read or write operation can proceed.

4.3.3 Space Manager Primitives used for DSM

The primitives within the Space Manager create, copy, alter, delete and migrate spaces when they are called to do so by the Microkernel or other processes. Certain Space Manager primitives are of significance in the development of DSM in RHODOS. They are the exception handler primitives and those that create and migrate spaces. These primitives are as follows:

- *exception_handler(procid_pointer, exception_address)* — This function handles the exceptions passed to it by the Microkernel. In particular, it identifies copy_on_write and copy_on_reference pages and passes control to the appropriate part of the Space Manager to handle these types of page fault.
- *space_create(process_sname, base_address, length, default_protection, space_type, return_sname)* — This primitive creates a space of *length* size at *base_address* containing pages which have *default_protection* protections and a space type of *space_type*. The protections indicate whether pages are read_only or read_write. Space types are used to differentiate between boot, nucleus, text, data, stack and swap spaces. When the space is created it is attached to process *process_sname*.
- *space_attach(space_pointer)* — This primitive will share the space

space_pointer between the owner of the space and the calling process.

- *page_copy_req(fault_address)* — This primitive requests that a copy of the page on which the fault occurred be sent to the return address of the request. The parameter *fault_address* contains the address at which the page fault occurred; this primitive is executed when a message is received from the Space Manager of the workstation requesting this page. This primitive will locate the page, send a copy to the requesting workstation and delete its own copy of the page.
- *page_copy(fault_address, page_contents)* — This primitive is executed when the page on which the fault occurred is received it maps the requested page into the local space. The parameter *fault_address* contains the address at which the page fault occurred and *page_contents* contains the page contents. The primitive uses *fault_address* to locate the page and populates it with *page_contents* and sends a message to the Process Manager to restart the process.

4.4 Development of Automatic Initialisation System

In RHODOS an executing DSM application is made up of one or more DSM processes, which share memory, executing on one or more workstations. These processes require initialisation. The design of the automatic initialisation process is described and justified in Chapter 3. This process includes the initialisation of the process synchronisation. Synchronisation is in the form of semaphore-based mutual exclusion and barrier-based process co-ordination. These semaphores and barriers must both be initialised before execution of the application commences.

As stated in Chapter 3, these basic initialisation operations could either be carried out manually, by the programmer, or automatically, by an initialisation system. Automatic initialisation has been proposed in Chapter 3 as a solution since it satisfies the design requirements of this DSM system.

4.4.1 Data Initialisation

The memory approach developed for RHODOS implies that a DSM shared

object is a space. Since spaces are made up of one or more pages, the unit of granularity used in RHODOS' DSM system is a page. The use of the space as the shared object requires the extension of the resource model of a process in RHODOS. The following spaces form the memory resource model of a RHODOS process: stack, data and text spaces. The memory attached to each DSM parallel process is made up of a local portion (the data space) and a global portion (the DSM space). The DSM spaces on all workstations are identical in size and position. Thus, the DSM parallel process spaces are shown in Figure 4.3.

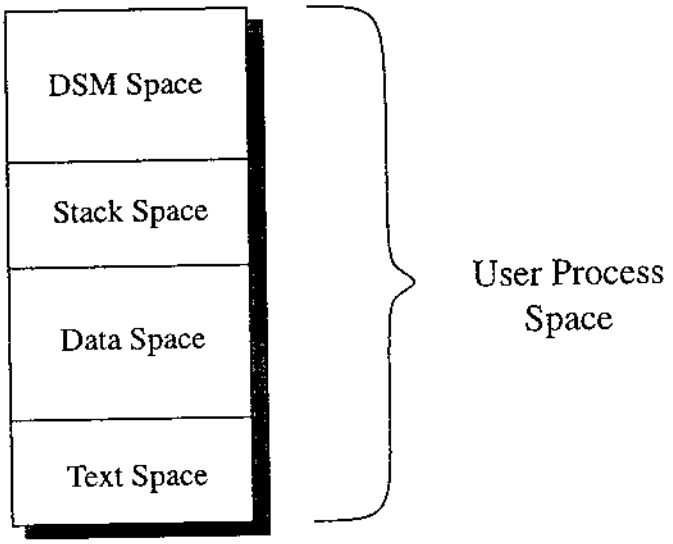

Figure 4.3 The User Process Space of a process using RHODOS DSM

The RHODOS synchronisation and co-ordination have been developed using semaphores and barriers. The main semaphore data structure, the *dsm_semaphore_table*, shown in Figure 4.4, holds information about the semaphores in five fields. The *sem_name* field contains the semaphore's SName or unique identifier. The *space_sname* field contains the SName of the DSM space originally created by the Parent Process. The status field contains the current status of the semaphore. This field can have one of three values as follows:

- LOCKED — a user process is currently in the critical section;
- REMOTE — the local Space Manager does not own the semaphore. Any request for this semaphore should be forwarded to the *probowner*; or

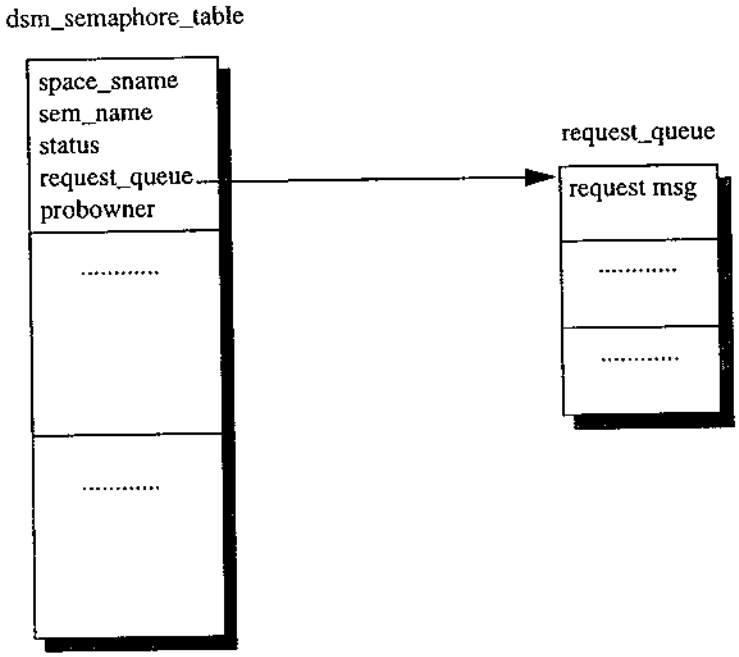

Figure 4.4 The dsm_semaphore_table Data Structure

- RELEASED — the semaphore is owned by the local Space Manager and no user process is currently in the critical region.

The *status* field is initialised to RELEASED on the Parent's workstation and REMOTE on the workstations of all Child Processes. (This means that all semaphores are initially owned by the Space Manager on the Parent's Workstation. This could become a bottleneck if there were a large number of semaphores. This implies that the initial ownership should be distributed among the workstations.) The *request_queue* field is used by the current owner of the semaphore; it contains a pointer to a queue which contains a list of all requests for the semaphore which have arrived while the semaphore's status was LOCKED. The *probowner* is used by the Space Manager of a process requesting entry to a critical section to locate the Space Manager which currently owns the semaphore.

The main barrier data structure, the *dsm_barrier_table* (Figure 4.5) holds information about the barrier in five fields. The *barrier_name* field contains the SName of the barrier. The *space_sname* field contains the SName of the DSM space originally created by the Parent Process. The *barrier_count* is used by the Barrier Manager as a count of the number of barrier messages received and *barrier_max* is the number of

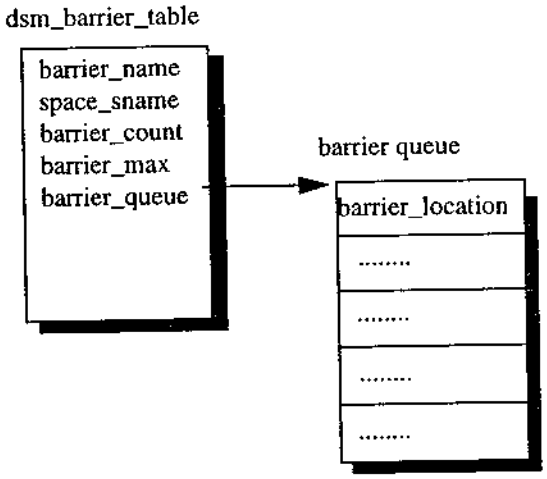

Figure 4.5 The dsm_barrier_table Data Structure

processes using the barrier. The *barrier_queue* field is also only used by the Barrier Manager; it contains a pointer to a list which contains the locations of all processes using the barrier.

4.4.2 DSM Parallel Process Initialisation

Parallel execution of a program means that there is a sequential process, which at one stage of execution forms a set of parallel processes (children of the parent process) [Goscinski 97]. Thus, the application is executed in parallel by a group of processes made up of a single Parent Process and a group of one or more Child Processes. Thus, the next step in the process of parallel application processing is to create the Child Processes. These processes can be created explicitly by the user on each workstation. However, as a result of some of the research reviewed in Chapter 2, this is considered to be an irksome burden for an application programmer. These processes should be created automatically by the operating system. This solution has been proposed in Chapter 3.

On RHODOS remote processes such as the Child Processes can be created by the operating system using either:

- *process_twin*() to create a copy of the local process followed by process migration to move the copy to a remote workstation; or
- *remote_process_create*() to create processes directly on the remote

workstations from a single executable image on disk.

The migration of the Child Process would place an extremely heavy load on the Parent Process' workstation, a potential bottleneck, and slow down the initialisation process. A far more efficient solution is to create the processes directly on the remote workstations. Remote process create, on the other hand, involves the DSM spaces being created concurrently on each participating workstation. The only possible bottleneck is the access to the executable image of the process on the shared disk. Remote process creation is, therefore, considered to be the better option.

The decision regarding the workstations on which to create the Child Processes is important for the speed of execution of the application as well as the functioning of the whole operating system. While the user should be able to indicate the optimal number of processes to execute an application, the operating system should be able to make the final decision based on the system load. The Global Scheduler in RHODOS maintains system load information. Thus, the operating system is able to employ the Global Scheduler as well as the Remote EXecution (REX) Manager to begin parallel execution. The REX Manager contacts the Global Scheduler which returns system load information to the REX Manager. Based on this information the REX Manager is able to identify idle or lightly loaded workstations which are available to take part in parallel execution of the application. The REX Manager creates a single Child Process from an executable image on disk on each of the remote workstations identified.

4.4.3 DSM System Initialisation Semantics

In Figure 4.6 the sequence of events and messages surrounding the automatic initialisation of the DSM system on RHODOS is depicted to the point where the semaphores and barriers are initialised.

The figure shows two workstations, the Source and Remote Workstations. Depicted on these workstations are a user process, the Parent Process on the Source Workstation and the Child Process on the Remote Workstation, and the Remote Execution (REX) Manager, Global Scheduler and Space Manager. The DSM Space itself is shown attached to the user processes.

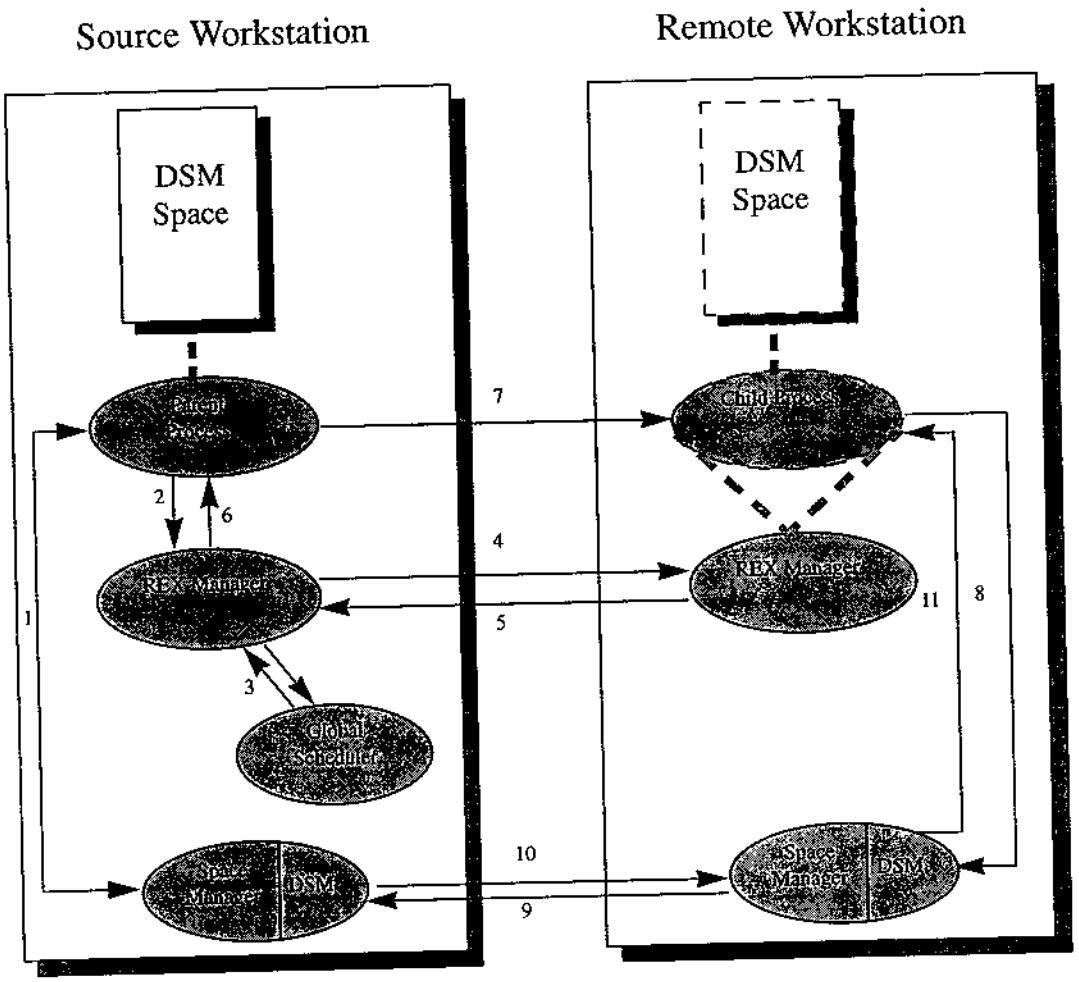

Figure 4.6 Semantics of Automatic Initialisation on RHODOS

As stated in Chapter 3, when a parallel application using DSM starts to execute the first process, the Parent Process, initializes the DSM system. The sections of code shown in Figures 4.7. and 4.8 are the initialisation code required by the Parent and Child Processes.

The initialisation process starts with the Parent Process executing a library call:

```
start_dsm(&sp_name,consistency_model,memory_size,
        opt_num_procs, sem, barrier, numsems, numbar-
        riers)
```

```
/*Declare a pointer to DSM global memory*/
GlobalMemory *glob = NULL;
main(){
     SNAME psn;           /*process sname */
     char    child[20];

/*DSM LOCAL VARIABLES*/
     SNAME sp_name;       /* declare space sname variable*/
     int num_procs;
     int opt_num_procs; /*optimal number of processes*/
     int memory_size;

/*synchronisation variables*/
     int numsems, numbarriers;
     SNAME barrier[numbarriers], sem[numsems];

     memory_size = sizeof(GlobalMemory);

      glob = (GlobalMemory *) start_dsm(&sp_name,
             consistency_model,memory_size,
             opt_num_procs, sem, barrier, numsems,
             numbarriers, &num_procs);
     ..............
     ..............
```

Figure 4.7 DSM Initialisation code for Parent Process

The *start_dsm*() library call (Figure 4.7) closely follows the design of the *start_dsm*() primitive shown in Section 3.5.3. For implementation purposes, this library call must have additional parameters. These parameters are variables which are assigned during the execution of the library call; they are not shown in Chapter 3's *start_dsm*() primitive. The *start_dsm*() library call initially requests the Space Manager to allocate a block of memory for the DSM space (Message 1) and place the SName of the space in *sp_name*. The Space Manager creates the space using the *space_create*() library call (the *dsm_create*() primitive in Chapter 3), which creates a space at a specified base address of *memory_size* bytes labelled with *consistency_model*. The

consistency_model parameter can either be labelled RELEASE (uses the write-update coherence protocol) or SEQUENTIAL (uses the write-invalidate coherence protocol). The newly created DSM space is given read_only permissions and then attached to the Parent Process and the SName of the created space is placed in *sp_name*.

When the DSM space has been created as part of the *start_dsm*() library call on the Parent Process' workstation, the execution of remote processes is started as follows:

```
num_procs = dsm_parinit ("child", opt_num_procs,
        *sp_name);
```

where *opt_num_procs* is the user supplied parameter which indicates the optimal number of processes required to execute the application. The operating system uses the services of the Remote EXecution (REX) Manager (Message 2) and the Global Scheduler to begin parallel execution. *child* is the name of the executable file of the child code.

The Global Scheduler nominates a number of available workstations, up to a maximum of *opt_num_procs*, and sends the addresses of these workstations to the REX Manager on the Parent Process' workstation (Message 3). The REX Manager then uses remote process creation to create a single Child Process on each of these available workstations (Message 4). The remote processes or Child Processes are created remotely from the executable image on disk (*child*). As each of the Child Processes is created the Parent Process' REX Manager is informed (Message 5). When all of the Child Processes have been created the Parent Process is informed and given *num_procs*, the number of processes successfully created together with the process identification numbers for each of the processes created (Message 6). The Child Processes each execute a *dsm_parstart(&slaveNum, &sp_name, &num_procs, numsems, numbarriers, sem, barrier*) library call as shown in Figure 4.8; this primitive causes the process to block waiting for a message from the Parent Process with the SName of the Parent Process' DSM space. When the latter message (Message 7) is received the Child Process attempts to attach to this space using a *space_attach*() library call. This call passes control to the Space Manager (Message 8) which attempts to perform the space attach. However since the space is on a remote machine the attach

```
/*Declare a pointer to DSM global memory*/
GlobalMemory *glob = NULL;
main(){
/* DSM variables*/
SNAME           sp_name;
SNAME           barrier[2];
SNAME           sem[2];
int num_procs;

uint32_t numsems = 1, numbarriers = 2;
     glob =(GlobalMemory *) dsm_parstart(&slaveNum,
          &sp_name, &num_procs, numsems, numbarriers,
          sem,barrier);
```

Figure 4.8 DSM Initialisation code for Child Process

fails. The Space Manager detects that the space is remote and it sends a message to the Parent Process' Space Manager requesting the DSM space's details (Message 9). The latter replies to this request with the base address of the space, size, and consistency type for the DSM space as well as the number of participating processes (*num_procs*) (Message 10).

Applications which use the Single Program Multiple Data computational model (SPMD) use *num_procs*. Each process executes the same program on its own block of the data set [Goscinski 97]. Each process has a unique number *slaveNum* allocated to it by the Space Manager. This number is used to identify the portion of the data set it should compute.

When the space information is received the DSM space is created the *space_attach*() is completed and Message 11 passes control back to the Child Process.

The semaphore and barrier initialisation is then carried out. On the Parent Process' workstation the library calls used are:

```
initsem(sp_name,sem, num_sems);
initbarrier(sp_name, barrier, num_barriers);
```

These are called from the *start_dsm*() function. The library calls cause the Space Manager to create the *dsm_semaphore_table* data structure, shown in Figure 4.4, and the *dsm_barrier_table* data structure, shown in Figure 4.5, and to initialise these data structures. This phase of the initialisation is not shown in Figure 4.6. When these synchronisation data structures have been created the information is sent to the Child Process' workstations where the semaphores and barriers are created and initialised. The semaphores and barriers are declared as an array of SNames in the Parent and Child Processes as shown in Figures 4.7 and 4.8. The programmer passes the number of semaphores and barriers required to the *start_dsm*() and *dsm_parstart*() functions and the *sem* and *barrier* variables are populated during the function calls.

At this stage each of the DSM processes, Parent and Child Processes, initialise their global and local variables so that their memories are all consistent. The DSM processes then synchronize at the end of this initialization phase, using a *dsm_barrier*() library call, before starting execution of the application.

4.5 DSM in RHODOS

The design of the DSM system integrated into the RHODOS distributed operating system follows the proposed DSM system presented in Chapter 3. RHODOS' DSM employs two consistency models sequential and release consistency. In this section the semantics of memory coherence and synchronisation in RHODOS' write-invalidate and write-update based DSM are described in order to show that the primitives used by programmers perform operations which are completely invisible to the programmers at user level. A simple section of Producer-Consumer code is used to demonstrate the actions taken by the DSM system when the code is executed. Furthermore, a section of skeleton code containing barriers is used to demonstrate the semantics of these barriers and how they are employed in RHODOS' DSM.

4.5.1 A Simple Example of Producer-Consumer Code using RHODOS' DSM

In Figure 4.9 a code segment is shown which will be used to illustrate the semantics of the memory consistency model and semaphore-type synchronisation in the RHODOS DSM system. The code shows a simple example of the classic Producer-

Consumer problem in which it is clear that the synchronisation primitives are no different from those normally used by programmers in shared memory code. The Producer Process produces three numbers and places them into the variables **x**, **y** and **z**, the Consumer Process then consumes the numbers from **x**, **y** and **z**. Since both processes access the three variables the access must take place within a critical section to prevent two processes from accessing it simultaneously. The critical section is

Producer	Consumer
`wait(sem[0]);`	`wait(sem[0]);`
`  place 1 into variable x`	`  consume element in variable x`
`  place 2 into variable y`	`  consume element in variable y`
`  place 3 into variable z`	`  consume element in variable z`
`signal(sem[0]);`	`signal(sem[0]);`

Figure 4.9 Producer-Consumer Code segment

bounded by a wait-signal pair. For these two processes to execute correctly the producer must initially execute before the consumer.

4.5.2 The DSM Table

A DSM data structure is necessary on each workstation to hold information regarding the shared memory region. This data structure is central to the development of DSM on RHODOS. The DSM Table is held in the Space Manager. Figures 4.10 and 4.11 show the structure of the DSM table for the write-invalidate and write-update based systems, respectively. The DSM table exists on each workstation and contains a record for each shared memory region on that workstation. Each record is made up of a unique identifier for the memory region and a list of the page records for each page in the region.

The *page_table* contain a number of fields: *status*, *version*, *probowner*, *request table*, *copyset* and *page_ptr*. The DSM system uses different fields according to the coherence protocol being employed. If the write-invalidate protocol is being used the following fields in the DSM Table are employed:

- *status* — The status can be: OWNER, indicating that the local space

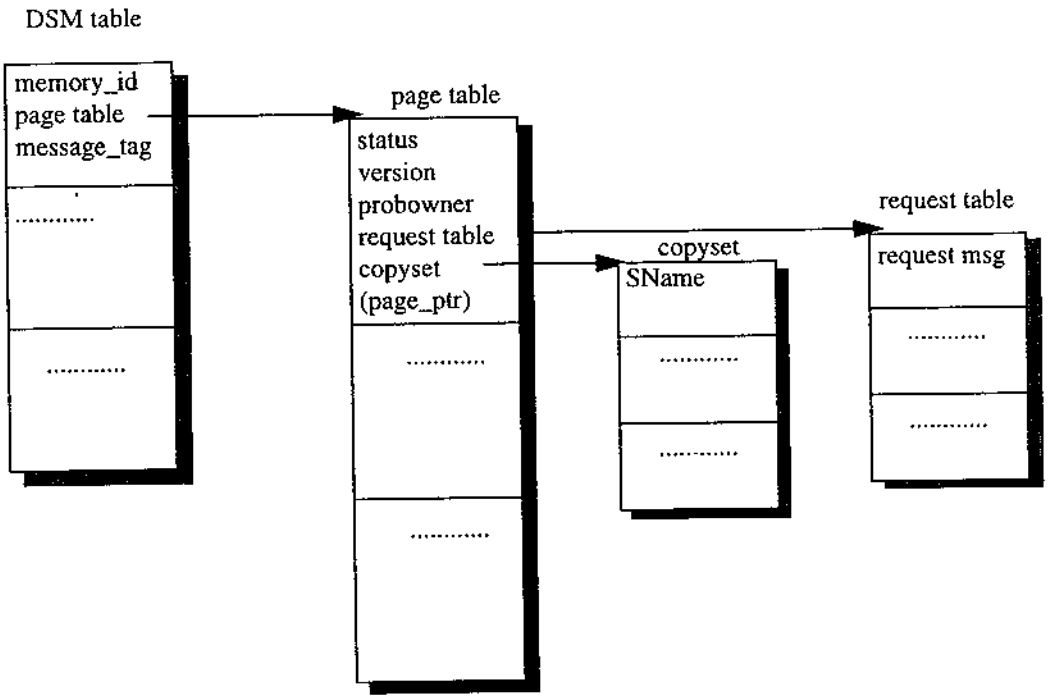

Figure 4.10 The DSM table as used for write-invalidate DSM

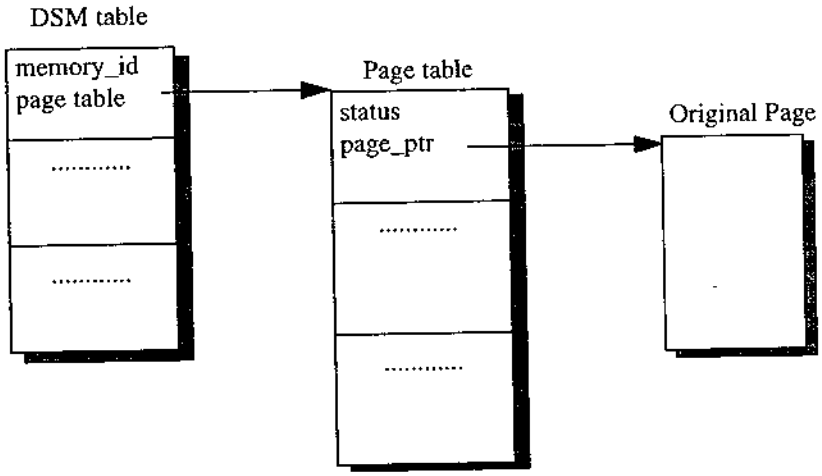

Figure 4.11 The DSM table as used for write-update DSM

manager owns the page; LOCKED, indicating that the local space manager owns the page and that it has been locked at that workstation for a period; or REMOTE, indicating that the page ownership is on a remote workstation.

- *version* — This field is an integer which is updated either when a page is

written to or a new copy of a page is brought to the workstation. The version number is attached to page request messages and is used to indicate whether the existing page is out of date and should be replaced.

- *probowner* — This field is the address of the Space Manager to which this workstation last granted ownership. Requests for a page are passed along the chain of probowners until the current owner is located.
- *request table* — This field hold page request messages. When a page is LOCKED, any incoming requests must be delayed to be handled later. The requests are held in a linked list to be handled in FIFO order when the page status is changed to OWNER.
- *copyset* — This field holds the pointer to a linked list of addresses. These addresses indicate the workstations which hold a readable copy of the page. The Space Manager that owns the page uses this list of page locations to invalidate all writable pages when granting write access to another Space Manager.

If the write-update protocol is being used the following fields in the DSM Table are employed:

- *status* — This field is overloaded in the two consistency models. In this case, the status can either be TWINNED or SINGLE, if twinned, a pointer (*page_ptr*) to the copy or twin of the page (original page). All page records are initialized to have a status of SINGLE, indicating that they have not been twinned.
- *page_ptr* — This field holds the pointer to a copy of the page which has been twinned. The original copy of the page is pointed to so that the two copies of the page can be compared when required to identify changes made to the page.

4.5.3 Write-Invalidate Based DSM in RHODOS

The design of the RHODOS' write-invalidate DSM system is based on the DSM system synthesised in Section 3.5.6.1. To show the design and behaviour of the RHODOS DSM system based on the use of sequential consistency employing the

write-invalidate coherence protocol a COW made up of three workstations labelled Source and Remote Workstations is used as shown in Figure 4.12. Three entities are

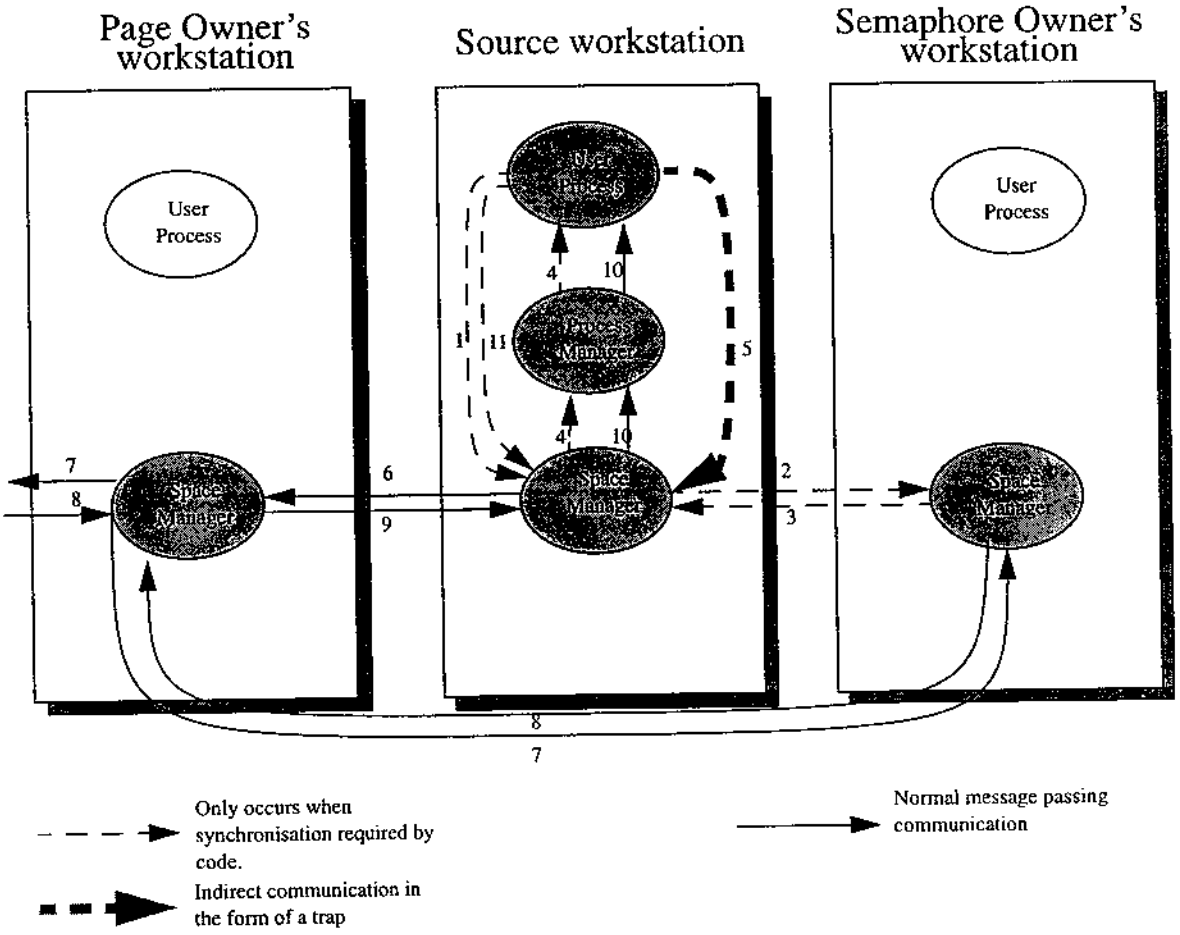

Figure 4.12 Design of write-invalidate based DSM on RHODOS

shown: a User Process; and the Space and Process Manager. Lines are used to depict messages being sent between the entities and the numbers indicate the message order. A description of the events that occur when the Producer code segment in Figure 4.9 is executed on the Source Workstation by the User Process, using write-invalidate based DSM [Silcock and Goscinski 97a], follows.

Before performing the write access on the shared variables a *wait*(*sem*[*0*]) call must be executed; the Producer Process sends a message (Message 1) to the Space Manager on the Source Workstation and blocks. The Space Manager sends a message (Message 2) to the probable owner of the semaphore. The probable owner is the

process which, according to this Space Manager's records, was the last owner of the semaphore. If ownership of the semaphore has been passed on, the message will be forwarded to the new probable owner until the actual owner is located. The semaphore can be in one of three states: REMOTE; LOCKED; or RELEASED. If it is REMOTE the local Space Manager is no longer the owner and the request must be forwarded to the probable owner of the semaphore as shown in the local Space Manager's records. If the semaphore is LOCKED the request is queued to be handled later when the semaphore is released. If the semaphore is RELEASED the state is changed to REMOTE and a reply (Message 3) is sent to the Space Manager on the Source Workstation. The latter changes the state of its semaphore to LOCKED and restarts the user process by sending a message (Message 4) to the Process Manager.

The page in the shared memory region, on which **x**, **y** and **z** are found, can be in one of three states:

- read-only — the only operation that may be performed on the page is a read operation; a write operation will cause an exception;
- read-write — both read and write operations can be carried out on the page; or
- offsite — a valid copy of the page does not exist in the memory of the workstation; a read or write operation on this page will result in an exception.

When the User Process tries to access memory which is not resident on that workstation (offsite) or attempts to write to a page which has read-only protections, a trap occurs and a message (Message 5) is sent to the Space Manager which uses an algorithm similar to that used to locate the semaphore owner to find the owner of the page. When the page owner has been located, the Space Manager on the Source Workstation sends a message (Message 6) to the Space Manager on the Page Owner's Workstation.

If write access is required and the page does not exist on the Source Workstation, an invalidation message (Message 7) is sent by the Space Manager on the Page Owner's Workstation to all of the workstations in the copyset. The copyset is a list of all workstations which have a copy of the page. The Space Managers on all

workstations in the *copyset* invalidate their copies of the page and send a reply (Message 8) to the Space Manager on the Page Owner's Workstation. When all the replies have been received a copy of the page is sent to the Source Workstation (Message 9) and the copy on the Owner's Workstation is invalidated.

If write access is required and the page does exist on the Source Workstation the page version number will accompany the request message (Message 6). As before an invalidation message (Message 7) is sent by the Space Manager on the Page Owner's Workstation to all of the workstations in the copyset. The Space Managers on all workstations in the copyset invalidate their copies of the page and send a reply (Message 8) to the Space Manager on the Page Owner's Workstation. When all the replies have been received the page version number in the request message is compared with the version number of the copy of the page on the Page Owner's Workstation. If the version number in the message is smaller than that of the local page a copy of the page is sent to the Source Workstation (Message 9) and the copy on the Owner's Workstation is invalidated. If the version numbers are the same the message is sent without the page itself since the page on the Source Workstation is not out of date.

If read access is requested, the Source Workstation is added to the owner's copyset and a copy of the requested page is sent to the Source Workstation (Message 9). The Space Manager on the Source Workstation receives the message from the Page Owner's Workstation.

When the page is received at the Source Workstation the Space Manager maps the page into the local space. If write access was requested it changes the ownership of the page to local and the protections on the page to read_write and the version number for the page is incremented. If the page itself did not accompany the message, i.e. the page on the Source Workstation was not out of date, the existing page is made read-writable and the version number for the page is incremented. The Space Manager then sends a message (Message 10) to the Process Manager which unblocks the User Process.

To exit the critical region, the User Process executes a *signal*(*sem*[*0*]) user library call and a message (Message 11) is sent to the Space Manager which changes

the status of the semaphore to RELEASED.

4.5.4 Write-Update Based DSM in RHODOS

The design of the RHODOS write-update based DSM system follows the DSM system synthesised in Section 3.5.6.2. To show the design and behaviour of the RHODOS DSM system based on the use of the release consistency model employing the write-update coherence protocol, a COW made up of three workstations labelled Source and Remote Workstations is used, as shown in Figure 4.13. Three entities are

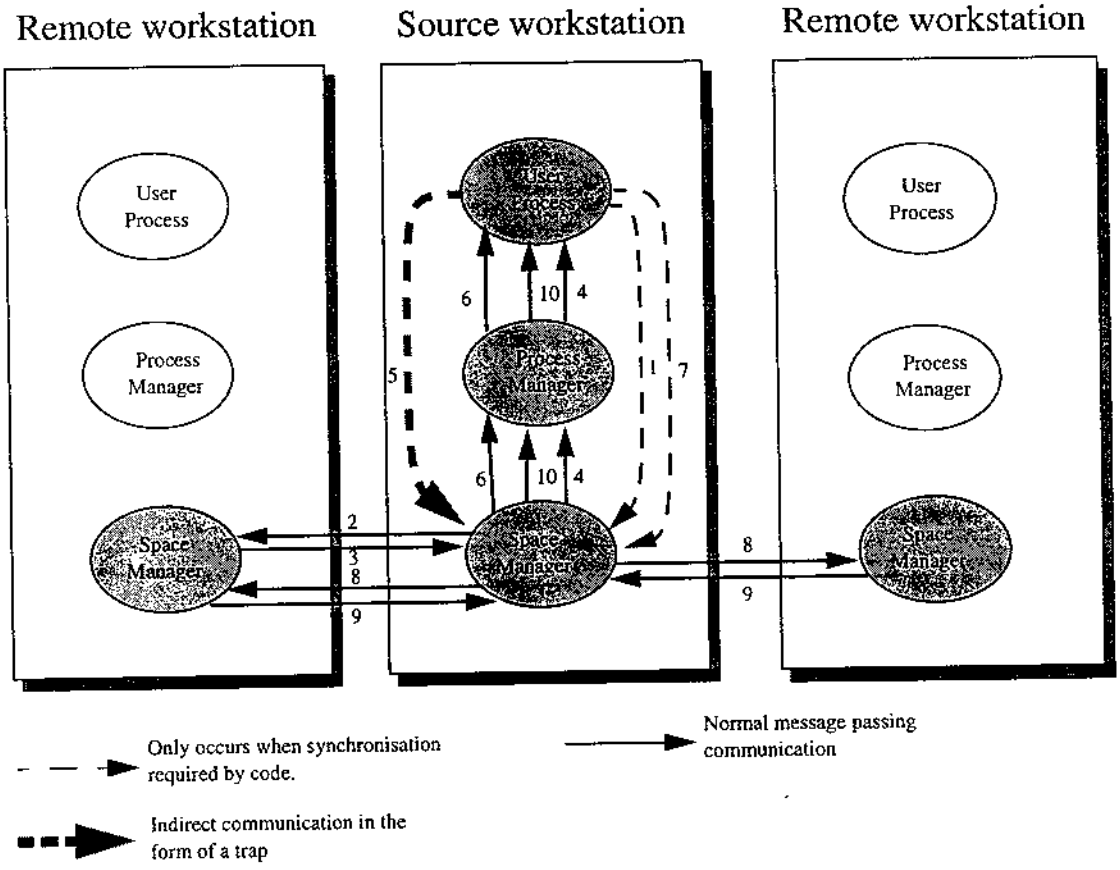

Figure 4.13 Logical Design of semaphores and memory consistency in write-update based DSM on RHODOS

shown: a User Process; and a Space and Process Manager. Lines are used to depict messages being sent between the entities and the numbers indicate the message order. The protections on the shared memory region, in which `x`, `y` and `z` are found, are read-only.

A description of the events that occur when the producer code segment in Figure 4.9 is executed by the User Processes, using write-update based DSM [Silcock and Goscinski 97b], on the Source Workstation follows.

In order to enter the critical section in which the write access on the shared variables will take place the *wait*(*sem*[*0*]) call must be executed, the Producer Process sends a message (Message 1) to the Space Manager on the Source Workstation and blocks. The Space Manager sends a message (Message 2) to the probable owner of the semaphore. If the semaphore is RELEASED the status is changed to REMOTE and a reply (Message 3) is sent to the Space Manager on the Source Workstation. The latter changes the state of its semaphore to LOCKED and restarts the user process by sending a message (Message 4) to the Process Manager.

The attempt to perform a write access on variable **x** in the shared memory region on the Source Workstation results in a write fault (Message 5) because the page has read-only protection. The write fault is handled by the Space Manager. The Space Manager twins the page and changes the protection for that page to read-write so that further accesses will not trigger a write fault. Twinning involves making an identical copy of the page. In the page record for the page containing **x** in the DSM table the status field is changed to TWINNED and a pointer to the copy of the page is placed in the *page_ptr* field. The Space Manager contacts (Message 6) the Process Manager to restart the User Process. The write operation can then proceed. If variables **y** and **z** are on the same page as **x**, the operations to write to them will also proceed without a page fault as the page now has read-write protections. Thus the page within the shared memory region is altered while the copy of the page pointed to by the DSM table remains unchanged. These two pages will be compared later to identify the changes.

When the User Process on the Source Workstation executes the *signal*(*sem*[*0*]) call to exit the critical section, Message 7 is sent to the Space Manager on the Source Workstation. In the *Diff Message*, shown in Figure 4.14 the shared memory region identifier is placed in *memory_id* and the number of pages containing differences (*num_pages*) is initialised to zero. The Space Manager searches sequentially through the page records of the shared memory region in the DSM table for page records with TWINNED status, as shown in Figure 4.15. When one is located num_pages in the

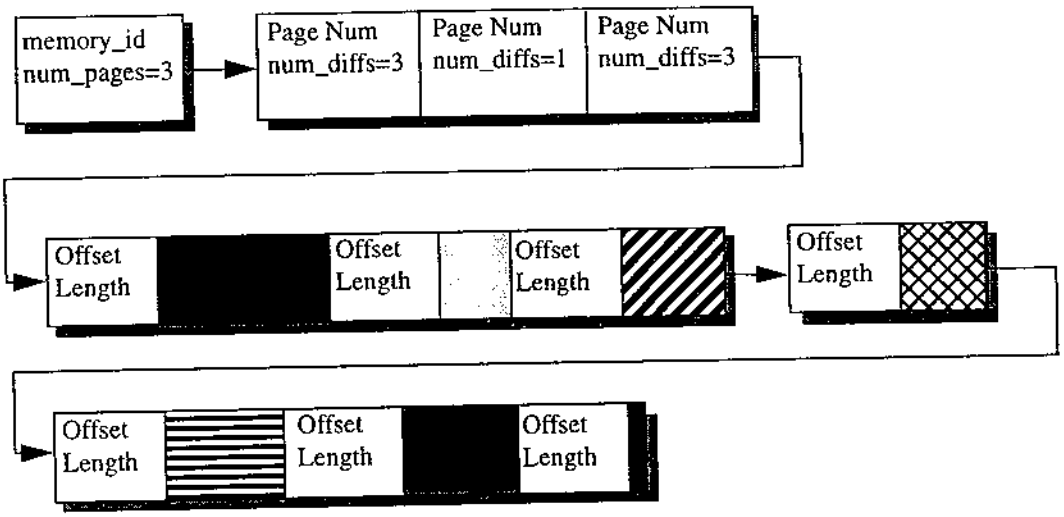

Figure 4.14 Diff Message Structure

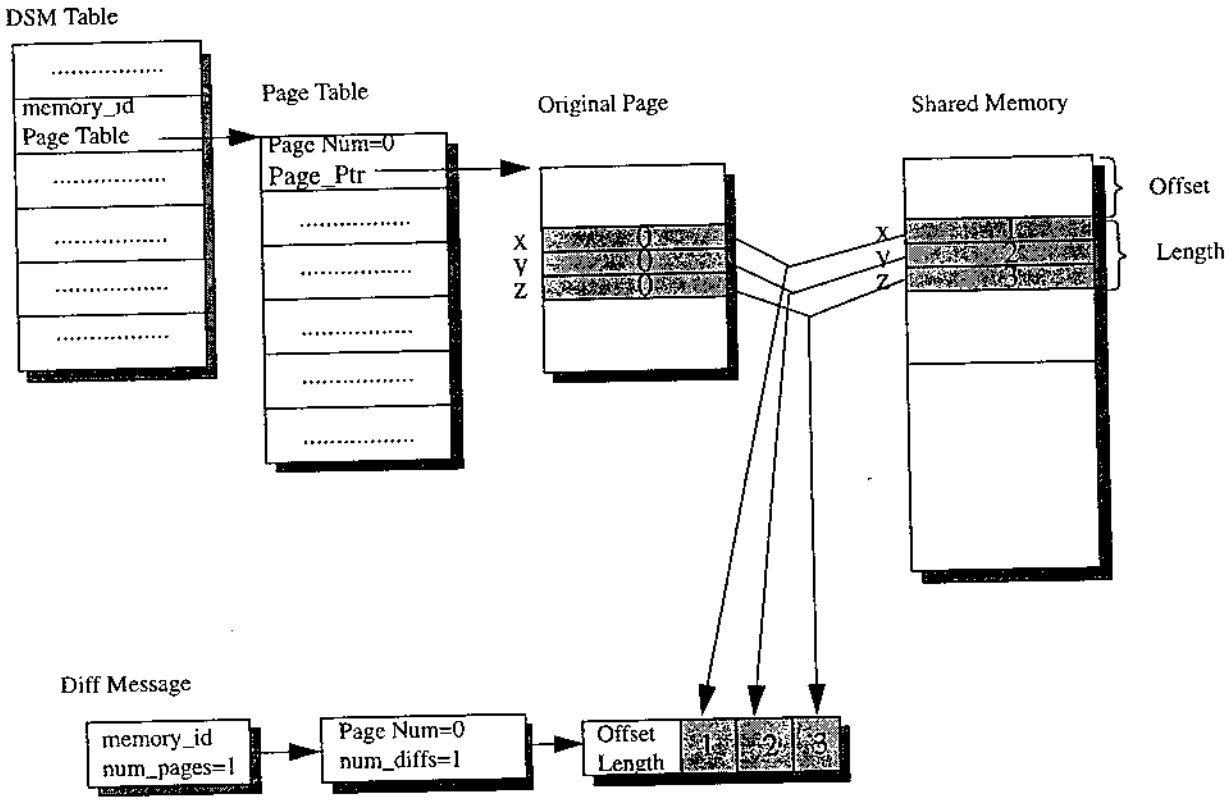

Figure 4.15 Overview of Diff Message generation

message is incremented and the original page, which is pointed to by the *page_ptr*, is

compared word by word with the corresponding page in the shared memory region. When a difference is found the offset of the beginning of the difference from the beginning of the page is stored in the *Diff Message* (*Offset*) (Message 8) and the number of differences found on the page (*num_diffs*) is incremented. The comparison continues until no difference is found when the length of the "chunk" of different data and the "chunk" itself are copied to the message. The search through the page for changes is continued to the end of the page and any further differences added to the message. The status field for the page in the DSM Table is changed to SINGLE. This process is repeated for each twinned page. When this is completed the Space Manager on the Source Workstation sends the *Diff message* (Message 8) to all Remote Workstations.

When the *Diff Message* is received at a Remote Workstation the shared memory region is updated to contain all the differences. When the message is received, the shared memory region with the same name as *memory_id* in the message is located. The page corresponding to *Page Num* in the message is then found and the address corresponding to *Offset* is located. The difference is then placed in the page at that position. This process is continued until the whole message has been consumed. The Space Managers on the Remote Workstations send acknowledgement messages (Message 9) to the Source Workstation. When the Space Manager on the Source Workstation has received all acknowledgements from the Remote Workstations it releases the semaphore and sends Message 10 to the Process Manager on the Source Workstation to restart the User Process. At this point all shared memory should be consistent. If, however, not all Remote Workstations acknowledge receipt of the *Diff Message* the Source Workstation will send the message again and await acknowledgements.

4.5.5 Semantics of barriers in write-invalidate and write-update based DSM in RHODOS

The design of the barriers used in the RHODOS DSM system is based on the DSM system synthesised in Section 3.5.5. Barriers are used in RHODOS to coordinate the DSM parallel processes. Processes block at a barrier until all processes have reached the same barrier, the processes then continue execution. In Figure 4.16 a piece

of skeleton code that demonstrates the use of barriers is shown.

```
initialisation code
dsm_barrier[barrier[0]]          /*This barrier ensures all
                                  processes have completed
                                  initialisation before the
                                  start of execution */
application code
dsm_barrier[barrier[0]]          /*This barrier ensures all
                                  processes have completed
                                  the first phase of the
                                  application code before
                                  the starting the next
                                  phase of execution */
application code
dsm_barrier[barrier[0]]          /*This barrier will ensure
                                  that all processes have
                                  completed execution before
                                  exiting*/
```

Figure 4.16 Code skeleton using barriers

In RHODOS barriers are managed by a centralized Barrier Manager. The Barrier Manager receives messages from all processes when they have reached the barrier. Once all these messages have been received the manager sends a message to all the processes allowing them to unblock and continue execution.

As in Figure 4.12 and Figure 4.13, Figure 4.17 and Figure 4.18 show a COW made up of three workstations labelled Source and Remote Workstations. Three entities are shown in each workstation: a User Process; and the Space and Process Managers. Lines are used to depict messages being sent between the entities and the numbers indicate the message order. A description of the events that occur when a *dsm_barrier*(*barrier*[0]) primitive is executed by the User Process on the Source Workstation follows.

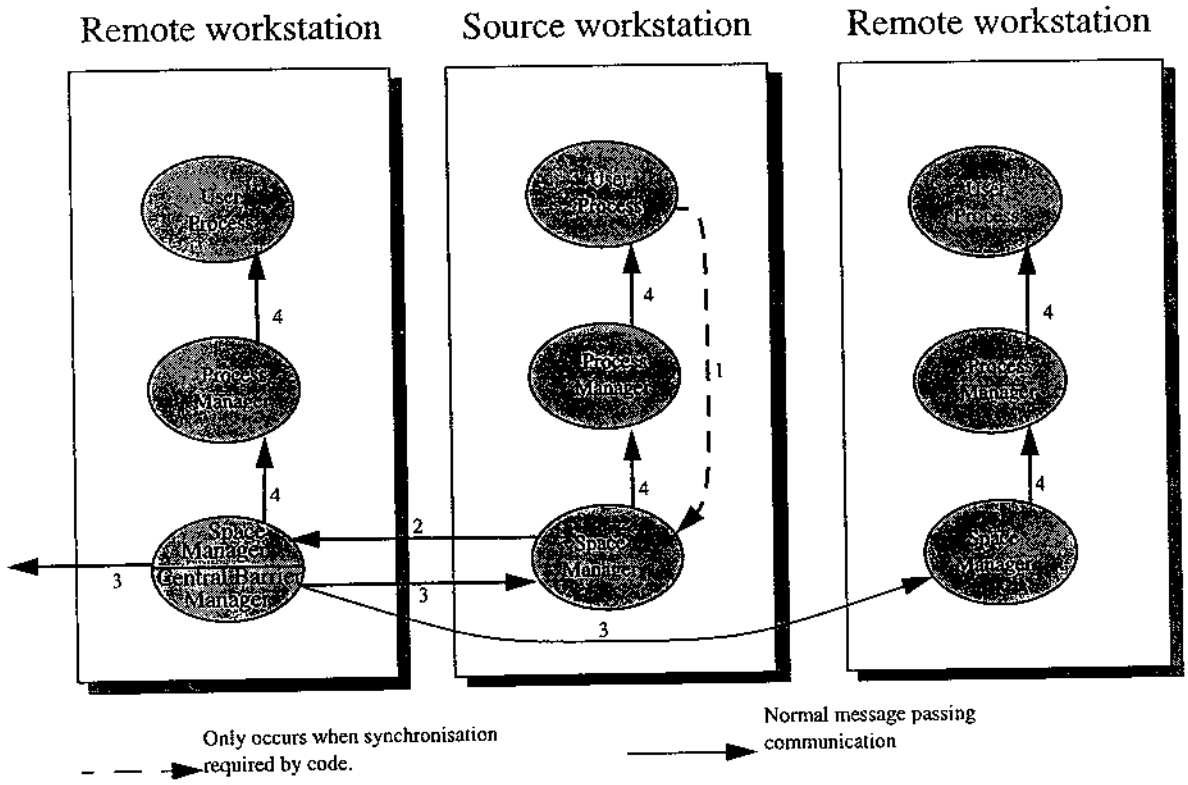

Figure 4.17 Logical Design of Barriers in write-invalidate based DSM on RHODOS

In write-invalidate based DSM when a *dsm_barrier(barrier[0])* call is executed (Figure 4.17) by the User Process on the Source Workstation, the User Process sends a message (Message 1) to the Space Manager on the Source Workstation and blocks. The Space Manager sends a message (Message 2) to the Central Barrier Manager. The Central Barrier Manager is part of the Space Manager on the workstation on which the barriers were initialised. The Central Barrier Manager increments the *barrier_count*. If *barrier_count* equals *barrier_max* the Barrier Manager has received barrier messages from all workstations listed in its barrier queue and it sends a message (Message 3) to all their Space Managers. The barrier_queue is a list of all workstations from which the Barrier Manager is expecting barrier messages. If the Barrier Manager times out waiting for barrier messages from all workstations in the barrier queue it can use the barrier queue to identify the workstation which has not sent a message. Currently there is no reaction to this situation but in the future the barrier manager will be made more reliable using this information. When all barrier messages are received the Space

Managers send a message (Message 4) to the Process Manager to restart the User Processes.

In RHODOS' write-update based DSM (Figure 4.18) barriers not only co-

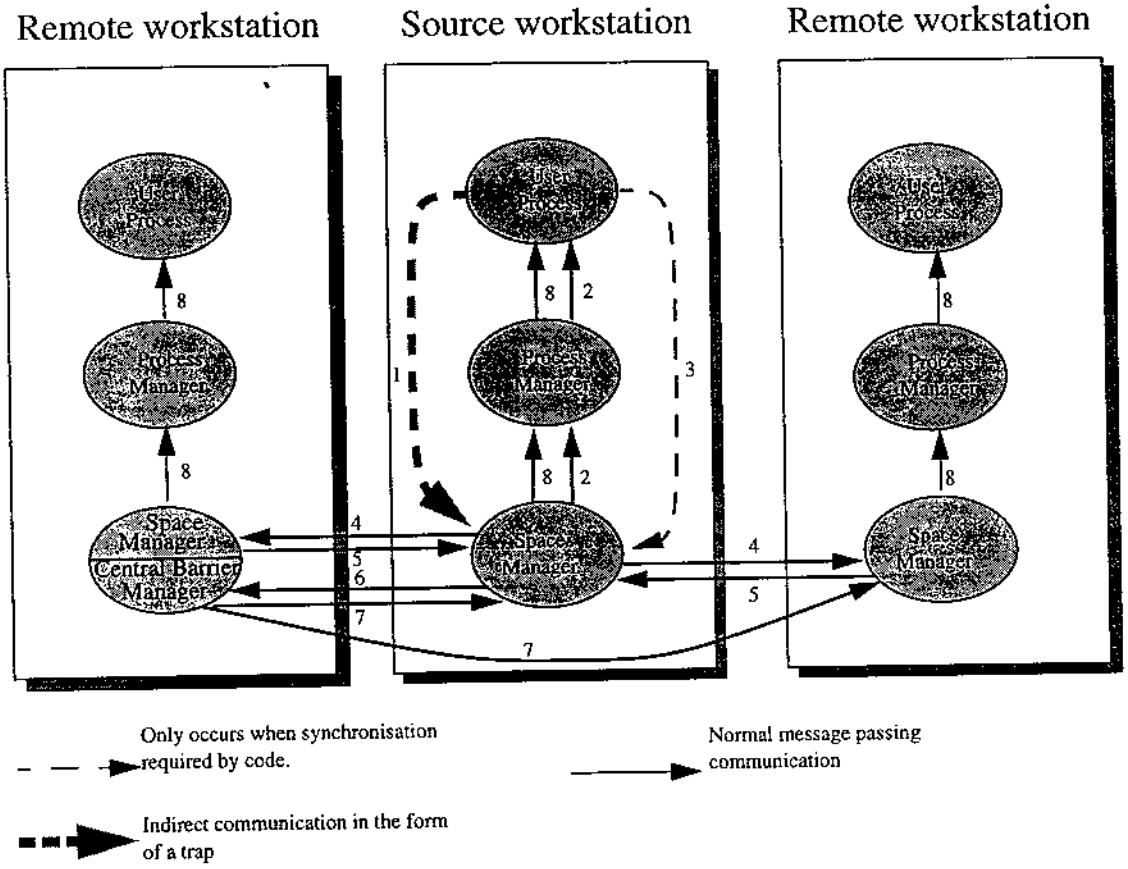

Figure 4.18 Logical Design of barriers in write-update based DSM on RHODOS

ordinate the processes but serve as points for memory update operations. Entry to a barrier has the same effect on the shared memory as a *signal*() because the first operation carried out by the barrier primitive is to make the shared memory consistent. Exiting from a barrier has the same effect as a *wait*() because the protection on the shared memory is changed to read-only.

Because the shared memory is made read-only before a barrier is exited, any subsequent attempt to perform a write access on any part of the shared memory after exiting a barrier results in a write fault (Message 1) which is handled by the Space

Manager. The Space Manager twins the page and changes the protection for that page to read-write so that further accesses will not trigger a write fault. Twinning involves making an identical copy of the page. The status field in the page record of the DSM table is changed to TWINNED and a pointer to the copy of the page is placed in the *page_ptr* field. The Space Manager contacts (Message 2) the Process Manager to restart the User Process. The write operation can then proceed.

When the User Process on the Source Workstation executes another *dsm_barrier(barrier[0])* call a message (Message 3) is sent to the Space Manager on the Source Workstation. The TWINNED pages are compared with their copies and a *Diff message* is generated using the same mechanism as that used to generate Diffs when a *signal*() is executed.

The *Diff Message* (Message 4) is then sent to the Space Managers on all Remote Workstations. The Space Managers incorporate the changes in the *Diff Message* into the shared memory. The Space Managers on the Remote Workstations send acknowledgement messages (Message 5) to the Space Manager on the Source Workstation. When the Space Manager has received all these acknowledgements from the Remote Workstations it sends a barrier message (Message 6) to the Barrier Manager. When the Barrier Manager has received barrier messages from all workstations in its barrier queue it sends a message (Message 7) to all their Space Managers. The Space Managers on all workstations then change the protection for the shared memory to read-only and send messages (Message 8) to their respective Process Managers to restart the User Processes.

4.6 Physically shared memory in RHODOS' write-update based DSM

In the RHODOS DSM system it is possible that the automatic initialisation may place more than one DSM process on a single workstation and for a DSM process to be migrated to a workstation on which a DSM process is already executing the same application. For both of these cases some mechanism is required for physical memory sharing.

In keeping with the DSM system synthesised in Chapter 3, physical memory

sharing in RHODOS' DSM has been designed to integrate transparently with the release consistent DSM system (Release consistent DSM was selected because it was found to perform better than sequentially consistent DSM [Silcock and Goscinski 97c]). Thus, in RHODOS' implementation of the DSM design from Chapter 3 the use of shared memory at application level is identical regardless of whether the memory is physically shared or distributed. The semantics of physically shared memory are shown in Figure 4.19. The figure shows two workstations, the Source and Remote

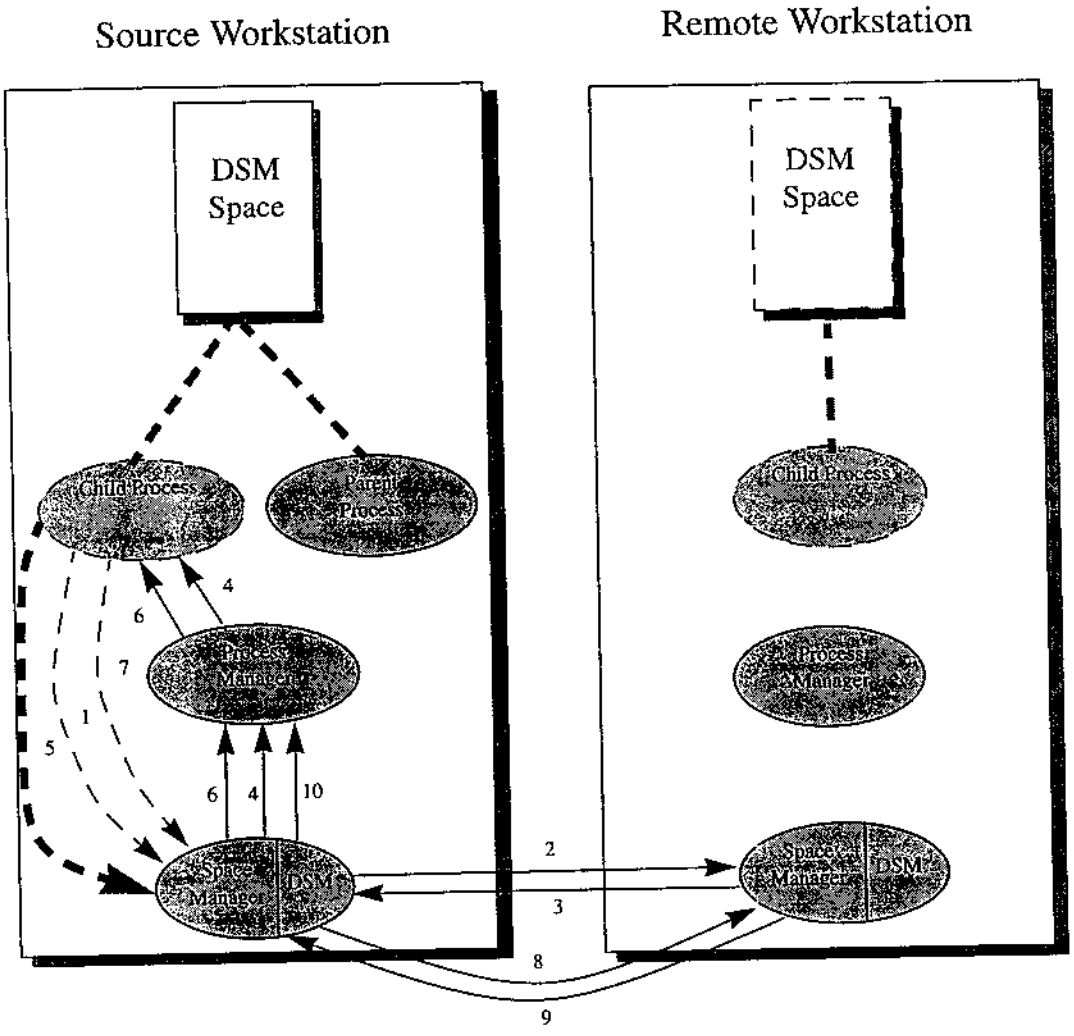

Figure 4.19 Semantics of Physically Shared Memory on RHODOS

Workstations. Depicted on these workstations are user processes, the Parent Process and one Child Process on the Source Workstation and a Child Process on the Remote Workstation, and the Process Manager and Space Managers. The DSM Space itself is

shown attached to the user processes.

Any RHODOS DSM Child Process which is starting invokes the *space_attach*() library call. The Space Manager searches for the a DSM space with the specified SName. If the Child Process is using the release consistency model and a DSM process is already executing the same application locally, the Space Manager finds that the DSM space already exists on the workstation and the new Child Process is simply attached to the existing DSM space. Thus, the DSM space is physically shared. The execution of applications using DSM is the same whether a child process is co-resident on a workstation with the parent or with another child process because once the initialisation phase is completed there is no difference between a child and the parent process. A description of the events that occur when the code segment in Figure 4.9 is executed by any two processes (in this case the Parent and Child Processes) which are physically sharing the DSM space, and using RHODOS' release consistent DSM [Silcock and Goscinski 97b] follows. The protections on the shared memory region, in which **x**, **y** and **z** are found, are read-only.

Before performing the write access on the shared variables a *wait*(*sem*[*0*]) call must be executed, the Child Process sends a message (Message 1) to the Space Manager on the Source Workstation and blocks. The Space Manager sends a message (Message 2) to the probable owner of the semaphore. If the semaphore is RELEASED the status is changed to REMOTE and a reply (Message 3) is sent to the Space Manager on the Source Workstation. The latter changes the state of its semaphore to LOCKED and restarts the user process by sending a message (Message 4) to the Process Manager. Because the semaphore's status is LOCKED, if the Parent Process attempts to enter the critical section it blocks and the request is queued in the same way a remote request is queued to be handled later when the Child Process has exited the critical section.

The attempt to perform a write access on variable **x** in the shared memory region on the Source Workstation results in a write fault (Message 5) because the page has read-only protection. The write fault is handled by the Space Manager. The Space Manager twins the page and changes the protection for that page to read-write so that further accesses will not trigger a write fault. The Space Manager contacts (Message 6)

the Process Manager to restart the User Process. The write operation can then proceed.

When the Child Process on the Source Workstation executes *signal*(*sem*[*0*]) call to exit the critical section, Message 7 is sent to the Space Manager on the Source Workstation. The Space Manager on the Source Workstation generates and sends the *Diff message* (Message 8) containing all changes made to the DSM space to all Remote Workstations. Since the local Child Process physically shares the DSM space with the Parent Process the updates are already visible to this process.

When the *Diff Message* is received at the Remote Workstation the shared memory region is updated. The Space Manager on the Remote Workstation sends an acknowledgement message (Message 9) to the Source Workstation. When the Space Manager on the Source Workstation has received all acknowledgements from the Remote Workstations it releases the semaphore and sends Message 10 to the Process Manager on the Source Workstation to restart the Child Process. At this point all shared memory should be consistent.

4.7 Conclusions

The goal of this chapter has been to demonstrate that the new DSM system proposed in Chapter 3 is entirely feasible and to validate the design decisions and synthesis of the proposed DSM system. In this chapter the development of DSM completely integrated into the memory management function of the RHODOS, microkernel and client-server based distributed operating system, has been presented.

The development of a transparent DSM system within RHODOS has demonstrated the feasibility of the proposed DSM system. Furthermore, the choice of a client-server and microkernel based distributed operating system has been substantiated — the RHODOS operating system is a microkernel and client server based distributed operating system which is flexible enough to provide the optimal environment for the DSM system as described in Chapter 3. Thus, the RHODOS operating system and its components have been identified as providing an appropriate environment for the development of the proposed DSM system. The Space Manager, which provides the memory management in RHODOS, has easily been adapted to incorporate the DSM functionality in a transparent manner. Furthermore, the DSM

system is able to use the existing interprocess communication facility as the basis for the DSM communication. The development of the write-invalidate and write-update based versions of the DSM system and their relationship with the semaphore-type synchronisation are shown to be feasible and to integrate well with the existing Space Management functions of RHODOS. This semaphore-type synchronisation gives a generic synchronisation mechanism which is easy to use and familiar to programmers. In addition, the distributed semaphore management improves the scalability of the system.

The automatic initialisation of the DSM processes shows that this DSM design provides users with a convenient and easy to use environment. The use of the RHODOS' Global Scheduler and REX Manager to provide the automatic initialisation facility shows that this operating system truly meets one of the operating system aims of user convenience. The physically shared memory in RHODOS displays that distributed and physically shared memory are employed by users in exactly the same manner.

It is shown that the design requirements of ease of programming, ease of use, transparency and scalability are all met by the RHODOS DSM implementation. The ease of programming requirement is met by the semaphore type synchronisation which is familiar to programmers who have written any shared memory code. The barriers are easy to use and require no specialised knowledge. Automatic initialisation provides ease of use because it allows programmers to insert a simple library call to initialise parallel execution on a number of workstations. The transparency requirement is met by the lack of specialised DSM related programming constructs other than the barrier which is kept as simple as possible. The use of dynamic distributed semaphore management makes the system scalable by not causing a bottleneck at a centralised manager.

Chapter 5 Programming and Performance Studies of RHODOS DSM

5.1 Introduction

The aim of this research is the synthesis and development of a distributed operating system integrated DSM system, such as that it should provide application programmers with an easy to use development and execution environment and should be both transparent and efficient. In the previous chapter the development of the DSM system within the memory management of the RHODOS distributed operating system was described. The development was carried out in order to validate the claims that operating system integrated DSM is feasible, transparent, easy to program and to use.

The goal of this chapter is to experimentally validate the claim that the DSM system proposed and developed in Chapters 3 and 4 not only forms an easy to use and transparent environment for the development and execution of parallel applications but also results in the efficient execution of these parallel applications.

In order to demonstrate that the goal has been achieved this chapter has been structured in the following manner. Firstly, in order to show that the DSM system meets the ease of programming and efficiency requirements specified in Chapter 3, the programming aspects and results of performance tests carried out on a set of test applications together with a discussion of the results for these tests are presented. Secondly, to illustrate the ease of use and transparency of the DSM system, the results of qualitative tests carried out and experience gained from allocating a moderately inexperienced undergraduate programmer the task of writing code for some test applications are given. The performance tests were carried out with a single process executing on each workstation. Furthermore, the results for experiments carried out using both distributed and physically shared memory, i.e. with more than one DSM parallel process executing on one of the workstations are given. The fact that these two systems are indistinguishable provides further evidence that the DSM system is fully

transparent.

The five test applications used for the experiments were chosen because they were a mixture of SPMD and MPMD computations and had been used by other DSM researchers allowing a comparative evaluation to be performed. The applications are: Red-Black Successive Over Relaxation (SOR), Matrix Multiplication, Jacobi, Quicksort and Travelling Salesman Problem (TSP).

5.2 Programming Aspects and Performance Studies

The objective of this section is to describe the experiments carried out on five applications with a single DSM parallel process executing on each workstation. In particular the programming aspects of RHODOS' DSM are highlighted, showing how this DSM system supports application programmers by allowing them to program using the same shared memory code that they would use when programming for a physically shared memory system. In conjunction with the programming aspects, the results for the performance tests presented here demonstrate that the DSM system is an efficient means of executing parallel applications.

5.2.1 Hardware Platform

The DSM system is currently implemented within the RHODOS operating system running on a cluster of eight Sun 3/50 workstations. The COW is connected by a 10-Mbps Ethernet. The granularity of the shared memory in RHODOS DSM is an 8K page. The experiments were carried out using one up to eight workstations for each application for both the write-update implementation, which is used to implement release consistency, and the write-invalidate implementation, which is used to implement sequential consistency, in RHODOS' DSM system.

5.2.2 Test Applications

A qualitative assessment of the easiness of programming is carried out based on algorithms and code of individual applications, paying attention to differences between physically shared memory and RHODOS DSM algorithms and code.

The performance is measured as the speedup of an application, where the

speedup is defined as $S_p = T_1/T_p$. T_1 is the time taken to execute the application on a single processor and T_p is the time taken to execute the same algorithm on p processors [Gropp 92]. In the tests the execution time for p processors from 1 to 8 was measured.

The applications used were:

- Red-black Successive Over-Relaxation (SOR) which exploits a Single Program Multiple Data (SPMD) computational model. The algorithm uses a large matrix to solve a partial differential equation. The matrix is divided between a group of identical processes; each process computes its own section of the matrix. The boundary values on the edges of the sections are the only parts of the matrix that are shared.
- Matrix Multiplication, which also uses a SPMD computational model in which two matrices are multiplied and the results placed in a third matrix. The matrices are split between the processes, with each process computing its portion of the result matrix, since there are no shared boundary values there is virtually no data sharing during the computation.
- Jacobi, which like SOR, uses a SPMD computational model to solve partial differential equations. The difference between the two problems is that the Jacobi algorithm uses a scratch array to hold results to prevent the old values being overwritten before they are used.
- Quicksort (QS) which has computational phases in which it uses the SPMD model and phases in which it uses the MPMD model. The algorithm sorts an array of elements. The array is continually sub-divided and shuffled until the size of each sub-divided array reaches some threshold value at which point the array is sorted by one process.
- Travelling Salesman Problem (TSP) which uses a Multiple Program Multiple Data (MPMD) computational model. TSP is based upon a branch and bound algorithm. In TSP all processes share the same queue of partial tours; there is no data decomposition.

The programming aspects, in particular easiness of programming, of each application are summarised by two phases of each algorithm: the initialisation phase which mainly addresses parallelism management, and the application phase. The

"`initialisation`" phase for each of the computation closely follows the description given in Section 4.4.3 and code given in Figures 4.7 and 4.8. This phase is very similar in each case and will not be discussed here any further.

5.2.3 Red-Black Successive Over Relaxation (SOR)

The Successive Over-Relaxation (SOR) algorithm is used to solve partial differential equations. The red-black version of this algorithm uses a 2-dimensional matrix, "chequered" with every second element marked red or black like a chess board. Each element in the matrix contains a single floating point number. In the algorithm the average of the surrounding (above, below, left and right) red neighbours of each black element are placed into that black element. The average of the surrounding black neighbours of each red element are then placed into the red element. One row is computed at a time, starting from the top of the matrix moving downwards. To parallelise this algorithm, the matrix is separated into N approximately equal blocks of data made up of contiguous rows, where N is the number of processes. The only common data that is accessed are the common boundaries between the blocks of data.

To implement this algorithm code supplied by Peter Keleher [Keleher 96] was modified. The pseudocode is shown in Figure 5.1. This code implements the red-black matrix using two separate 2-dimensional arrays, referred to as *red* and *black*. The matrix size used for the write-invalidate based DSM tests is 64 × 2048. This size ensures that each row occupies exactly one page (8kb), eliminating false sharing, and allowing each phase to cause only one page fault for each iteration. Due to memory limitations, a larger number of rows was not possible. For the testing of the write-update based DSM a 128 × 128 array is used.

The matrix is implemented as one piece of memory, shared by all processors (using write-invalidate based DSM and write-update based DSM). Barriers are used to synchronise each phase. In the first phase each process writes to its section of the black array, then in the second phase, each process writes to its portion of the red array. This implementation also means that there is no need for semaphores since the processes only access their portion of the matrix.

To achieve parallelism, the arrays are split up into contiguous blocks of rows,

```
initialisation
dsm_barrier(barrier[0])          /*All processes have com-
                                   pleted initialisation*/
for X iterations
    for (j = begin to end)
        for (k = first_column to last_column)
            black[j][k] = (red[j-1][k] + red[j+1][k] +
                           red[j][k-1] + red[j][k]) / 4
        j = j + 1
        if (j is > end)
            break from j loop
        for (k = first_column to last_column)
            black[j][k] = (red[j-1][k] + red[j+1][k] +
                           red[j][k] + red[j][k+1]) / 4

    dsm_barrier(barrier[0])      /*Ensures all processes have
                                   calculated "black" values
                                   before calculating "red"
                                   values*/

    for (j = begin to end)
        for (k = first_column to last_column)
            red[j][k] = (black[j-1][k] + black[j+1][k] +
                         black[j][k-1] + black[j][k]) / 4
        j = j + 1;
        if (j > end)
            break from j loop
        for (k = first_column to last_column)
            red[j][k] = (black[j-1][k] + black[j+1][k] +
                         black[j][k] + black[j][k+1]) / 4

    dsm_barrier(barrier[0])      /*Ensures that all processes
                                   have calculated "red" values
                                   before starting next itera-
                                   tion*/

dsm_barrier(barrier[0])          /*Ensure that all processes
                                   have completed execution
                                   before exiting*/
```

Figure 5.1 Pseudocode of the SOR algorithm

and each process is given one block on which to operate. Therefore, for N processes, each process will get approximately 1/Nth of the arrays. A unique identification number is given to each process to determine its piece of the arrays. The algorithm

(Figure 5.2) shows the decomposition algorithm, where M is the unique identifier for the process.

```
chunk_size = M / number_of_processes
remainder = M%number_of_processes;
if (process_number < remainder)
{
    first_row = process_number * (chunk_size + 1);
    last_row = first_row + chunk_size;
}
else
{
    first_row = (chunk_size * process_number)+remainder
    last_row = first_row + (chunk_size - 1);
}
```

Figure 5.2 Data decomposition algorithm

The results for write-invalidate based DSM Red-Black SOR on a 64 × 2048 matrix and write-update based DSM Red-Black SOR on a 128 × 128 matrix are shown in Figures 5.3 and 5.3; the speedup for 8 processors are 5.6 and 4.3, respectively.

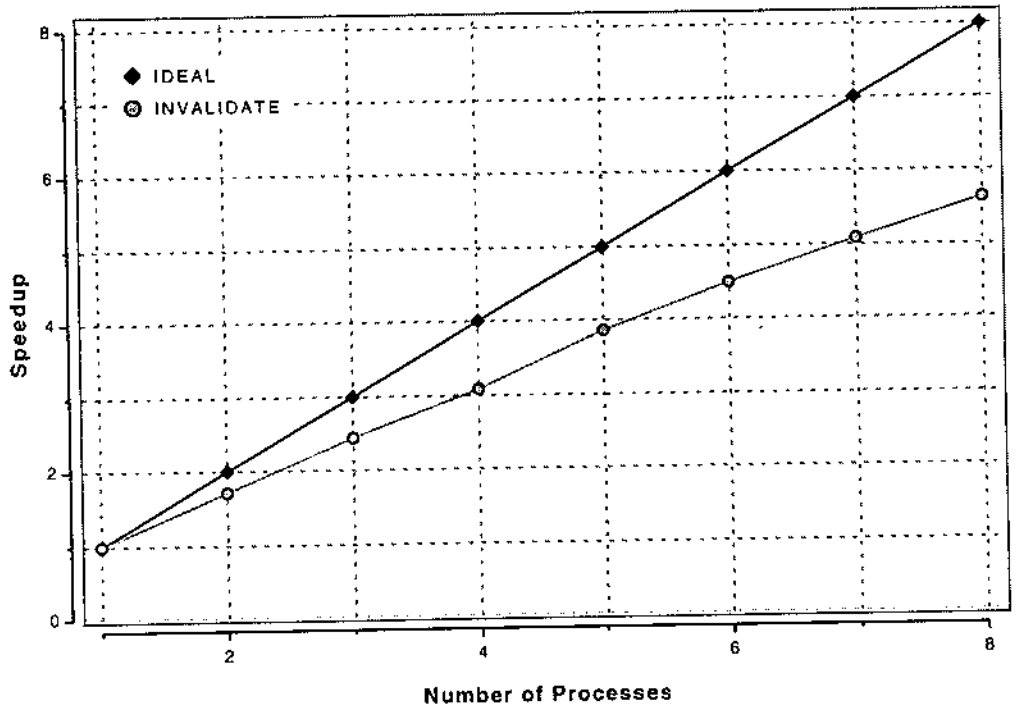

Figure 5.3 Speedup for SOR using RHODOS' write-invalidate DSM on a matrix of 64×2048 elements

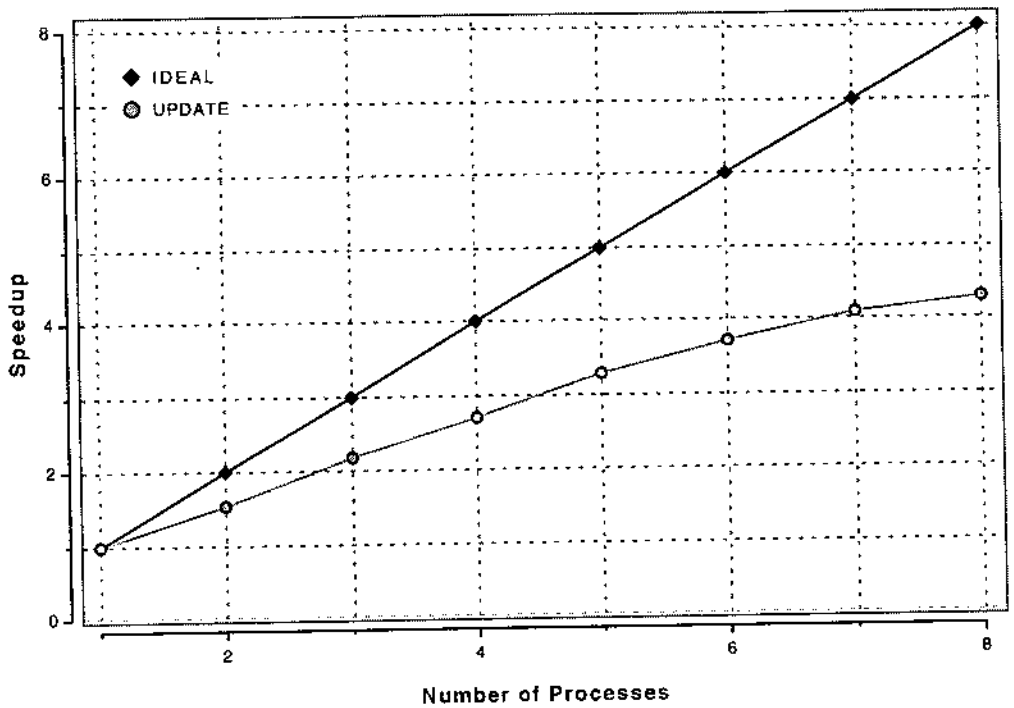

Figure 5.4 Speedup for SOR using RHODOS' write-update based DSM on matrix of 128×128 elements

Discussion: The programmer is required to initialise the DSM, using the automatic initialisation code shown in Figures 4.7 and 4.8, but thereafter can write the program using normal shared memory techniques without any concern for the fact that DSM is being used. The initialisation code includes a single parameter which determines whether write-update or write-invalidate based DSM is being used. Barriers at the start and end of the SOR code ensure that the initialisation phase is finished and that all processes have completed executing before any of the processes exit. The barrier during the execution is required as the processes access boundary rows in adjacent sections of the matrix. When calculating the *red* values for the boundary row each process requires the *black* values calculated by another process. The barrier ensures that all processes have completed the calculation of the *black* values before the computation of any *red* values commences.

It must be noted that in this experiment the dimensions chosen for the array used in the write-invalidate based DSM were deliberately chosen to reduce false sharing. A decision was made to use a matrix with page-size rows in order to allow the comparison of the results for RHODOS DSM with those achieved by other researchers

using page-based implementations. In [Carter et al. 95b] rows which are multiples of page-size reduce false sharing, while in [Lu 95a] page-size rows reduce the number of page faults at the beginning of each phase. Realistically, such implementation details should not concern the programmer and in a normal situation, there may be more false sharing than in this experiment. A larger matrix for the write-update based DSM would certainly have shown an improvement in performance.

5.2.4 Matrix Multiplication

The algorithm (Figure 5.5) performs the multiplication (cross product) of two

```
initialisation
dsm_barrier(barrier[0])          /*This barrier ensures all
                                  processes have completed ini-
                                  tialisation before the start
                                  of execution */

    for row = 0 to N {
        for col = 0 to N {
            sum = 0
            for i = 0 to N {
                sum = sum + input1[row][i] + input2[i][col]
            }
            output[row][col] = sum
        }
    }

dsm_barrier(barrier[0])          /*This barrier synchronises the
                                  processes to ensure that they
                                  end only after the other
                                  processes have completed*/
```

Figure 5.5 Pseudocode for Matrix Multiplication

input matrices putting the result into an output matrix. The main loop of this code was originally used for tests on Munin [Carter et al. 95b], and was modified by us to suit RHODOS' DSM. The three matrices, *input1*, *input2*, and *output* were implemented as three shared 2-dimensional arrays of integers, all of size 256 × 256. The two input arrays contained a set of random integers from 0 to 10.

Discussion: The method used to parallelise this algorithm is very similar to that used for SOR in that each process is allocated a section of each matrix to work on using the same data partitioning algorithm shown in Figure 5.2. However, there are no shared

boundary values in this algorithm as there are in SOR. Each process reads from its section of the input matrices and writes to its section of the output matrix. Upon completion, the resulting matrix may be accessed by any process. Due to the nature of matrix multiplication, no processes need to access the same elements of data, nor do they rely on each others results. Therefore, no barriers are needed during computation, only a barrier at the start and finish of computation. The latter ensure that all processes have completed initialisation and execution respectively before continuing execution and exiting.

In Figure 5.6 the speedup for matrix multiplication using write-invalidate based

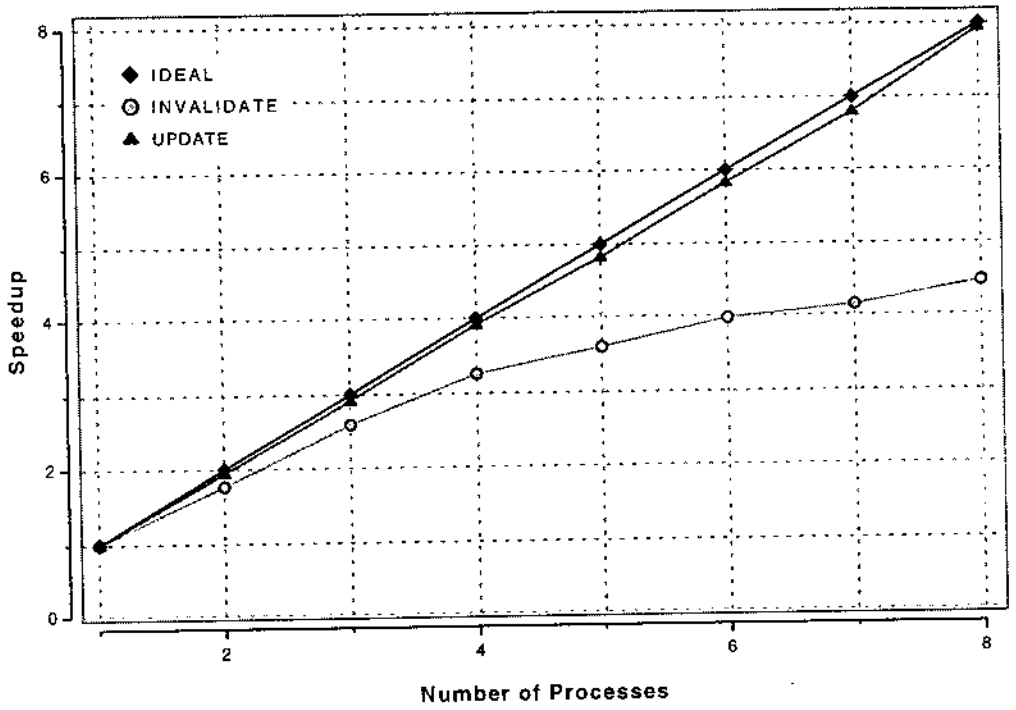

Figure 5.6 Speedup for Matrix Multiplication using write-invalidate and write-update based DSM on a matrix of 256×256 elements

DSM and write-update based DSM, compared to the ideal or linear speedup for 1 up to 8 workstations is shown. For the write-invalidate based DSM the speedup starts to drop away when 5 processors are used. This may be due to the small matrix size (256 × 256); each processor performs a small amount of computation compared to the amount of communication required to transfer the pages to the workstation on which they are required. (A larger matrix was not possible due to memory restrictions.) The speedup

for 8 processors is 4.5.

For the write-update based DSM the 8 processor speedup is 7.9. The speedup shown here is very close to ideal. This is because this application uses a SPMD computational model with no data sharing. The only time the memory is updated is at the barrier at the end of computation. This differs from the other SPMD model-based computations, Red-Black SOR and Jacobi, where the boundary values are shared and the memory must be updated during computation.

5.2.5 Jacobi

Much like SOR, the Jacobi algorithm is used to solve partial differential equations. It uses 2 matrices, *grid* and *scratch*, placing the average of the surrounding (above, left, right, below) elements of the *grid* array into the *scratch* array, and then places the *scratch* array into the *grid* array. This is performed an arbitrary number of times. The code was based on an algorithm in [Amza et al. 96]. The pseudocode for the Jacobi algorithm is shown in Figure 5.7.

One shared 2-dimensional array of floats is used for the *grid* array, and a local 2-dimensional array of floats is given to each process for the *scratch* array. For the write-invalidate based version, a size of 64 $\times$ 2048 was used, because as in SOR this size ensures that each row occupies exactly one page (8kb). By doing this, false sharing is eliminated, thus, each phase causes only one page fault for each iteration. However, due to the fact that the write-update based version requires more memory to store copies of pages while they are being updated, a matrix size of 60 $\times$ 1024 was used. The same method is applied to achieve parallelism, i.e. allocating equal sections of the matrix to each process to work on using the same data partitioning algorithm shown in Figure 5.2. As in SOR the boundary values of these sections of the array are shared with other process. The programmer is required to insert barriers to synchronise the Jacobi processes. As in all cases pre- and post-execution barriers ensure the processes all commence execution only after the completion of initialisation and exit only after all processes have completed execution. The barriers during execution ensure that all processes have completed the computation of their scratch arrays before they are written to the grid arrays.

```
initialisation
for a matrix of size DOWN × ACROSS

dsm_barrier(barrier[0])          /*Ensures initialisation com-
                                   pleted*/

for current = 0 to ITERATIONS{
    for i = 1 to DOWN
            for j = 1 to ACROSS
               scratch[i][j] = (grid[i-1][j] + grid[i+1][j] +
                                grid[i][j-1] + grid[i][j+1])/4

dsm_barrier(barrier[0])          /*Ensures computation of scratch
                                   arrays have been completed
                                   before scratch arrays are writ-
                                   ten to grid arrays */
    for i = 1 to DOWN
        for j = 1 to ACROSS
            grid[i][j] = scratch[i][j]

dsm_barrier(barrier[0])          /*Ensures all processes have
                                   updated grid array before com-
                                   mencing next iteration  */
}

dsm_barrier(barrier[0])          /*Synchronises processes to
                                   ensure that they end only after
                                   the other processes have com-
                                   pleted*/
```

Figure 5.7 Pseudocode for the Jacobi algorithm

Discussion: The results for the Jacobi tests are shown in Figure 5.8. The speedup for write-invalidate (64X2048 matrix) and write-update (60X1024 matrix) on 8 processors are 5.4 and 5.2 respectively. The write-invalidate based DSM appears to perform better in this case however the results are due to the manipulating of the matrix size to make each row exactly page size reducing false sharing. The tests were repeated using write-invalidate based DSM this time with a matrix size of 60 × 1024. The results for the latter test are also shown in Figure 5.8. The speedup for 8 processors in this case is 3.6. This result shows the impact that false sharing has on the performance of the write-invalidate based DSM systems.

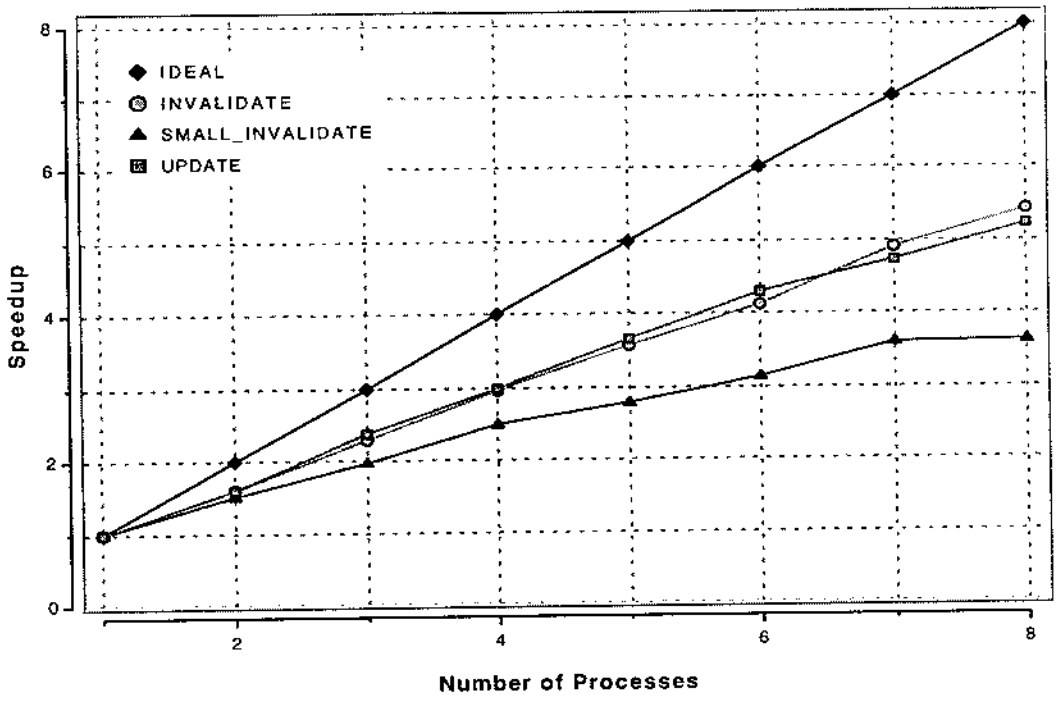

Figure 5.8 Speedup for Jacobi on matrices of 64X2048 (INVALIDATE) and 60X1024 (SMALL-INVALIDATE and UPDATE) elements

5.2.6 Quicksort

Quicksort is a recursive sorting algorithm which repeatedly divides lists into sublists so that the contents of one list are less than the contents of another. When the size of the sublists reaches some default size (1KB in this case) they are sorted sequentially using a bubblesort algorithm. The Quicksort code used here was provided by Keleher [Keleher 96]; some adjustments were made to the code for use on RHODOS DSM. The pseudocode for Quicksort is shown in Figure 5.9.

Quicksort was implemented using the three shared data structures: the array being sorted; the task queue containing the indices of the subarrays to be sorted; and a count of the number of processes waiting for work. The task queue serves to parallelise Quicksort because it holds the details of the unsorted sublists. The processes continually remove these details and either partition the sublists or, if they are of the default size, sort them sequentially. Partitioning a list involves dividing it into two where the contents of one of the sublists created are strictly greater than the contents of the other. The programmer must insert a semaphore to protect accesses to the task

```
initialisation
dsm_barrier(barrier[0])          /*Ensures all processes have
                                   completed initialisation
                                   before the start of execution*/
while (true){

    wait(sem[0])                 /*Entry to critical region guar-
                                   antees exclusive access to task
                                   queue */

        if (Number of tasks waiting = Number of Processes)
            signal(sem[0])       /*Exits critical region*/
            quit()
        else
            signal(sem[0])       /*Exits critical region*/
            wait for a new task
            task = new_task
    if (new_task <= threshold)
        bubblesort new_task
    else
        Select a pivot for new_task
        Partition the elements around the pivot so that
        those elements before the pivot are smaller than the
        pivot and those after the pivot are greater than the
                                   pivot

    dsm_barrier(barrier[0])      /*Ensures all processes have
                                   completed execution before
                                   exiting*/
}
```

Figure 5.9 Pseudocode for Quicksort

queue. As in all applications barriers are used to synchronise the start and finish of execution.

The sequential bubblesort performed on the sublists with the default size are carried out without synchronisation as each process is sorting a section of the array which will only be accessed by that process. The default size in this experiment was set at 1024 elements.

Discussion. Quicksort displays false sharing when more than one process attempts to sort different subarrays which happen to be on the same page. This means that the pages thrash between workstations. The speedups for Quicksort using write-

invalidate and write-update based DSM are shown in Figure 5.10. The speedup for 8

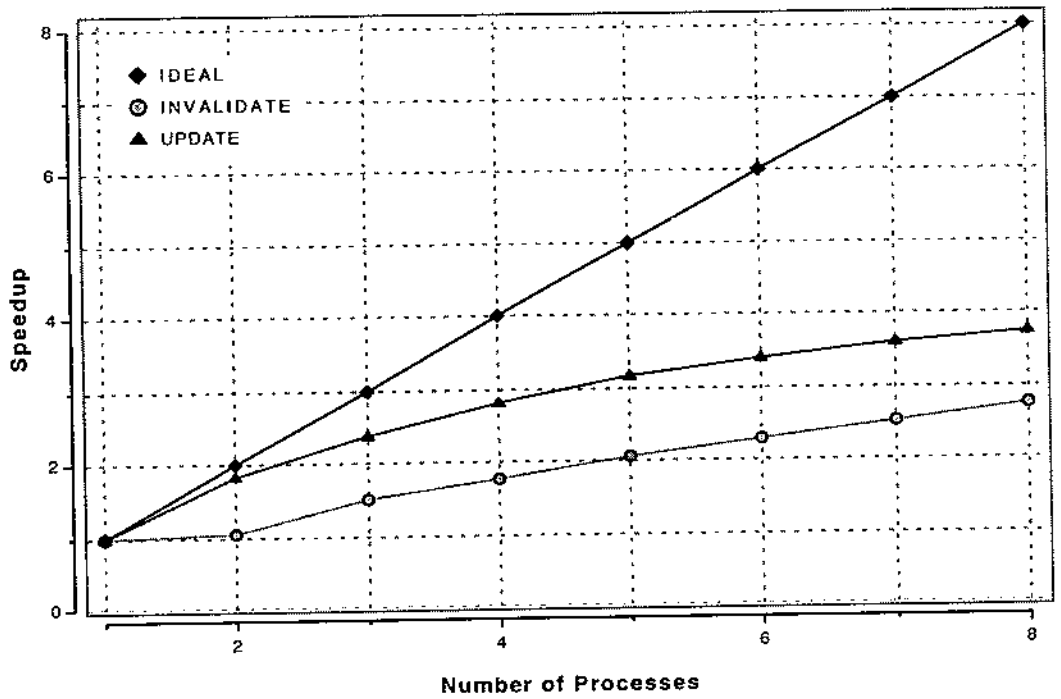

Figure 5.10 Speedup for Quicksort using invalidation and update-based DSM on an array of 256K elements

processes are 2.8 and 3.8, respectively. The speedup drops away from the *IDEAL* speedup because of the large number of synchronisation accesses performed. The task queue is always accessed in a critical region and whenever a task is removed from the task queue the critical region is exited. However, if the task removed requires further subdivision the critical section must be re-entered after the subdivision is performed so that the new tasks can be added to the queue and the critical section exited. Each time a critical section is entered messages to gain access to the synchronisation variable must be sent. In the case of the write-update based implementation exiting the critical section causes the operating system to go through the process of updating the shared memory.

5.2.7 Travelling Salesman Problem (TSP)

The Travelling Salesman has a group of cities to visit. The problem is to find the shortest path which the salesman must take in order to visit each city only once. Each path is known as a *tour*. This algorithm maintains a *priority queue* which contains partially evaluated tours. The head element of the queue is the tour with the smallest

lower bound for the remaining portion of the tour. If the remaining number of cities required to complete the tour is below the threshold, n, the remainder of the tour is computed sequentially, otherwise, the partial tour is expanded by a single city and the resulting partial tours are put into the tour queue. The current best tour is maintained. As each tour is removed from the tour queue a lower bound on the remaining part of the tour is computed. If the sum of this lower bound and the current length exceeds the current best tour the tour is rejected. The code for the TSP problem was originally from [Keleher 96]. It was adapted it for RHODOS DSM.

The input for TSP is an array which represents the distances between the cities which the salesman must visit. The shared data structures are:

- the length of the global minimum tour and the tour itself;
- an array containing partially evaluated tours and unused tours;
- a priority queue of pointers to partially evaluated tours; and
- the tour stack the elements of which point to unused tour structures.

Effectively, the larger the threshold, the larger the portion of the work to be completed sequentially. These sequential portions are then carried out in parallel. In this case, a threshold of 13 was used which in [Carter et al. 95b] is referred to as a coarse-grained solution and results in better speedups than the finer grained solutions, which use a smaller threshold. The pseudocode for TSP is shown in Figure 5.11.

Discussion: The major bottleneck in the TSP code is that to access the priority queue. Processes must wait for the semaphore before accessing the queue itself. This semaphore is not released until new tours have been put back onto the queue. An 18-city tour was used. The speedup shown by RHODOS' write-invalidate and write-update based DSM for TSP are shown in Figure 5.12. As can be seen in this figure the 8-processor speedups are 4.2 and 6.9, respectively.

5.3 The Outcomes

In this section the experience gained from the programming of applications for RHODOS DSM system and comparative evaluation of the obtained results against the results of other projects are presented.

```
initialisation

dsm_barrier(barrier[0])          /*Ensures all processes have
                                  completed initialisation*/

while(){
    wait(sem[0])                 /*Entry to critical region,
                                  guarantees exclusive access
                                  to priority queue*/
        if (empty queue)
            signal(sem[0])       /*Exits the critical region */
            exit()
        Add to the queue until the head
        tour looks promising
        Path = Tour from head of queue
        delete head tour
    signal(sem[0])               /*Exits the critical region */
}
length = Recursively try all cities not in Path to find the
         shortest tour length
wait(sem[1])                     /*Entry to the critical region
                                  guarantees exclusive access
                                  to Shortest_length*/
    if (length < Shortest_length)
        Shortest_length = length
signal(sem[1])                   /*Exits the critical region */

dsm_barrier(barrier[0])          /*Ensures all processes have
                                  completed execution before
                                  exiting */
```

Figure 5.11 Pseudocode for TSP.

5.3.1 Experience Gained from Programming of Parallel Applications for RHODOS DSM

In this section the experience gained form giving the task of programming applications for RHODOS DSM to an undergraduate student are reported. As stated in the previous section the five applications used were Red-Black Successive Over Relaxation, Matrix Multiplication, Jacobi, Quicksort and Travelling Salesman Problem.

Versions of code for Quicksort, Red-Black SOR and TSP which were written

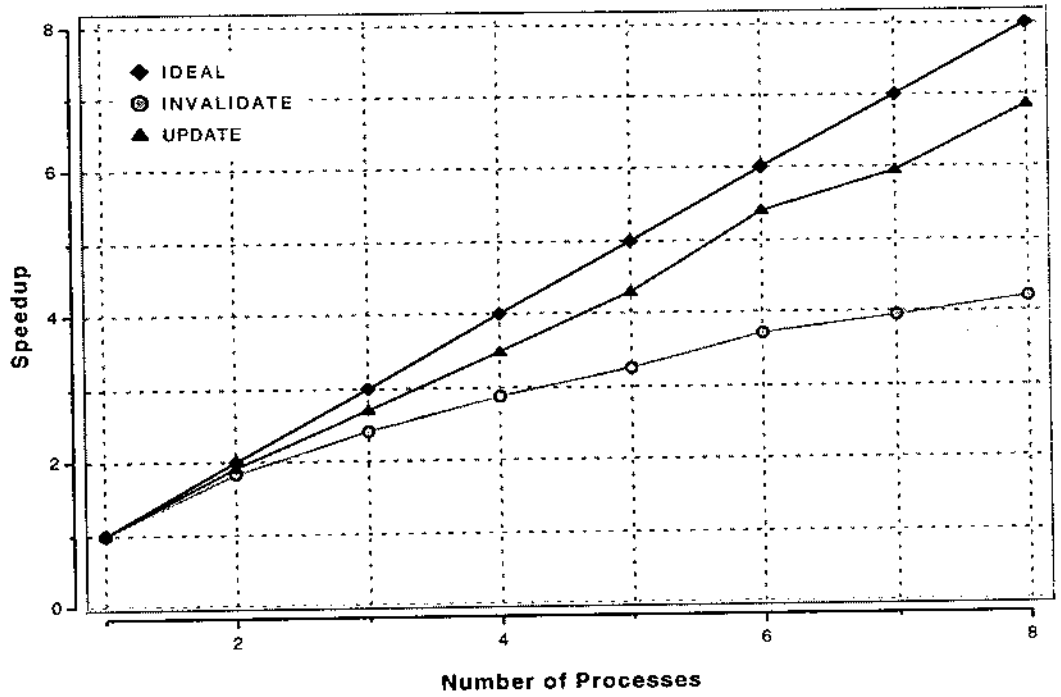

Figure 5.12 Speedup for TSP using invalidation and update-based DSM for an 18-city tour

for TreadMarks were obtained from Keleher [Keleher 96] and versions of Quicksort, Red-Black SOR, Matrix Multiplication and TSP which were written for IVY and Munin were obtained from the ftp site at Rice University.

Initially, TSP and Quicksort were converted for RHODOS DSM by the author. The conversion involved the removal of all TreadMarks, Munin or IVY related code followed by the addition of the initialisation and synchronisation code for RHODOS DSM. Performance tests were then carried out on these two applications measuring their speedup. Subsequently, the task of implementing (by either converting existing DSM code or writing the DSM code using a supplied algorithm) the remainder of the applications was given to an undergraduate student who had completed the first two years of a computer science degree, and as such he was a relatively inexperienced programmer. The task was part of the duties given to this student after being awarded a vacation scholarship. He had completed a number of units which involved some programming, including: *Basic Programming Concepts*, *Data Structures and Algorithms* and *Operating Systems* [Goscinski et al. 95]. The latter unit involved some concurrency issues, while all three units used C as the programming language.

This student was given a small amount of background on general DSM issues and shown the Quicksort and TSP implementations before being asked to convert the Matrix Multiplication and Red-Black SOR code and write the Jacobi code using an algorithm given in [Amza et al. 96] for execution in the RHODOS DSM environment. With his knowledge of the C language and the introduction to concurrency provided by his course work, he was easily able to complete the task of converting and writing the required code. The notable thing was that he had little knowledge of the implementation details of the RHODOS DSM system, yet he was able to perform his programming task easily without a need for this knowledge thereby demonstrating the transparency of the DSM system.

While it is difficult to quantify ease of programming and use, this experience indicates that the implementation of the DSM system proposed in Chapter 3 has met the goals of: easy to use, easy to program and transparency. In the next section the results of the performance tests will demonstrate that the system is also efficient.

5.3.2 Comparative Evaluation

In this section, the programming and performance aspects of RHODOS' DSM are discussed and compared with other DSM implementations. The comparison of performance is made difficult by the use of different hardware.

In [Carter et al. 95b] the results of tests performed comparing seven applications using message passing, "traditional DSM" and Munin on a network of Sun 3/60s with a 10Mbps Ethernet are reported. The "traditional DSM" used for this comparison is a page-based DSM which uses a write-invalidate based protocol similar to RHODOS' write-invalidate based DSM. Thus, the results generated in this part of Carter's tests can be compared with RHODOS DSM's performance results for TSP and Quicksort. The speedup shown by Carter for TSP using 8 processors is approximately 6.5, RHODOS DSM's speedup is 4.22 and that for Quicksort using 8 processors is approximately 2.4 while RHODOS DSM's speedup is 2.8. The major difference between RHODOS' write-invalidate based DSM and Carter's traditional DSM is that the latter's synchronisation is implemented through locks which are attached to particular shared variables instead of semaphores which define an area of code in

which shared variables are accessed. RHODOS DSM's solution provides the latter, semaphore-like, synchronisation. The different synchronisation methods will account for the variation in the speedup of TSP. RHODOS' implementation of DSM separates the synchronisation mechanism from the paging mechanism while Carter's implementation integrates the two. In the latter the locks are implemented as variables within the shared memory. In order to gain access to the lock a writable copy of the page containing the lock variable must be moved to the workstation requesting the access. This page also contains the data to be protected if the lock variable declaration has been placed adjacent to the data being protected. Thus, when the requesting workstation obtains the lock it also gets a writable copy of the page it will require in the critical section. This is, in effect, prefetching the page. While this implementation has improved performance this improvement can only be seen if application programmers place the lock variable in the correct position. Carter's implementation has sacrificed transparency for the sake of efficiency. On the other hand, the DSM described in this thesis provides transparency, one of the requirements identified in Chapter 3 as desirable for DSM design. As has been demonstrated in Section 5.3.1, any application programmer who is familiar with shared memory programming using semaphores will be able to use RHODOS' DSM without requiring additional instructions and a deeper understanding of the application than that required for shared memory programming.

Munin, also described in [Carter et al. 95b], is an object based DSM which has multiple consistency protocols. Programmers must select the protocol best suited to each variable in the application. Munin shows a speedup of 6.25 for 8 processors for Quicksort while RHODOS write-update based DSM shows a speedup of 3.8 and a speedup of approximately 6.8 for TSP while RHODOS write-update based DSM shows an 8-processor speedup of 6.85. The multiple consistency protocol approach of Munin places a considerable load on application programmers — in order to extract optimal performance from the DSM programmers must choose the correct consistency protocol for each variable.

Peter Keleher reported on the performance of TreadMarks in his Doctoral Thesis [Keleher 95]. TreadMarks uses a hybrid of the release consistency model called lazy release consistency which has been described in Section 2.4. RHODOS write-

update based DSM uses the release consistency model. Treadmarks uses locks rather than the semaphores used in RHODOS DSM. However, it is difficult to compare RHODOS DSM's speedups with those reported in Keleher's thesis and subsequent papers as the network hardware used in the Treadmarks tests has a total throughput of 1.2Gbps compared with RHODOS 10Mbps ethernet.

5.4 The Influence of Physically Shared Memory on the Performance of DSM

In this section the results of tests performed to investigate the influence of executing several DSM processes which share memory physically on a single workstation are reported.

It would be reasonable to expect that on a COW supporting DSM-based parallelism a multiprogramming approach where more than one process is executing on a single workstation would improve the overall performance, as another process may be able to execute while one was blocked. The issue is whether a single application could benefit from this approach. Performance tests carried out by researchers into DSM have measured the speedup of applications using a one to one ratio of processes to workstations [Carter et al. 95b], [Keleher 95], [Li and Hudak 89], thus, the effect of executing more than one process on one workstation is not clear. It is important to investigate the level of performance improvement, if any, exhibited by DSM-based parallel applications which physically share memory on a single workstation using a multiprogramming approach.

RHODOS DSM uses automatic initialisation (Section 4.4) to start the remote DSM processes and initialise the shared memory and synchronisation variables after the first DSM process has started. The operating system uses the system load information to place processes on workstations. An important aspect of the placement decision is whether the operating system should allow more than one process to execute on one workstation. Thus, performance test are required in order to ascertain whether there is any performance advantage to be gained through physically shared memory.

5.4.1 The Scope of Study

The goal of the performance study on RHODOS' DSM is to test whether there is any performance advantage to be gained from executing a parallel DSM-application where more than one process runs on a workstation. Furthermore, it is necessary to demonstrate that the DSM system is transparent because there is no difference between invoking and using physically shared and distributed shared memory.

The write-update based (release consistent) implementation was used for these tests because the performance tests described in Section 5.2 and in [Silcock and Goscinski 97c] have shown that RHODOS' write-update based DSM performs better than the write-invalidate based DSM. The performance study reported in Section 5.2 measured the performance improvements gained from executing several applications in parallel on between 1 and 8 workstations of a COW. By comparing the speedups for the performance tests where the ratio of processes to workstations is greater than one to one with those for tests carried out on the same applications using a one to one ratio it is possible to see whether there is any performance advantage to be offered by increasing the number of processes executing on each workstation in the COW. The applications used are:

- Travelling Salesman Problem (TSP); and
- Matrix Multiplication.

These two applications were selected out of the five applications used in Section 5.2 because they demonstrate two different kinds of parallelism. The algorithm for TSP uses functional parallelism (MPMD computational model) to solve the problem. On the other hand Matrix Multiplication exhibits data parallelism (SPMD computational model).

5.4.2 Multiprogramming and Physically Shared Memory

Multiprogramming was introduced in an effort to increase uniprocessor utilisation to a hundred percent. When a single process is executing on a uniprocessor that process may block waiting for the completion of an I/O operation or input from the programmer. The processor will then be idle until the process has completed its execution. If more than one process exists in memory in a ready to run state, one of

those processes could be selected by the operating system to execute when the executing process blocks. Thus, multiprogramming can be used to ensure that when one process blocks for any reason another process executes and the CPU is never idle while there is work to be done.

Furthermore, if these processes executing on one uniprocessor co-operate i.e., these processes synchronise with one another, which means that at times the processes block awaiting entry to a critical region, physically shared memory could be exploited. Thus, if two processes performing the same application were executed on the same workstation such that one could execute while the other were blocked it seems reasonable to expect that the overall execution time of the application would be improved.

Physical memory sharing in RHODOS' DSM has been designed to provide full DSM transparency. This means that there is no difference in invoking and reacting to results as actions of the DSM system performed on remote and local workstations. The design of the physically shared memory implemented in RHODOS is described in Sections 3.5 and 4.6 of this thesis.

5.4.3 Travelling Salesman Problem

The experiments were carried out using one up to eight workstations for each application for write-update based implementation of RHODOS DSM. One process executed on each workstation except one workstation on which two processes executed. The performance is again measured as the speedup of an application.

Discussion: The speedup shown by RHODOS' write-update based DSM for TSP where the speedup is plotted against the number of processes is shown in Figure 5.13. The results show a speedup for 9 processes of 6.4 when the process to workstation ratio is greater than one to one. The speedup for 8 processes when the ratio is one to one is 6.8. This indicates that there is no advantage to increasing the number of processes and duplicating them on the same workstation. Figure 5.14 shows the speedup against the number of workstations employed, where the ratio of processes to workstations is varied. This figure shows even more clearly that when more than one process is executing on one workstation the performance for nine processes on eight

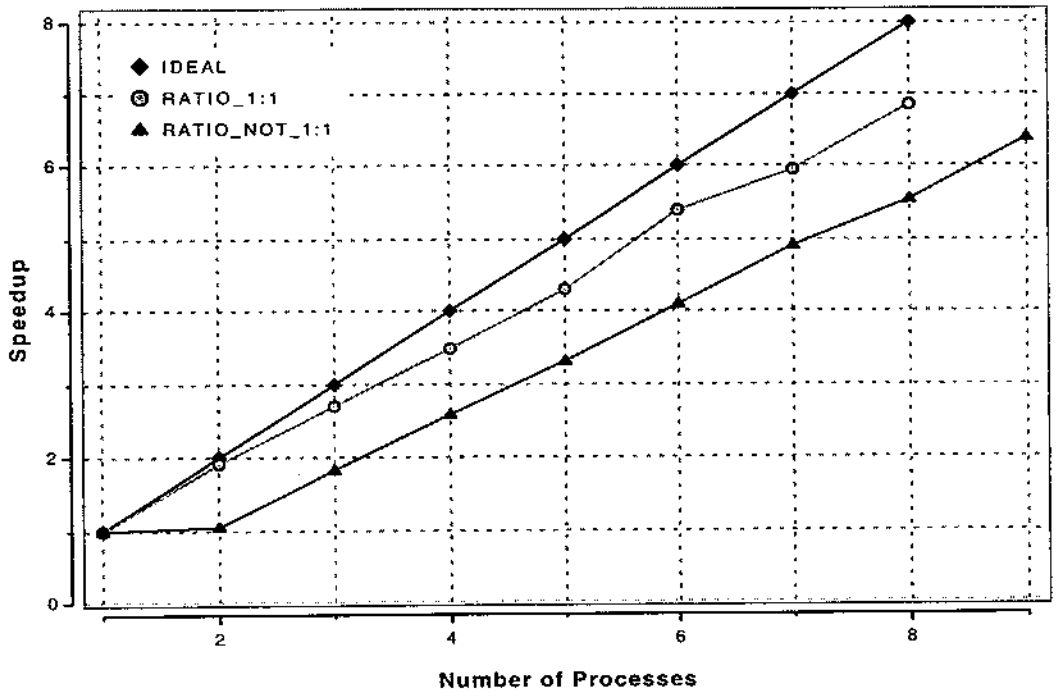

Figure 5.13 Speedup vs number of processes for ratio of processes to workstations is 1:1 and greater than 1:1 for the Travelling Salesman Problem

workstations is slightly worse than that for eight processes on eight workstations. Clearly the amount of time that the workstation is idle while the application is waiting for access to the critical region is not high enough to compensate for time spent context switching between processes.

5.4.4 Matrix Multiplication

The experiments were also carried out using one up to eight workstations for each application for write-update based implementation of RHODOS DSM. One process executed on each workstation except one workstation on which two processes executed.

Discussion: In Figure 5.15 the speedup versus the number of processes for matrix multiplication using write-update based DSM is illustrated. The three lines shown are for an ideal speedup and a process to workstation ratio of both greater than and equal to one to one for 1 up to 8 processors. The speedup for the latter two are 4.12 and 7.93, respectively. In Figure 5.16 the speedup vs number of workstations is shown

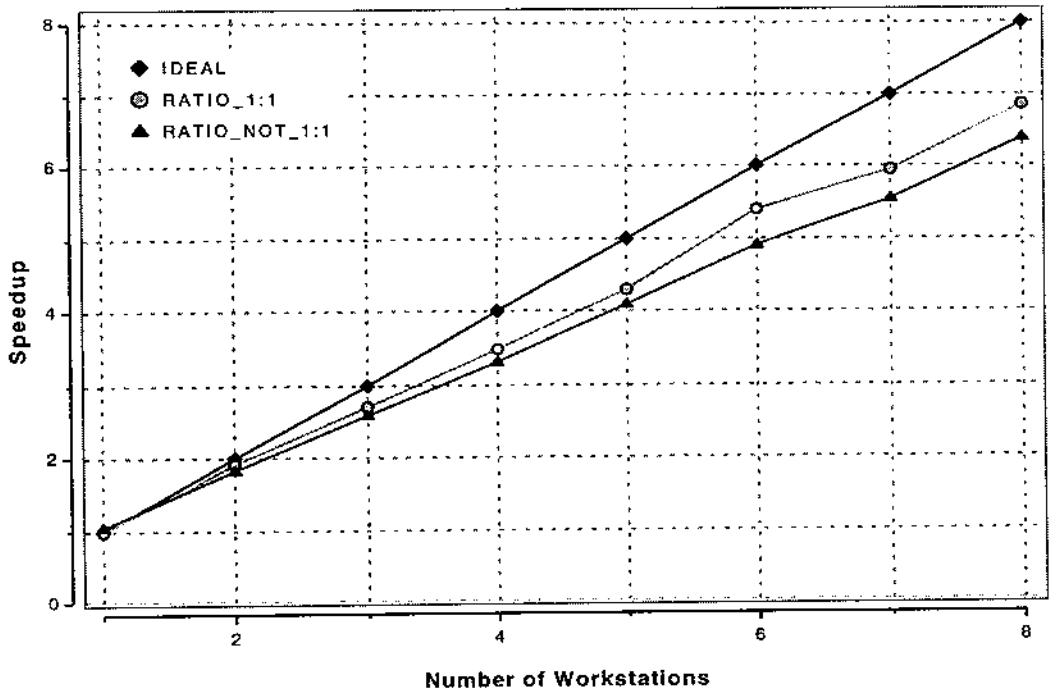

Figure 5.14 Speedup vs number of workstations for ratio of processes to workstations is 1:1 and greater than 1:1 for the Travelling Salesman Problem

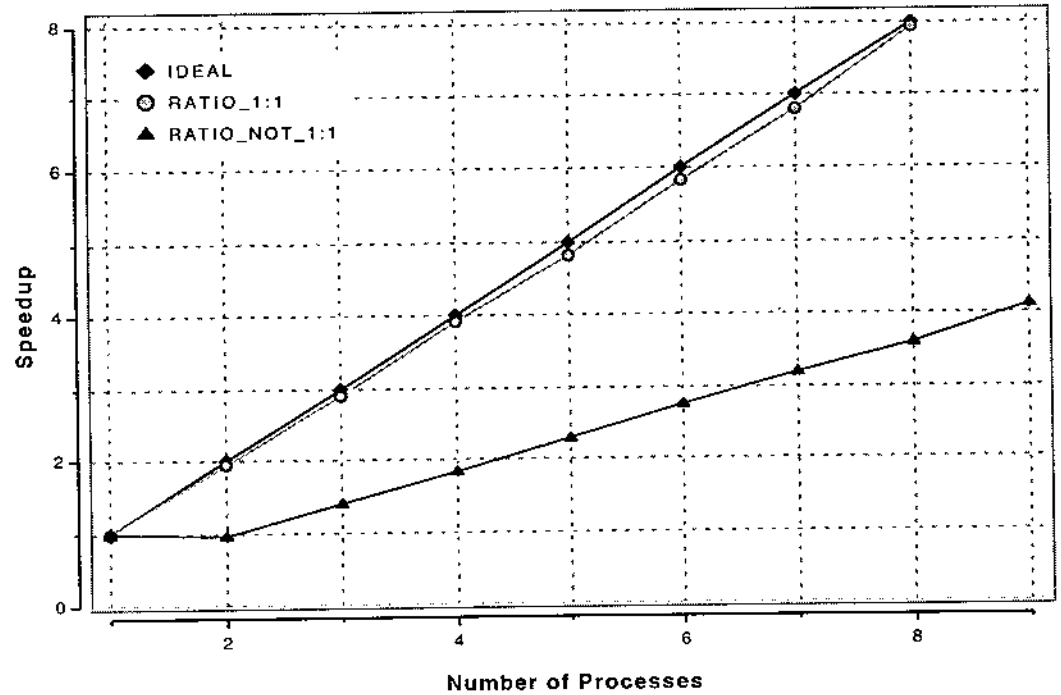

Figure 5.15 Speedup vs number of processes for ratio of processes to workstations is 1:1 and greater than 1:1 for the Matrix Multiplication

as opposed to processes in Figure 5.15. The speedup for the one to one ratio shown

here is very close to ideal. This is because matrix multiplication uses a SPMD computational model. There is no data sharing because each process works on its own section of the matrix independently and the whole matrix is updated only when the processes reach the barrier at the end of the computation. The speedup for 9 processes

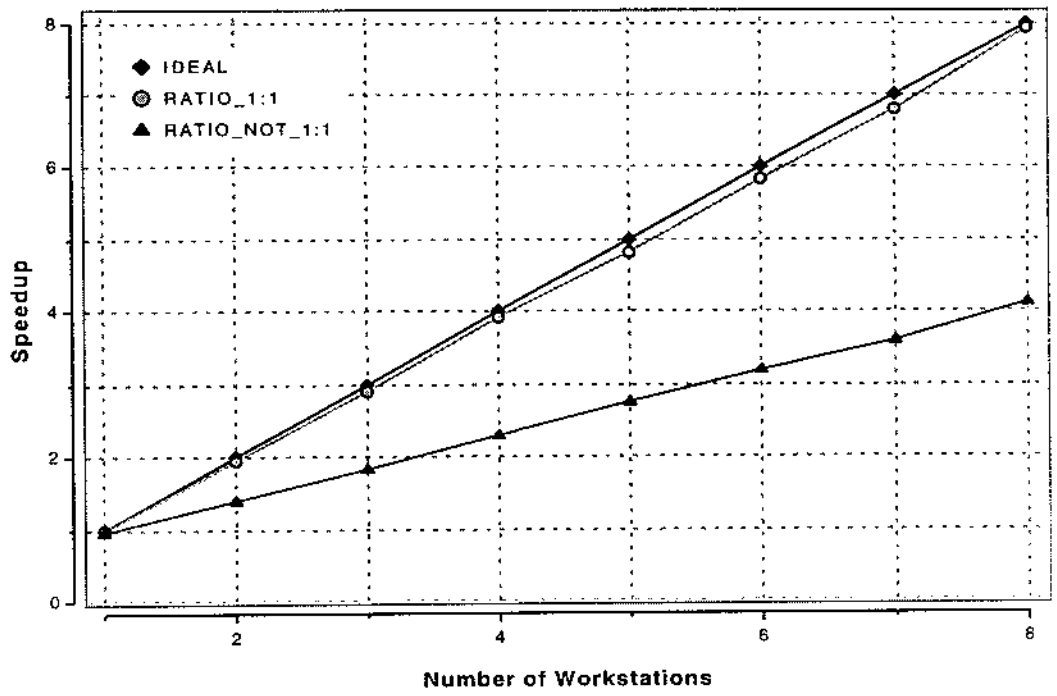

Figure 5.16 Speedup vs number of workstations for ratio of processes to workstations is 1:1 and greater than 1:1 for the Matrix Multiplication

on eight workstations is dramatically less than that for eight processes on eight workstations: 4.12 as opposed to 7.93. The reason for this is that because matrix multiplication uses a SPMD computational model, the matrix is divided by the number of processes executing. Each process then executes on its section of the matrix. This means that executing on each workstation is a process that is multiplying one ninth of the matrix except for the workstation on which two processes are executing. On this workstation two ninths of the matrix is being multiplied. This acts as a bottleneck as the speedup is reduced to that of approximately four processes, as shown in the figure. This implies that there is no physical memory sharing of which the parallel processes on one workstation can take advantage.

This differs from the TSP example where all processes remove tours from the

same tour queue and work independently; thus, the amount of computation carried out by one process is flexible. As such the two processes executing on one workstation will each perform approximately half the work of one process executing by itself on a workstation. In addition, because the SPMD model requires no data sharing there are no synchronisation points other than the start and finish barriers. Thus the processes do not block awaiting access to critical sections and the effect is that the two processes on one workstation run sequentially.

5.5 Conclusions

The goal of the chapter has been to validate the claim that the DSM system proposed in Chapters 3 and developed in Chapter 4 provides an environment for the development and execution of parallel applications which is:

- easy to program;
- easy to use;
- efficient; and
- transparent.

The ease of use and transparency of the DSM system have been demonstrated through the description of the ease with which a somewhat inexperienced undergraduate programmer was able to convert, write and run applications for the testing of the DSM system. Furthermore, the description of the tests performed using physically shared memory shows that the latter is indistinguishable from distributed shared memory; this is further evidence that the DSM system is fully transparent. Moreover, the performance tests carried out in Section 5.2 show that the DSM system meets the efficiency requirement.

Many of the applications currently used to test DSM software use the SPMD computational model in which the data is divided into sections on which each process works independently of other processes with little or no actual data sharing. The performance tests described in this thesis that use the MPMD model are TSP and the first phase of Quicksort. The larger number of synchronisation involved in problems using the MPMD model makes these problems less likely to show a dramatic performance improvement. Thus, more research is required with a larger range of

problems in order to identify the group of problems which show the greatest benefit from using DSM.

The tests performed on more than one process per workstation were used to establish whether it is worthwhile to allow the global scheduler to place more than one DSM process performing the same application to execute on the same workstation. It is clear from the initial results shown that there is no advantage for problems using either the MPMD or particularly SPMD computational models in increasing the number of processes executing on each workstation.

This chapter establishes that the aim of the research has been achieved; it is possible, as expected, to develop a programmer friendly and efficient DSM system fully integrated into a distributed operating system.

Chapter 6 Conclusions and Further Work

In this chapter the conclusions reached from the research reported in this thesis and the innovations of this research are discussed. In addition, further work to be carried out is reported.

6.1 Conclusions

The aim of this research was to synthesise and develop a distributed shared memory system as an integral part of an operating system in order to provide application programmers with a convenient environment which allows the easy development and execution of parallel applications in a transparent and efficient manner.

It has been demonstrated that to satisfy these requirements, the operating system into which the DSM system is integrated should be a distributed operating system. This aim has been achieved through:

- the synthesis of a DSM system within a microkernel and client-server based distributed operating system. This DSM system uses:
  - both the sequential and release consistency models, employing write-invalidate and write-update based coherence protocols, respectively; and
  - a unique automatic initialisation system which allows the programmer to start the parallel execution of a group of processes with a single library call. The number and location of these processes are determined by the operating system based on system load information;
- the implementation and testing of the developed DSM system within the RHODOS distributed operating system;
- the validation of the ease of use, ease of programming and transparency of the RHODOS DSM system through the programming work of a novice programmer developing applications for the system; and

- the validation of the system performance through the completion of performance studies on the RHODOS DSM system using applications which use both the SPMD and MPMD computational models.

The major points gained from a study of existing DSM systems were that an operating system integrated DSM is required. This system should use a weak consistency model and provide application programmers with an environment for which it is easy to write parallel applications and which supports their efficient execution. Following this, a set of new and original design requirements for the proposed DSM have been identified as, ease of programming, ease of use, transparency, efficiency and scalability. As an outcome, the DSM system presented in this thesis has a novel approach in that it provides programmers with a complete programming environment in which they are easily able to develop and run their code only exploiting their knowledge of writing code for centralised shared memory systems or indeed run existing shared memory code.

The ease of use and transparency of the DSM system have been demonstrated through the description of the ease with which a somewhat inexperienced undergraduate programmer was able to convert, write and run applications for the testing of the DSM system. Furthermore, the results of the performance tests on five applications demonstrate that RHODOS' DSM meets the efficiency requirement. These results show that RHODOS' DSM performs comparably with other implementations.

Automatic initialisation provides the system with ease of use and transparency because it allows programmers to insert a simple library call to initialise parallel execution on a number of workstations. The operating system handles the decisions about the number of child processes to start and where to execute them, in fact the system actually starts the child processes on the remote workstations. The use of RHODOS' Global Scheduler and REX Manager to provide the automatic initialisation facility shows that, with the inclusion of a DSM system, this operating system is able to support parallel processing and truly meets one of the operating system aims of user convenience.

Moreover, the inclusion of two consistency models in the proposed design

makes it possible for application programmers with a knowledge of DSM to choose the model they think is the most appropriate for their application. The design and semantics of the sequential and release consistent DSM systems and their relationship with the semaphore-type and barrier synchronisation validate the claim that the proposed system is feasible.

Further, the manner in which RHODOS' operating system integrated DSM deals with physically shared memory highlights the fact that distributed and physically shared memory are employed by users in precisely the same manner.

The innovations of this research are:

- The synthesis and development of the first DSM system fully integrated into a distributed operating system which allows application programmers to easily and transparently write and run their shared memory applications, and achieve high performance execution;
- The proposal and development of placement and architecture of the DSM system within the memory server of a microkernel and client-server based distributed operating system which provide a flexible and configurable environment for the DSM system. The placement in a memory server makes DSM an integral part of the operating system functionality and as such provides users with a complete environment which allows the development and execution of shared memory code in the same manner as that for centralised memory;
- The proposal, synthesis and development of the DSM system as an event driven set of co-operating and distributed software entities described by two basic protocols, the DSM parent protocol and DSM child protocol; and
- The proposal, design and development of an automatic initialisation system for the parallel DSM processes which provides a novel way of exploiting the remote process creation and scheduling ability of the operating system to make the programmer's task easier and make the DSM system transparent.

In summary, client-server and microkernel based distributed operating system integrated DSM makes shared memory operations transparent and almost completely

removes the involvement of the programmer beyond classical activities needed to deal with shared memory; only barrier-based synchronisation of parallel processes is needed. From all of these results it can be concluded that DSM, when implemented within a client-server and microkernel based distributed operating system, is one of the most promising approaches to parallel processing and guarantees performance improvements with minimal programmer involvement.

The principles of experimental computer science have been employed to achieve the research aim. As "proof-of-concept", the proposed DSM system has been implemented within the RHODOS distributed operating system. Furthermore, as "proof-of-performance" the implementation has been used to show that the performance levels are at least as good as those for other DSM systems which are built on top of or as part of existing operating systems.

6.2 Further Work

Many of the applications currently used in research to test DSM software use the SPMD computational model in which the data is divided into sections on which each process works independently of other processes. In the tests performed in this thesis only TSP and part of Quicksort were performed using the MPMD computational model. In order to prove that DSM can be useful outside the research environment it would be valuable to perform these tests on more problems that use the MPMD computational model that involve real sharing of the data. This may lead to the identification of a subset of problems that show great performance benefits from execution using DSM.

The write-invalidate and write-update coherence protocols have been used to implement the sequential and release consistency models, respectively. The performance of the two consistency models should be tested when implemented using the other coherence protocol, i.e. implementing sequential consistency using the write-update protocol and release consistency using the write-invalidate protocol. If sequential consistency was implemented using the write-update protocol updates would be propagated as soon as a write operation were performed. Alternately if write-invalidate were used to implement release consistency, all copies of a page that was

about to be written to in a critical section would be made invalid before the write took place. The pages would be replaced with updated copies when required later. Although similar research was carried out in [Keleher 95], it is important to observe whether the findings for that research apply in the RHODOS DSM environment.

Centralised barrier management has been used in this and in the DSM systems studied. Where small numbers of workstations are used to execute the parallel applications this solution is acceptable. However, with larger numbers of workstations the centralised manager could become a bottleneck and some form of decentralised barrier management should be implemented.

Currently RHODOS DSM processes are fixed on the workstation on which they start execution. Since the RHODOS operating system has a migration facility the DSM system should be extended to allow the migration of executing DSM processes.

Bibliography

[Adve and Hill, 90] S. Adve and M. Hill, *Weak Ordering - A New Definition*, Proceedings 17th Annual International Symposium on Computer Architecture, May 1990.

[Ananthanarayanan et al. 92] R. Ananthanarayanan, M. Ahamad and R. LeBlanc, Jr., *Application Specific Coherence Control for High Performance Distributed Shared Memory.* In Proceedings of The Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS-III), March 1992.

[Amza et al. 96] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, *Treadmarks: Shared Memory Computing on Networks of Workstations*, IEEE Computer, Vol. 29, No. 2, Feb 1996.

[Bal et al, 92] H. Bal, M. Kaashoek and A. Tanenbaum, *Orca: A Language for Parallel Programming of Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 18, No. 3, March 1992.

[Bershad et al. 91] B. Bershad and M. Zekauskas, *Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*, CMU Technical Report CMU-CS-91-170, September 1991.

[Bershad et al. 93] B. Bershad, M. Zekauskas and W. Sawdon, *The Midway Distributed Shared Memory System*, Proceedings of the IEEE COMPCON Conference, IEEE, 1993.

[Carriero and Gelernter 86] N. Carriero and D. Gelernter, *The S/Net's Linda Kernel.*, ACM Transactions on Computer Systems, Vol. 4, No. 2, May 1986.

[Carter 93] J. Carter, *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*, PhD Thesis, Rice University, September 1993.

[Carter et al. 95a] J. Carter, D. Khandekar and L. Kamb. *Distributed Shared Memory: Where We Are And Where We Should Be Headed*, Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, May 1995.

[Carter et al. 95b] J. Carter, J. Bennett, and W. Zwaenepoel. *Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems*, ACM Transactions on Computer Systems, Vol. 13 No. 3, August 1995.

[Coulouris et al. 93] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems. Concepts and Design.* Addison-Wesley Publishing Company, 1993.

[Chen and Dasgupta 91] R. C Chen and P. Dasgupta, *Implementing Consistency Control Mechanisms in the Clouds Distributed Operating System.* In Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11), May 1991.

[De Paoli et al. 95] D. De Paoli, M. Hobbs and A. Goscinski, *Microkernel and Kernel Server Support for Parallel Execution and Global Scheduling on a Distributed System,* In Proceedings IEEE First International Conference on Algorithms and Architectures for Parallel Processing, April 1995.

[Dinning 89] A. Dinning, *A Survey of Synchronization Methods for Parallel Computers.* IEEE Computer, 22, 7, July 1989.

[Fleisch and Popek 89] B. Fleish and G. Popek, *Mirage: A Coherent Distributed Shared Memory Design,* Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP-12), December 1989.

[Fleisch et al. 94] B. Fleish, R. Hyde and N. Juul, *Mirage+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers,* Software - Practice and Experience, John Whiley and Sons, Ltd, March 1994.

[Gharachorloo et al. 90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J, Hennessy, *Memory Consistency for Scalable Shared-Memory Multiprocessors,* Computer Architecture News, 18(2), June 1990.

[Goscinski 91] A. Goscinski, *Distributed Operating Systems. The Logical Design, Addison-Wesley Publishing Company.*

[Goscinski et al. 95] A. Goscinski, P. Horan, S. Kutti, D. Newlands, J. Teague, G. Webb and W. Zhou, *Computer Science/Software Development and Information Systems Curricula within the School of Computing and Mathematics,* School of Computing and Mathematics, Deakin University, August 1995.

[Goscinski 97] A. Goscinski, *Parallel Processing on Clusters of Workstations,* Proceedings of the IEEE Singapore Conference on Networks, SICON '97, April 1997.

[Gropp 92] W. Gropp, *Parallel Computing and Domain Decomposition,* Proceedings of the Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, 1992.

[Grunwald and Vajracharya 94] D. Grunwald and S. Vajracharya, *Efficient Barriers for Distributed Shared Memory,* 8th International Symposium on Parallel Processing, April 1994.

[Hellwagner 90] H. Hellwagner, *A Survey of Virtually Shared Memory Systems*, TUM-19056 Technische Universitat Munchen, December 1990.

[Hobbs et al. 95] M. Hobbs, D. De Paoli, A. Goscinski, G. Wickham and R. Panadiwal, *Generic Memory Objects for Supporting Distributed Systems*. Proceedings of the International Conference on Automation, Indore, India, December 1995.

[Iftode and Singh 97] L. Iftode and J. P. Singh, *Shared Virtual Memory: Progress and Challenges*, Technical Report, TR-552-97, Department of Computer Science, Princeton University, October 1997.

[Keane et al. 95] J. Keane, A. Grant and M. Xu, *Comparing Distributed Memory and Virtual Shared Memory Parallel Programming Models. Future Generation Computer Systems*, Special Issue: Massive Parallel Computing. North-Holland, 1995.

[Keleher 95] P. Keleher, *Lazy Release Consistency for Distributed Shared Memory*, PhD Thesis, Rice University, January 1995.

[Keleher 96] P. Keleher, *Private Communication*, April 1996.

[Lamport 79] L. Lamport, *How to make a multi processor computer that correctly executes Multiprocess programs*, IEEE Transactions on Computers, Vol. C-28, September 1979.

[Levelt et al. 92] W. Levelt, M. Kaashoek, H. Bal, and A. Tanenbaum, *A Comparison of Two Paradigms for Distributed Shared Memory*, Software-Practice and Experience, vol. 22, November 1992.

[Li and Hudak 89] K. Li and P. Hudak, *Memory Coherence in Shared Virtual Memory Systems*, ACM Transactions on Computer Systems, Vol. 7, No. 4, November 1989.

[Li 88] K. Li, *IVY: A Shared Virtual Memory System for Parallel Computing*, Proceedings of the 1988 International Conference on Parallel Processing, 1988.

[Li 86] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale 1986.

[Libes 85] Don Libes, *User-Level Shared Variables*, Proceedings of the Tenth USENIX Conference, 1985.

[Lu 95a] H. Lu, *Message Passing Versus Distributed Shared Memory on Networks of Workstations*, PhD Thesis, Rice University, May 1995.

[Lu et al. 95] H. Lu, S. Dwarkadas, A.L. Cox and W. Zwaenepoel. *Message Passing Versus Distributed Shared Memory on Networks of Workstations*, Proceedings of the 1995 Conference on Supercomputing'95, December 1995.

[Lu et al. 97] H. Lu, S. Dwarkadas, A.L. Cox and W. Zwaenepoel. *Quantifying the Performance Differences between PVM and TreadMarks*, Journal of Parallel and Distributed Computation, Vol. 43, No. 2, pp. 65-78, June 1997.

[Mohindra 93] A. Mohindra, *Issues in the Design of Distributed Shared Memory Systems*, PhD Thesis, Georgia Institute of Technology, May 1993.

[Mosberger 94] D. Mosberger, *Memory Consistency Models*, Operating Systems Review, January 1993.

[Nitzberg and Lo 94] B. Nitzberg and V. Lo, *Distributed Shared Memory: A Survey of Issues and Algorithms*, In Casavant T.L. and Singal M. (eds), Readings in Distributed Computing Systems, IEEE Press, 1994.

[Protic et al. 97] J. Protic, M. Tomasevic and V. Milutinovoc, *Tutorial on Distributed Shared Memory: Concepts and Systems*, http://galeb.etf.bg.ac.yu/~vm/tutorial/multi/dsm/software/software.html, 1997.

[Ramachandran and Singhal 95] M. Ramachandran and M. Singhal, *Decentralised Semaphore Support in a Virtual Shared-Memory System*, The Journal of Supercomputing, 9, 51-70, 1995.

[Ramachandran 95] M. Ramachandran, *Algorithms to Implement Semaphores in Distributed Environments*, PhD Thesis, The Ohio State University, 1995

[Rhodes 97] D. Rhodes, *Private Communication*, January 1997.

[Scales and Gharachorloo 97] D. Scales and K. Gharachorloo, *Towards Transparent and Efficient Software Distributed Shared Memory*, Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, OS Review, December 1997.

[Silberschatz and Peterson 88] A. Silberschatz and J. Peterson, *Operating Systems Concepts*. Addison-Wesley Pub. Company, 1988.

[Silcock and Goscinski 95] J. Silcock and A. Goscinski. *Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems*, Technical Report TR C95/20, June 1995.

[Silcock and Goscinski 97a] J. Silcock and A. Goscinski, *Invalidation-Based Distributed Shared Memory Integrated into a Distributed Operating*

System, Proceedings of IASTED International Conference Parallel and Distributed Systems (Euro-PDS'97), June 1997.

[Silcock and Goscinski 97b] J. Silcock and A. Goscinski, *Update-Based Distributed Shared Memory Integrated into RHODOS' Memory Management*, Proceedings of Third International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'97), Springer-Verlag, December 1997.

[Silcock and Goscinski 97c] J. Silcock and A. Goscinski, *Performance Studies of Distributed Shared Memory Embedded in the RHODOS' Operating System*, Proceedings of The 4th Australasian Conference on Parallel and Real-Time Systems (PART '97), September, 1997.

[Silcock and Goscinski 98] J. Silcock and A. Goscinski, *The Influence of the Ratio of Processes to Workstations on the Performance of DSM*, Proceedings of The 21st Australasian Computer Science Conference (ACSC'98), February, 1998.

[Snyder 95] L.Snyder, *What is Experimental Computer Science*, Technical Report TR C95/18, Deakin University, March 1995.

[Tam et al. 90] M. Tam, J. Smith and D. Farber, *A Survey of Distributed Shared Memory Systems.*, ACM SIGOPS, June 1990.

[Tanenbaum 95] A. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.