Order Number 8728365

# Shared virtual memory on loosely coupled multiprocessors

Li, Kai, Ph.D.

Yale University, 1986



U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# Shared Virtual Memory
# on
# Loosely Coupled Multiprocessors

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the degreee of
Doctor of Philosophy

by

Kai Li

December, 1986

# Abstract

## Shared Virtual Memory on Loosely Coupled Multiprocessors

Kai Li

Yale University

1986

This dissertation demonstrates that parallel programs using shared virtual memory on loosely coupled multiprocessors can achieve orders-of-magnitude speedups over a uniprocessor and that it is practical to implement a shared virtual memory on existing architectures. Virtual memory has proven benifits. Today, almost every high performance sequential computer has one. While one can easily imagine how virtual memory would be incorporated into a shared-memory parallel machine, on a multiprocessor in which the physical memory is distributed, the implementation of virtual memory is not obvious.

This dissertation presents algorithms for solving memory coherence problems in a shared virtual memory on loosely coupled multiprocessors. It discusses basic mechanisms for process scheduling (including process migration) and memory management. Many different strategies are presented, analyzed, and compared, and a few of the most viable ones are chosen for implementation.

A prototype system, IVY, has been implemented on a local area network of Apollo workstations. The experimental results show that parallel programs using the shared virtual memory system yield almost linear and occasionally super-linear speedups. The success of this implementation suggests that shared virtual memory on loosely coupled multiprocessors can exploit the total processing power and memory capabilities in a far more unified way than the traditional "message-passing" approach.

## Commendatory Poem by Dr. Otto C. Steinmayer

Ad amicum *KAI LI*, cum exercitationem academicam pro gradu
Doctoris Philosophiae, quam de
*λογισμῶι παραλλήλωι* conscripsit, manibus solvisset.

CAELI, mirabar quid secum machina volvat,
quoque modo arcanas particulas agitet

morsaque. Cogitat? estne memor tam dura silex? aut
segminis in textu mensque animusque latent?

Quid non? Ingenia ast inter se ferrea nexis
praeceptum ut pariter multa sequantur idem.

Computus exiguo prius actus tempore simplex
arte tuá paritur pluries in minimo!

Multijugus currus citius volat, et capita aiunt
quam solum melius plusque valere dua.

En, venit in lucem perfectus, docte, libellus;
laetor quod tandem laurea serta feras.

Perge, Syracosiá[1] maius quid habens ratione;
scibilia astutus calculet omnia abax.[2]

Tractet plura licet mundi, carissime, harenis.
erga te meus est innumeralis amor.

---

[1] Cf. Archimedes *Arenarius*

[2] Priscianus p.688

## Acknowledgements

I would like to thank my advisors, Paul Hudak and Alan Perlis. Paul has provided me with support, encouragement, advice, and freedom during my last two years at Yale. His high standards of research and his scientific attitude towards writing substantially improved this dissertation. Alan is an infinite source of ideas. He taught me how to look into possibilities in research instead of following the conventions of others. The other members of my committee, Bill Gropp and Garret Swart, were also helpful. Bill answered innumerable questions about scientific computing and suggested several parallel benchmark programs. Garret read the thesis and corrected several mistakes.

I owe a great debt to my persevering counsellor and good friend John Ellis. John convinced me of the possibilities of implementing the shared virtual memory when I first had the ideas in March 1984 and has continued to contribute to the development of ideas since then. John read this thesis very carefully. His comments significantly improved its range, accuracy, and conciseness.

Dennis Philbin read and edited the manuscript twice, which resulted in countless improvements and clarifications.

Nat Mishkin showed me how to poke around the Apollo Aegis kernel, which made my modifications to the OS possible. Discussions with Michael Fischer helped me with the distributed manager algorithms in Chapter 2. Part of the idea of distributing copy sets in Chapter 2 is the result of a discussion with Larry Stewart. David Jefferson brought to my attention page table compactions and memory replacement algorithms for large-scaled multiprocessors.

I thank my officemates Norman Adams from whom I learned many things about the U.S. and about system programming, Richard Kelsey who wrote my favorite game, Asteroids, and Jim Philbin who encouraged me many times when I was depressed in debugging parallel programs. They gave me many useful suggestions for this dissertation.

Michella Schubert, Chris Hatchell and Andrea Pappas took care of many details which complicate the life of the graduate student struggling with his dissertation. Richard Guillemette and David Teodosio saved me from losing my dissertation files when my disk crashed.

Otto Steinmayer contributed his poem for the dissertation.

Sally Mckee gave me her car and saved me from walking in snow at night during my thesis writing.

My sincere thanks to my parents who supported me for many, many years. I am privileged to be in the family. I dedicate this dissertation to them.

# Contents

# Chapter 1

# Shared Virtual Memory

Shared virtual memory on a loosely coupled multiprocessor can achieve orders-of-magnitude speedups over a uniprocessor for many parallel programs, and it is practical to implement such a memory on existing architectures.

## 1.1 Introduction

The benefits of a virtual memory go without saying; almost every high performance sequential computer in existence today has one. In fact, it is hard to believe that parallel architectures would not benefit from virtual memory. One can easily imagine how virtual memory would be incorporated into a *shared*-memory parallel machine, because the memory hierarchy need not be much different from that of a sequential machine. On a multiprocessor in which the physical memory is *distributed*, the implementation is not obvious. It is the thesis of this dissertation that such an implementation is not only possible, it is also desirable.

The *shared virtual memory* proposed here provides a virtual address space that is shared among all processors in a loosely coupled multiprocessor system. Application programs can use the shared virtual memory just as they do a traditional virtual memory, except, of course, that processes can run on different

processors in parallel.

The shared virtual memory not only "pages" data between physical memories and disks, as in a conventional virtual memory system, but it also "pages" data between the physical memories of the individual processors. Thus data can naturally *migrate* between processors on demand. Also, just as a conventional virtual memory swaps *processes*, so does the shared virtual memory. Thus the shared virtual memory provides a natural and efficient form of *process migration* between processors in a distributed system. This is quite a gain because process migration is usually very difficult to implement. In effect, process migration subsumes *remote procedure calls*.

This dissertation discusses design methods for implementing a shared virtual memory system on loosely coupled multiprocessors; it also presents the results of experiments run on the prototype system.

The prototype system, IVY has been implemented on a local area network of Apollo workstations. The experimental results of non-trivial parallel programs run on the prototype show the viability of a shared virtual memory. The success of this implementation suggests an operating mode for such architectures in which parallel programs can exploit the total processing power and memory capabilities in a far more unified way than the traditional "message-passing" approach.

## 1.2 Multiprocessors

The terminology of parallel computing is not very precise, and it varies greatly in the literature. The term *multiprocessor* sometimes refers to a number of processors that communicate with each other but are not geographically distributed; other times its definition not only includes vector computers but also includes geographically distributed computers. In this thesis, a multiprocessor is simply defined as a computer architecture with the following two attributes:

- more than one processor (or processing element), and

- one or more communication links that allow data transfer between the processors.

According to Flynn's classification of computer organizations [Flynn 66], this definition includes all Multiple Instruction Multiple Data (MIMD) architectures and those Single Instruction Multiple Data (SIMD) architectures having more than one processor. Geographically distributed computers, such as a network of computers, are included because they belong to the MIMD category.

Multiprocessors can be classified by their communication mechanisms and memory configurations. Figure 1.1 provides a table of possible configurations and examples of existing architectures that fit each configuration. The communication mechanisms are divided into two classes, tightly coupled and loosely coupled. A multiprocessor is *tightly coupled* if a processor can reference the memory of another processor by a single instruction. For example, a multiprocessor with a shared memory is tightly coupled. A multiprocessor is *loosely coupled* if a processor requires interrupts rather than one instruction to make a remote memory reference. In this case, the only way that one processor communicates with another is by sending packets of data via a communication link. A local area network of computers falls into this category.

Memory configurations are divided into three classes: global memories, global memories built by distributed local memories, and local memories. From the point of view of clients, there are also many tightly coupled multiprocessors that have both global memories and local memories.

Tightly coupled multiprocessors with only physical global memories usually use multicaches to keep their global memories coherent so that the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. When the cache on a processor is fairly large, the hit ratio is expected to be high. This kind of machine is good for fine-grained parallel applications. Since the communication between caches and global memory is via a fast bus, it is hard to provide more than a few tens of processors [Archibald 85]. Firefly [Thacker 84], Dragon [McCreight 84] and

| | *Global memories* | *Global memories built by Local memories* | *Local memories* |
|---|---|---|---|
| *Tightly coupled* | Firefly, Dragon, Synapse, etc. | CM*, Concert, Butterfly, RP3, etc. | Illiac IV, Warp, etc. |
| *Loosely coupled* | *not possible* | | Cosmic Cube, VAXclusters, LAN, etc. |

Figure 1.1: Classification of multiprocessors.

Synapse [Frank 84] belong in this category.

Tightly coupled multiprocessors with global memories built by distributed local memories include machines that take varied approaches to the data coherence problem. Some of the architectures in this category solve the data coherence problem in hardware, some solve it in software, and some solve it in both. Multiprocessors such as CM* [Fuller 78, Jones 80], Concert [Anderson 82] and Butterfly [Larus 84] use shared buses and clusters or crossbars to reference remote memories. RP3 [Pfister 85] is a large-scale architecture constructed by the combination of multicaches, buses, and communication networks.

Certain tightly coupled multiprocessors have only local memories in which a processor can use a single instruction to access the memories of its neighbor processors. Examples in this group are Illiac IV [Barnes 68] and Warp [Annaratone 86]. In Illiac IV, a register of a processor can communicate with a register of another processor via a routing network. In Warp, an one-dimensional systolic array processor, processors communicate through crossbars.

All loosely coupled multiprocessors have only local memories. These multiprocessors do not have a direct remote memory reference; instead processors communicate by sending packets through their communication links which can be either local network links or fast bus links. Examples include the Cosmic Cube [Seitz 85], VAXclusters [Kronenberg 86], and local area networks of computers [Metcalfe 76, Leach 83].

As Figure 1.1 shows quite plainly, in the spectrum of multiprocessor configurations, there is one conspicuous hole. The goal of this research is to fill this hole by investigating how to build a shared memory on loosely coupled multiprocessors, and to explore its realization on existing machines.

## 1.3 Shared Memory vs. Message Passing

Since the basic communication mechanism in loosely coupled multiprocessors is sending data via communication links, it is natural to use message passing as low-level software support. This raises the question of whether a shared memory based system can be more efficient than a message passing based system for parallel computing.

Message passing in concurrent systems is characterized by multiple threads of control. A pure message passing system usually does not have any shared global data; instead processes access ports or mailboxes to achieve interprocess communication. Parallel programs need to use primitives such as *send* and *receive* explicitly through channels, ports, or mailboxes. Although programmers can use these primitives to synchronize parallel programs, they need to be conscious of data movement between processes at all times.

*Remote procedure call* is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel [Nelson 81]. A remote procedure call mechanism allows programmers to worry less about data movement and and provides clients with a fairly transparent interface so that remote procedure calls look much

like local procedure calls. However, the transparency of remote procedure calls is limited because a remote procedure call mechanism actually simulates the execution in the same address space using completely different address spaces.

Since both message passing and remote procedure calls deal with multiple address spaces, they both have difficulties with passing complex data structures. In fact, the difficulty of passing complex data structures is the main drawback of message passing and remote procedure calls for parallel programming. For example, passing a list data structure by sending messages will introduce considerable complexity in programming and substantial overhead in both space and time [Herlihy 82]. In a remote procedure call, there is no good way to pass a pointer argument [Nelson 81]. This problem becomes more severe when the data structures are fundamental to a language being implemented on a parallel machine.

In contrast, a shared memory multiprocessor has no difficulty passing pointers because processors share a single address space. Therefore, there is no need to pack and unpack the data structures containing pointers in messages. Passing a list data structure simply requires passing a pointer.

Another problem with message passing systems is the difficulty of process migration because there are multiple address spaces. When migrating a process, all the operating system resources allocated by the process have to be moved together; this is very expensive [Powell 83]. In the case where a process has a few opened ports and files, the pending messages and file access control blocks need to be transferred. Furthermore, the code and the stack of the process have to be moved because there is no easy way to translate the contents of different address spaces efficiently on the fly.

In a shared memory multiprocessor system, a process migration only requires moving a process from the ready queue on the source processor to the ready queue on the destination processor because process control block, code, and stack are all in the same address space.

Both data structure passing and process migration are important for imple-

menting parallel programming languages. Although some implementations of parallel programming languages are based on a message passing facility, experience with implementing existing parallel languages has shown that a shared memory architecture can greatly simplify the implementations [Hudak 86b, Carriero 86b]. In sum, shared memory is highly desirable for parallel computation.

## 1.4 Shared Memory on Loosely Coupled Multiprocessors

Many tightly coupled multiprocessors with shared memories are commercially available, and the computing power of some of these machines can compete with fast vector machines [Hillis 85, Pfister 85]. Tightly coupled multiprocessors should perform better than loosely coupled multiprocessors for parallel computation. So why bother to implement a shared memory on a loosely coupled multiprocessor?

First of all, to date, tightly coupled multiprocessors have been expensive, and it is not yet clear whether they are truly cost effective. Loosely coupled multiprocessors, on the other hand, are relatively less expensive, which is why it is common to have a local area network of computers.

Secondly, loosely coupled multiprocessors such as a network of workstations provide programmers with a comfortable programming environment. For example, a graphic bit-map display is something that a centralized computing environment cannot offer. On most tightly coupled multiprocessors, one has to write parallel programs to achieve fast computation speed and parallel programming is more difficult than sequential programming. It is clear that tightly coupled multiprocessors are far from replacing the programming environment of a network of workstations.

Furthermore, loosely coupled multiprocessors have higher availability than

tightly coupled multiprocessors. In a loosely coupled multiprocessor system such as a network of computers, processors do not rely on each other much; when some processors go down, other processors may still be usable. Tightly coupled multiprocessors are poor in this aspect. Processors in a tightly coupled multiprocessor usually depend on one another in the sense that when one processor crashes, the whole system goes down.

Although loosely coupled multiprocessor systems offer low cost, good programming environments, and high availability, their utilization is usually poor. For example, on the Apollo ring [Leach 83] at Yale University, a large percentage of the nodes are often idle, especially after working hours. One can imagine that if a shared memory on a loosely coupled multiprocessor could be built to achieve orders-of-magnitude speedup over a uniprocessor, the unused computing power in the existing loosely coupled multiprocessor could be effectively harnessed at no additional hardware cost.

Another reason, perhaps a more important one, is that if it is possible to build a shared memory on a loosely coupled multiprocessor, it should be possible to build a shared memory on a network of tightly coupled multiprocessors. This suggests a radical idea in parallel architectures—building a shared memory based parallel architecture by using loosely coupled links and software instead of using expensive interconnection networks.

## 1.5 A Shared Virtual Memory System

A *shared virtual memory* is a single address space shared by a number of processors (Figure 1.2). Any processor can access any memory location in the address space.

The memory mapping managers in a shared virtual memory implement the mapping between local memories and the shared virtual memory address space. They keep the address space coherent at all times; that is, the value returned by a read operation is always the same as the value written by the most recent

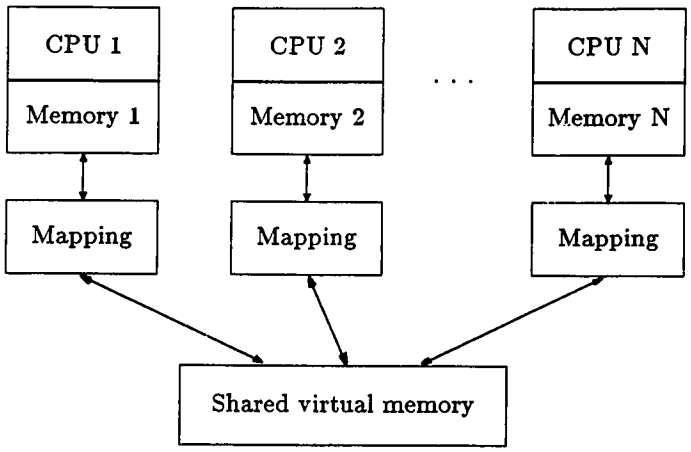


Figure 1.2: Shared virtual memory mapping

write operation to the same address.

A shared virtual memory address space is partitioned into pages. Pages that are marked read-only can have copies residing in the physical memories of many processors at the same time. But a page currently being written can reside in only one processor's physical memory. If some processor wants to write a page that is currently residing on other processors, it must get an up-to-date copy of the page and then tell the other processors somehow to invalidate their copies.

The memory mapping manager views its local memory as a big cache of the shared virtual memory address space for its associated processor. Like the traditional virtual memory [Denning 80], the shared memory itself exists only *virtually*. A memory reference may cause a page fault when the page containing the memory location is not in a processor's current physical memory. When this happens, the memory mapping manager retrieves the page from either disk or the memory of another processor. This thesis discusses both centralized manager algorithms and distributed manager algorithms, and shows that a class of distributed manager algorithms can retrieve pages efficiently on page faults

while keeping the memory coherent.

A parallel program usually uses only one shared virtual memory address space by creating a number of processes (threads or tasks). These processes are lightweight—they share the same address space and the cost of a process context switch, process creation, or process termination is small [Levin 86]. For example, the cost of a process creation is the same as that of a few procedure calls. One of the important goals of the shared virtual memory is to get processes of a program to execute on different processors in parallel. To do so, the appropriate process manager and memory mapping manager must be integrated with the memory mapping manager, which requires a simple operating system on top of the shared virtual memory. The whole system is called a *shared virtual memory system.*

The process manager provides clients with a set of process control primitives and a set of traditional process synchronization primitives. The main goal of the process manager is to achieve process transparency, that is, any process can run on any processor and can migrate from one processor to another at run time. The process manager also does simple process scheduling. Since the communication cost of sending messages on loosely coupled multiprocessor is high, process scheduling is important. This thesis presents several process scheduling mechanisms and load balancing strategies.

The memory allocation manager takes care of dynamic memory allocation in the shared virtual memory address space. Although dynamic memory allocation for a uniprocessor environment has been studied since the beginning of operating systems development, dynamic memory allocation for the shared virtual memory is quite different because processes truly run in parallel. The main goal of the memory allcoation manager is to allocate memory efficiently while providing clients with a convenient interface. This dissertation shows how to modify a sequential memory allocation algorithm for this purpose.

In general, a shared virtual memory system presents clients with the same interface as a tightly coupled multiprocessor with a shared memory except

that the memory synchronization unit is a page. The shared virtual memory system allows application programs to use the memory in the same manner as a traditional virtual memory, except, of course, that processes can run on different processors in parallel.

The performance of parallel programs on a shared virtual memory system mainly depends on two things: the number of parallel processes (or threads) the degree of data sharing (or *granularity of parallelism* [1]). The shared virtual memory system will be effective if a parallel program can keep many processors busy and does not have any memory contention. The number of parallel processes determines the maximum number of processors that the program can use in parallel. The granularity of parallelism is related to the memory contention. A fine-grained parallel program does not necessaryly create memory contention because the share virtual memory allows a page to have multiple read copies so that read accesses to shared data will not create contention at all if the program exhibits *locality* of references.

One of the main justifications for the traditional virtual memory is that memory references in sequential programs generally have a high degree of locality [Denning 72, Denning 80]. Although memory references in parallel programs may behave differently from those in sequential ones, a single process is a sequential program, and should exhibit a high degree of locality. Contention among parallel processes for the same piece of data depends on the algorithm, but a common goal in designing parallel algorithms is to minimize such contention for optimal performance. The shared virtual memory attempts to use the behavior of parallel programs for MIMD multiprocessors to take advantage of the nature of loosely coupled multiprocessors, just as the traditional virtual memory system attempted to use the behavior of sequential programs to take advantage of the nature of random-access disk devices.

[1] A parallel program has fine-grained parallelism if its parallel processes access shared data frequently.

## 1.6 Prototype and Experiments

Since parallel programs are complex and the interactions between parallel processes are often unpredictable, the only convincing way to justify a shared virtual memory on loosely coupled multiprocessors is to implement a prototype system and run some realistic experiments on it. I have implemented a prototype shared virtual memory system called IVY (Integrated shared Virtual memory system developed at Yale). It is implemented on top of a modified Aegis operating system of the Apollo DOMAIN [Apollo 81, Leach 83]. The system can be used to run parallel programs on any number of processors on an Apollo ring network.

IVY consists of 5 modules: simple RPC, memory mapping, process management, memory allocation, and initialization. The last three modules form the IVY client interface. Each module consists of a set of primitives that can be used by application programs. Since IVY is an *integrated* system, the primitives are placed in a library file. One can produce an IVY image by compiling a program and binding it with the desired library. Such an image file can be executed on any number of nodes in the network.

I have written a set of benchmark parallel programs and run them on the prototype system. These benchmark programs represent a spectrum of likely practical parallel programs that have reasonably fine granularity of parallelism and side-effects in shared data structures. The benchmark programs consist of:

- a linear equation solver,
- a three-dimensional partial differential equation solver,
- sorting,
- dot-product,
- the traveling salesman problem, and
- matrix multiply.

I ran the programs on the prototype system and collected three kinds of statistical data: speedups, memory coherence algorithm comparison, and shared

virtual memory reference miss ratio. The experiments show that, in some cases, the shared virtual memory system can even yield super-linear speedups for the parallel programs because the system can exploit not only the power of the processors but also the power of the combined physical memories.

The results of these experiments strongly support my thesis that shared virtual memory on loosely coupled multiprocessors can achieve orders-of-magnitude speedups over a uniprocessor for parallel programs, and that it is practical to implement it on existing architectures.

## 1.7 Related Work

There is a large body of literature related to the research in this dissertation. Below is a review of some closely related work in virtual memory and parallel computing on loosely coupled multiprocessors.

Research on virtual memory management began in the 1960s [Denning 70] and has been an important topic in operating system design ever since. The research focused on the design of virtual memory systems for uniprocessors. An important observation was that sequential programs exhibit *locality* of reference [Denning 72]. The *Working set* was an important concept used in virtual memory design to maximizes the localities of programs [Denning 68, Denning 80].

A number of the early systems used memory mapping to provide access to different address spaces. The representative systems are Multics and Tenex [Daley 68, Bobrow 72]. In these systems, processes in different address spaces can share data structures in mapped memory pages. But the memory mapping design is for uniprocessors.

Spector proposed a *remote reference/remote operation model* [Spector 81, Spector 82] in which a master process on a processor performs remote references and a slave process on another processor performs remote operations. Using processor names as part of the address in remote reference primitives, this model allows a loosely coupled multiprocessor to behave in a way sim-

ilar to CM* [Fuller 78, Jones 79] or Butterfly [Larus 84] in which a shared memory is built from local physical memories in a static manner. Although implementing remote memory reference primitives in microcode can greatly improve efficiency, the cost of accessing a remote memory location is still several orders-of-magnitude more expensive than a local memory reference. The model is useful for data transfer in distributed computing, but it is unsuitable for parallel computing.

Among the distributed operating systems for loosely coupled multiprocessors, Apollo Aegis [Apollo 81, Leach 82, Leach 83] and Accent [Rashid 81, Fitzgerald 86] have had strong impact on the integration of virtual memory and interprocess communication. Both Aegis and Accent permit mapped access to data objects that can be located anywhere in a distributed system. Both of them view physical memory as a cache of virtual storage. Aegis uses mapped read and write memory as its fundamental communication paradigm. Accent has a similar facility called *copy-on-write* and a mechanism that allows processes to pass data *by value*. The data sharing between processes in these systems is limited at the object level, the system designs are for distributed computing rather than parallel computing.

Realistic parallel computing work on loosely coupled multiprocessors has been limited. Much work has focused on message passing [Finkel 85, Seitz 85, Cheriton 86]. It is possible to gain large speedups by message passing, but programming applications is difficult [Cheriton 86]. Furthermore, as mentioned above, message passing has difficulties in passing complicated data structures.

Another direction has been to use a set of primitives to access a global space that is used to store shared data structures of processes [Cheriton 86, Carriero 86a]. Although programming the global space does not require data movement as much as message passing, programmers still have to explicitly use the primitives. In a primitive global-space system, passing complex data structures and process migration are as difficult as in message passing systems, since accessing the data structures and process migration are by value or by name.

Furthermore, using primitives may greatly reduce the efficiency of parallel programs because a primitive operation requires at least one procedure call, which costs much more than a simple memory reference. If there is a primitive operation in an inner loop of a parallel program, the execution of such a program on one processor may be much slower than that of the best sequential program in which accessing the corresponding data structure is a simple memory reference. The shared virtual memory system would not have this problem because it presents clients with a real shared memory address space on which there is no need to use any primitive operations. So once the pages holding a data structure are paged in, accessing the data structure is the same as accessing it on a uniprocessor.

## 1.8 Organization of The Thesis

Chapters 2 and 3 are somewhat theoretical discussions concentrating on the algorithms for designing a shared virtual memory system. Chapter 2 discusses the algorithms for solving the memory coherence problem in the shared virtual memory mapping managers. Chapter 3 studies the issues and the algorithms for designing the process management unit in a shared virtual memory system.

Chapters 4 and 5 discuss how to implement a shared virtual memory system on existing machines. Chapter 4 is a detailed discussion on the engineering issues in implementing a shared virtual memory system on different types of existing loosely coupled multiprocessors. Chapter 5 describes an example, a prototype system implemented on an Apollo ring network of workstations. Readers who are interested in implementation but not in detailed engineering issues may skip Chapter 4. Readers with no interests in implementation may skip both chapters.

Chapter 6 presents the experimental results of a set of benchmark programs run on the prototype system. Finally, Chapter 7 summarizes the contribution this research makes to the field and proposes future research directions.

# Chapter 2

# Memory Coherence

The *shared virtual memory* provides a virtual address space that is shared among all processors in a loosely coupled multiprocessor system, as depicted graphically in Figure 2.1. The shared memory itself exists only *virtually*. Application programs can use it in the same way as a traditional virtual memory, except, of course, that processes can run on different processors in parallel.

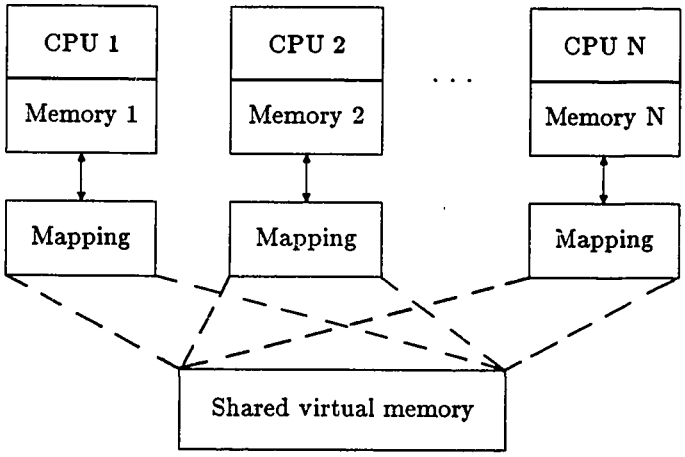


Figure 2.1: Shared virtual memory mapping.

Each processor has a memory mapping manager that not only "pages" data

between physical memories and disks, as in a conventional virtual memory system, but it also "pages" data between the physical memories of the individual processors. Thus data can naturally *migrate* between processors on demand. The main difficulty in building a shared virtual memory mapping manager is solving the *memory coherence problem*. This chapter presents, analyzes and compares a number of algorithms for solving this problem.

## 2.1 Memory Coherence Problem

A memory is *coherent* if the value returned by a read operation is always the same as the value written by the most recent write operation to the same physical address. A memory architecture with one access path in all the layers should have no coherence problems. A single access path, however, cannot satisfy today's high performance requirements. The coherence problem was introduced when caches appeared in uniprocessors (see [Smith 82] for a survey), and has become more complicated with the introduction of a multicache for shared memories on multiprocessor systems [Tang 76, Censier 78, Goodman 83, Thacker 84, Frank 84, Yen 85, Katz 85].

The memory coherence problem in a shared virtual memory system differs from that in multicache systems. A multicache multiprocessor system usually has a number of processors sharing a physical memory through their private caches. The connection between the caches and the memory unit is a fast bus. Since the size of a cache is relatively small, a sophisticated coherence protocol is usually implemented in the multicache hardware such that the time delay of conflict writes to a memory location is small. A shared virtual memory on a loosely coupled multiprocessor has no physically shared memory (it only *virtually* exists). The size of the private memory of a processor is as large as the shared virtual memory. So, the coherence problem in the shared virtual memory needs to be solved in software.

There are two cases that cause coherence problems in a shared virtual mem-

ory:

- A process on processor $i$ reads a piece of data $D$ without noticing that there is a copy of $D$ on processor $j$ that has been modified by another process on processor $j$.
- A process has a piece of data $D$ on processor $i$, the process migrates to processor $j$ and gets a copy of $D$, and then the process migrates back to processor $i$ after $D$ is modified.

These two cases can be merged into one if the problem is viewed in terms of processors:

- processor $i$ reads a piece of data $D$ without noticing that there is a copy of $D$ on processor $j$ which has been modified.

Coherence can be maintained if a processor is allowed to update a piece of data only while no other processor is updating or reading it. Under this constraint many processors can read a piece of data as long as no other processor is updating it; this is a form of the well-known readers/writers problem. It is obvious that the size of "a piece of data" can be arbitrary.

## 2.2 Design Choices for Memory Coherence

There are two design choices that greatly influence the implementation of a shared virtual memory: the granularity of the memory units, and the strategy for maintaining coherence.

### 2.2.1 Granularity

The size of the "memory units" that are to be coherently maintained is an important consideration in a shared virtual memory. This section gives several criteria for choosing this granularity.

In a typical loosely coupled multiprocessor system, sending large packets of data (say one thousand bytes) is not much more expensive than sending small

ones (say less than ten bytes) [Spector 81]. This similarity in cost is usually due to the typical software protocols and overhead of the virtual memory layer of the operating system. As a result, relatively large memory units are possible in the shared virtual memory.

On the other hand, the larger the memory unit, the greater the chance for contention. Memory contention occurs when two processors attempt to write to the same location (as in a shared memory system) as well as when two processors attempt to write to different locations in the same memory unit. Although clever memory allocation strategies might minimize contention by arranging concurrent memory accesses to locations in different memory units, such a strategy would lead to the inefficient use of memory space and introduce an inconvenience to the programmer. So, the possibility of contention indicates the need for relatively small memory units.

A suitable compromise in granularity is the typical *page* used in a conventional virtual memory implementation. The page sizes of today's computers vary, typically from 256 bytes to 8K bytes. If the page size on an existing system is small, choosing this size of a memory unit has several advantages. First, experience has shown that page size such as 1K bytes are suitable with respect to contention, and as mentioned above they should not impose undue communications overhead as long as a page can fit into a packet. In addition, such a choice allows us to use existing page fault schemes (hardware mechanisms) that allow single instructions to trigger page faults and trap to appropriate fault handlers. More precisely, a program can set the access rights to the pages in such a way that memory accesses that could violate memory coherence cause a page fault, and thus the memory coherence problem can be solved in a modular way in the page fault handlers. If the page size is larger than a packet size, the cost of sending a page may be too expensive for implementing a shared virtual memory. In this case, it is important to use special protocols to reduce the cost of page moving [Zwaenepoel 85].

One of the main justifications for virtual memory, of course, is that mem-

ory references in sequential programs generally have a high degree of *locality* of reference [Denning 72, Denning 80]. Although memory references in parallel programs may behave differently from those in sequential ones, a single process is a sequential program, and should exhibit a high degree of locality. Contention among parallel processes for the same piece of data depends on the algorithm, but a common goal in designing parallel algorithms is to minimize such contention for optimal performance. With the communication performance of today's loosely coupled architectures, it is plausible to implement a shared virtual memory by taking the page size of an existing system as the granularity.

### 2.2.2 Memory Coherence Strategies

The memory mapping managers in a shared virtual memory implement the mapping between local memories and the shared virtual memory address space. They keep the address space coherent at all times. The memory mapping manager views its local memory as a big cache of the shared virtual memory address space for its associated processor.

When a processor performs a read or write in the address space of a shared virtual memory, it may create a coherence problem. When this happens, the memory mapping manager retrieves the page from either disk or the memory of another processor. As mentioned earlier, page size granularity lets us use hardware page protection mechanisms to cause a fault when an invalid memory reference occurs, and thus resolve coherence problems in page-fault handlers. Therefore, our algorithms for solving the coherence problem are manifested as fault handlers, their servers, and the page tables on which they operate.

The data structure for maintaining the coherence has at least the following information about each page:

- *access* — indicating the accessibility to the page,
- *copy set* — containing the processor numbers that have read copies of the

page, and

- *lock* — for synchronizing multiple page faults by different processes on the same processor and for synchronizing remote page requests.

Following uniprocessor virtual memory convention, the data structure is called a *page table*. Usually every processor has a page table on it, but the same page entry in different page tables may be different.

It is helpful to first consider the spectrum of choices one has for solving the coherence problem. These choices can be classified by the way in which one deals with *page synchronization* and *page ownership*, as shown in Table 2.1.

| **Page synchronization method** | **Page ownership strategy** | | | |
|---|---|---|---|---|
| | Static | Dynamic | | |
| | | Centralized manager | Distributed manager | |
| | | | Fixed | Dynamic |
| Invalidation | *not appropriate* | *okay* | *good* | *good* |
| Writeback | *not appropriate* | *not appropriate* | *not appropriate* | *not appropriate* |

Table 2.1: Spectrum of solutions to the memory coherence problem.

### Page synchronization

There are two basic approaches to page synchronization: *invalidation* and *write-back*. Both approaches use a page as the memory synchronization unit to keep shared virtual memory coherent.

In the *invalidation* approach, there is only one owner processor for each page. This processor has either write or read access to the page. If a processor has a write fault, its fault handler will

- invalidate all the copies of the page that contains the faulting memory location,
- change the access of the page to write,
- move a copy the page to the processor if the processor does not have a copy of the page, and
- return to the faulting instruction.

After returning, the processor "owns" that page and can proceed with the write operation and other read or write operations until the page ownership is relinquished to some other processor. If a processor has a read fault, the fault handler will

- change the access of the page, that contains the faulting memory location, on the processor that has write access to read,
- move a copy of the page to the processor and set the access of the page to read, and
- return to the faulting instruction.

After returning, the processor can proceed with the read operation and other read operations to this page in the same way that normal local memory does until the page is given to someone else.

In the *writeback* approach, a processor treats a read fault just as it does in the *invalidation* approach. If a processor has a write fault, the fault handler will

- write to all copies of the page and
- return to the faulting instruction.

The philosophy of a shared virtual memory requires that pages be shared freely, but this means that *every* write to a shared page needs to generate a fault on the writing processor and update all copies.

It is possible to combine writeback and invalidation. This combination differs from the writeback approach in dealing with write faults, which has many variations in this class of algorithms. For example:

- If only one processor has a copy of the page that contains the the faulting memory location, then
  - move the page to the faulting processor,
  - set the access of the page on the faulting processor to write, and
  - invalidate the copy on the original processor,
- If the page has more than one copy, then update all the copies.
- Return to the faulting instruction.

This type of algorithm is better than the pure *writeback* ones because it is possible to have write access to a page so that some write operations to shared pages can proceed without generating faults. The combination scheme, however, does not overcome the difficulty of implementing updates, and still causes unnecessary faults for write operations to shared pages. Clearly doing these updates will be expensive, and algorithms using writeback do not seem appropriate for loosely coupled multiprocessors. Thus they are not further considered in this thesis, as indicated in Table 2.1.

### Page ownership

The ownership of a page can be handled either *statically* or *dynamically*. In the static approach, a page is always owned by the same processor. Other processors are never given full write access to the page; rather they must negotiate with the owning processor, and must generate a write fault every time they need to update the page. As with the writeback approach, static page ownership is an expensive solution for existing loosely coupled multiprocessors. Furthermore, it constrains desired modes of parallel computation. Thus in this thesis I only consider dynamic ownership strategies, as indicated in Table 2.1.

The strategies for maintaining dynamic page ownership can be subdivided into two classes: *centralized* and *distributed*. The processor that controls page

ownership is called the *page manager*, and thus there are centralized or distributed managers. Distributed managers can be further classified as either *fixed* or *dynamic*, referring to the distribution of ownership data.

The resulting combinations of strategies are shown in Table 2.1, where I have marked as inappropriate all combinations involving writeback synchronization or static page ownership. This thesis only considers the remaining choices—algorithms based on a centralized manager, a fixed distributed manager, and a dynamic distributed manager.

## 2.3 Communication Model and Cost Measurement

In order to study different algorithms and compare them with one another in a uniform way, this chapter uses a communication model and some cost measurement functions. The communication model assumed here characterizes both bus-like and ring-like communication links which are the most widely used network connections in today's loosely coupled multiprocessors.

The communication model consists of four elements: processor, sending queue, receiving queue, and communication link. A processor consists of a CPU and its local memory. The sending and receiving queues are message buffers interconnected with the communication link. The communication link is assumed to be a complete connection such that any processor can communicate with any other processor directly. Furthermore, it is possible to have a broadcast facility on the communication link so that any processor can broadcast a message to all other processors. Figure 2.2 shows an example configuration.

The sending queue always maintains the messages sent by the processor and sends them in order to their destinations. Similarly, the receiving queue queues the arriving messages and dispatches each message to its proper server. Both sending and dispatching messages are atomic operations so that messages sent

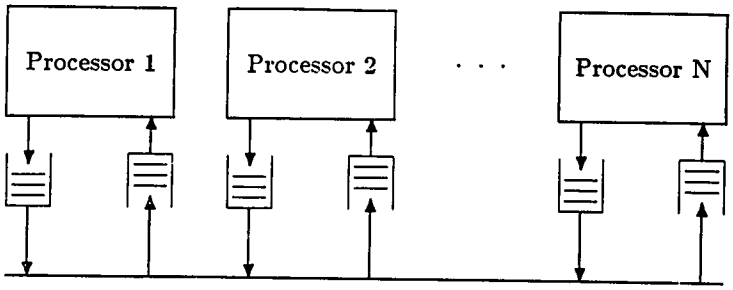


Figure 2.2: Communication model.

and received are strictly ordered. If processor $i$ sends messages $m_1$ and $m_2$ to processor $j$, then processor $j$ will receive $m_1$ and $m_2$ and dispatch them in order.

Algorithms should be evaluated based on their system performance. In other words, algorithm $a_1$ is better than algorithm $a_2$ if the system based on $a_1$ can finish parallel programs faster than the one based on $a_2$. Instead of running many parallel programs, this chapter evaluates algorithms by estimating the cost for each page fault and the cost spent on each processor. The estimation of the former gives a good idea about the response time of a page fault. The latter relates the balance of the work load in terms of page faults on different processors. Both parameters contribute directly to the throughput of a system.

Since the communication and faulting mechanism costs dominate the cost of local memory references on most loosely-coupled architectures, remote operations dominate the efficiency of an algorithm. This chapter assumes that any remote operation requires two messages, a request and a reply, and that a reliable communication protocol is used so that once a processor sends a request (no matter whether it is a point-to-point message, broadcast, or multicast), it will eventually receive a reply. For simplicity, the additional overhead of the reliable protocol is ignored.

The following notations are used in estimating the cost of sending messages and receiving messages:

- the cost of sending a message is denoted by $C_s$.
- the cost of receiving a message is denoted by $C_r$.
- the cost of an invalidation of $m$ copies is denoted by $C_v(m)$.
- the parallel cost of $C_x$ and $C_y$ is denoted by $C_x \oplus C_y$, which is not less than the maximum of $C_x$ and $C_y$ but less than $C_x + C_y$.

The parallel cost is used for sending broadcast or multicast messages. For instance, a multicast message is normally received by $k$ processors where $k > 1$. If all $k$ processors are idle, the cost of receiving the broadcast message is $C_r$. If some of them have ready processes, the cost of receiving the broadcast message is greater than $C_r$. The notion of $\oplus$ is a way of characterizing this vague cost measurement.

In the discussion of algorithms throughout this chapter, $C_{read}(i)$ denotes the the cost of a read page fault on processor $i$, and $C_{write}(i)$ denotes the cost of a write page fault. These two functions show the response time of a page fault. $C_{total}(i)$ denotes the total cost in dealing with page faults on processor $i$. This estimate follows from these definitions:

- $C_{inv}(i)$ is the total cost of invalidations on processor $i$.
- $f_r(i)$ is the total number of read page faults on processor $i$.
- $f_w(i)$ is the total number of write page faults on processor $i$.
- $r_p(i)$ is the total number of page fault requests received on processor $i$.
- $r_{inv}(i)$ is the total number of invalidation requests received on processor $i$.
- $r_{loc}(i)$ is the total number of page locating requests received on processor $i$.
- $s_{fwd}(i)$ is the total number of forwarding requests sent from processor $i$.

The estimates assume $N$ processors in the shared virtual memory.

## 2.4 More about Invalidation

### 2.4.1 Invalidation and Synchronization

*Invalidation* is probably the right approach to synchronizing pages in loosely coupled multiprocessor systems when building a shared virtual memory. Is it possible to do an invalidation efficiently with a traditional synchronization mechanism?

In a parallel program based on a global memory, accesses to shared data structures are synchronized by synchronization primitives. When no synchronization primitive is used, it is assumed that simultaneous memory accesses can be done in any possible order. Consider a scenario in which a processor has a write fault to a page with two read copies. Initially, both processor 1 and processor 2 have read access to the page. Assume processor 1 has a write fault which results in sending an invalidation message to processor 2. Before processor 2 receives the invalidation request, however, it does not know what is happening to the page; its processes can still read the page (depicted in the region between $a$ and $b$ in Figure 2.3). After the invalidation is complete, processor 1 has write access to the page, but processor 2 has no way of knowing what is in the page until there is a read fault or write fault that takes the page back.

This observation motivates us to wonder whether the invalidation operation can be deferred to the point where a write fault occurs, or in other words, whether the region between $a$ and $b$ on processor 2 can be eliminated. Such an elimination is called a *deferred invalidation*. The deferred invalidation can extend the time of read access of the read copies of a page so that parallel programs may get more parallelism (Figure 2.4). Obviously, the read access time on processor 2 in Figure 2.4 is longer than the one in Figure 2.3. The extended region between $a$ and $b$ can in fact be very long.

Unfortunately, this strategy is not correct in applications with traditional

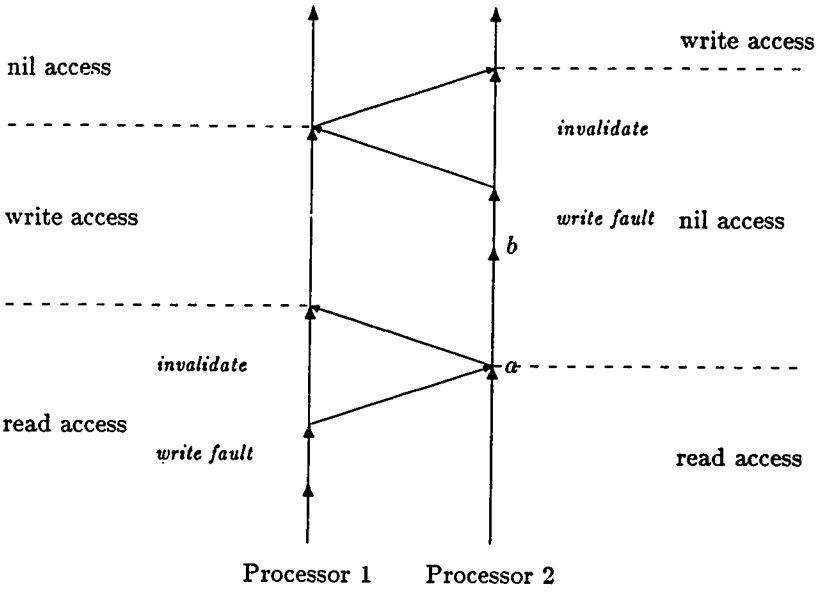


Figure 2.3: Invalidation of a page.

synchronization mechanisms. For example, it does not work in the consumer-producer problem. The following example shows why:

*Proccess 1:*

```
WHILE true DO BEGIN
    ...
  P( Sem );
    ...
  x := z;
    ...
  V( Sem );
    ...
  END;
```

*Process 2:*

```
WHILE true DO BEGIN
    ...
  P( Sem );
```

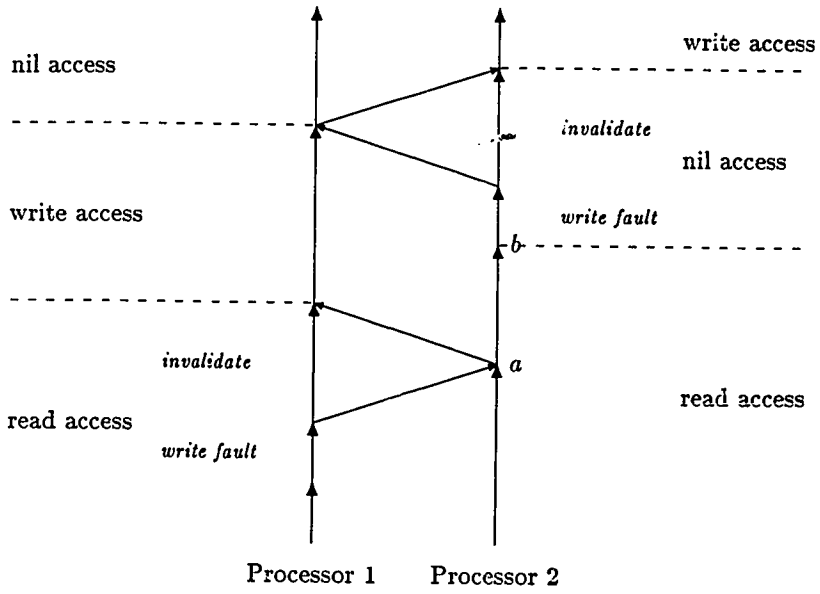


Figure 2.4: Deferred invalidation of a page.

```
    ...
y := x;
    ...
V( Sem );
    ...
END;
```

Suppose variable x is in the shared virtual memory. Process 2 uses x and process 1 assigns x on each iteration. When the two processes in the example above run on different processors, they access x in an arbitrary way. The synchronization of accessing x is done by a traditional synchronization mechanism known as P/V operations [Dijkstra 68]. Assume that the process scheduler in the system guarantees that in some finite amount of time, both loops get executed at least once (a commonly accepted "fairness" criterion of scheduling), then an observed computation sequence might be:

```
x := z;          (process 2)
x := z;          (process 2)
  ...              ...
x := z;          (process 2)
y := x;          (process 1)
```

The read of x will not get the most recent value of x, because process 2 may not generate a read page fault on the page where x resides if a deferred invalidation is performed. Thus, the idea of the deferred invalidation does not guarantee the memory coherence in this case.

Another idea to gain more parallelism is to make the invalidation operation asynchronous. The execution time of an asynchronous invalidation is shorter than a synchronous invalidation because the former does not need to wait for a reply. Figure 2.5 shows that an asynchronous invalidation on processor 1 takes less time than the one in Figure 2.3; and the region between *c* and *d* on processor 1 is the benefit gained by the asynchronous invalidation. The invalidation on processor 2 cannot be asynchronous because processor 2 needs to make a copy of the page which is carried back by the invalidation reply. This idea still does not work in the example above; process 1 may still use the stale value of x.

The failure of the two approaches to gaining more parallelism in invalidation does not mean that the observation is necessarily wrong, but means that traditional synchronization mechanisms are not appropriate. It is conceivable that one can invent a new synchronization mechanism to guarantee correctness. One possibility is to defer invalidation until synchronization points (P/V operations, I/O, context switches, and so on).

### 2.4.2 Invalidation Methods

There are at least three ways to invalidate the copies of a page: *individual*, *broadcast*, and *multicast*. The individual invalidation is just a simple loop:

*Invalidate:*

```
Invalidate( p, copy_set )
    FOR i in copy_set DO
```

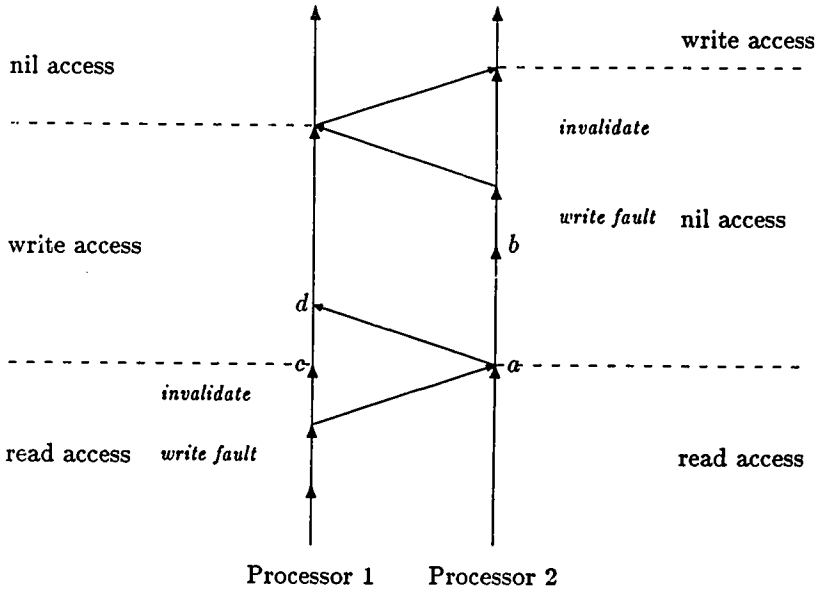


Figure 2.5: Asynchronous invalidation of a page.

```
send an invalidation request to processor i;
```

Broadcast invalidation does not need a copy set; it is just a simple broadcast message. Multicast invalidation depends on the multiprocessor interface. The server of the invalidation operation is simple:

*Invalidate server:*

```
PTable[ p ].access := nil;
```

For $m$ copies on an $N$ processor system, the cost of an invalidation can be

expressed as follows:

$$C_v(m) = \begin{cases} 0 & \text{m = 0,} \\ 2(m-1)(C_s + C_r) & \text{individual,} \\ mC_s + (m-1)C_r + \bigoplus_{i=1}^{N-1} C_r & \text{broadcast,} \\ mC_s + (m-1)C_r + \bigoplus_{i=1}^{m-1} C_r & \text{multicast.} \end{cases}$$

An individual invalidation sends $2(m-1)$ messages and receives $2(m-1)$ messages. A broadcast invalidation sends $m$ messages and receives $N+m-2$ messages of which $N-1$ messages are received in parallel. A multicast invalidation needs to send $m$ messages and receive $2(m-1)$ messages in which $(m-1)$ messages are received in parallel.

In general, multicast invalidation is the best approach. Unfortunately most of the loosely coupled systems do not have a multicast facility of this kind. Broadcast invalidation is expensive when $N$ is large.

## 2.5 Centralized Manager Algorithms

### 2.5.1 A Monitor-like Centralized Manager Algorithm

Our centralized manager is similar to a *monitor* [Hoare 74], consisting of a data structure and some procedures that provide mutually exclusive access to the data structure. The centralized manager resides on a single processor, and maintains a table called `Info` which has one entry for each page, each entry having three fields:

- The *owner* field contains the single processor that owns that page; namely, the most recent processor to have write access to it.
- The *copy set* field lists all processors that have copies of the page. This allows an invalidation operation to be performed without using broadcast.
- The *lock* field is used for synchronizing requests to the page, as will be described shortly.

Each processor also has a page table called PTable which has two fields: *access* and *lock*. This table keeps information about the accessibility of pages on the local processor.

In this algorithm, a page does not have a fixed owner, and there is only one manager that knows who the owner is. The owner of a page sends a copy to processors requesting a read copy. As long as a read copy exists, the page is not writable without an *invalidation* operation, which causes invalidation messages to be sent to all processors containing read copies. Since this is a monitor-style algorithm, it is easy to see that the successful writer to a page always has the truth of the page. When a processor finishes a read or write request, a *confirmation* message is sent to the manager to indicate completion of the request.

Both Info table and PTable have page-based locks. They are used to synchronize the local page faults (i.e. fault handler operations) and remote fault requests (i.e. server operations). When there is more than one process on a processor waiting for the same page, the locking mechanism prevents the processor from sending more than one request. Also, if a remote request for a page arrives and the processor is accessing the page table entry, the locking mechanism will queue the request until the entry is released.

The algorithm is characterized by fault handlers and their servers:

**Algorithm 2.1** *MonitorCentralManager*

```
Read fault handler:
    Lock( PTable[ p ].lock );
    IF I am manager THEN BEGIN
        Lock( Info[ p ].lock );
        Info[ p ].copyset
            := Info[ p ].copyset ∪ {ManagerNode};
        receive page p from Info[ p ].owner;
        Unlock( Info[ p ].lock );
        END;
    ELSE BEGIN
        ask manager for read access to p and a copy of p;
        send confirmation to manager;
```

```
        END;
    PTable[ p ].access := read;
    Unlock( PTable[ p ].lock );
```

*Read server:*

```
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        PTable[ p ].access := read;
        send copy of p;
        END;
    Unlock( PTable[ p ].lock );
    IF I am manager THEN BEGIN
        Lock( Info[ p ].lock );
        Info[ p ].copyset
            := Info[ p ].copyset ∪ {RequestNode};
        ask Info[ p ].owner to send copy of p to RequestNode;
        receive confirmation from RequestNode;
        Unlock( Info[ p ].lock );
        END;
```

*Write fault handler:*

```
    Lock( PTable[ p ].lock );
    IF I am manager THEN BEGIN
        Lock( Info[ p ].lock );
        Invalidate( p, Info[ p ].copyset );
        Info[ p ].copyset := {};
        Unlock( Info[ p ].lock );
        END;
    ELSE BEGIN
        ask manager for write access to p;
        send confirmation to manager;
        END;
    PTable[ p ].access := write;
    Unlock( PTable[ p ].lock );
```

*Write server:*

```
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        send copy of p;
        PTable[ p ].access := nil;
        END;
    Unlock( PTable[ p ].lock );
```

```
IF I am manager THEN BEGIN
    Lock( Info[ p ].lock );
    Invalidate( p, Info[ p ].copyset );
    Info[ p ].copyset := {};
    ask Info[ p ].owner to send p to RequestNode;
    receive confirmation from RequestNode;
    Unlock( Info[ p ].lock );
    END;
```

The *confirmation* message indicates the completion of a request to the manager, so that the manager can give the page to someone else. Together with the locking mechanism in the data structure, the manager synchronizes the multiple requests from different processors.

A read page fault on the manager processor needs 2 messages, one to the owner of the page, another from the owner. A read page fault on a non-manager processor needs 4 messages, one to the manager, one to the owner, one from the owner, and one for confirmation. Let us assume that processor 1 is the manager; then

$$C_{read}(i) = \begin{cases} 2(C_s + C_r) & \text{if } i = 1, \\ 4(C_s + C_r) & \text{if } i \neq 1. \end{cases}$$

A write page fault costs the same as a read page fault except that it includes the cost of invalidation:

$$C_{write}(i) = \begin{cases} 2(C_s + C_r) + C_v(m) & \text{if } i = 1, \\ 4(C_s + C_r) + C_v(m) & \text{if } i \neq 1 \end{cases}$$

where $m$ is the number of read copies of the faulting page.

The total paging cost of the algorithm on a single processor is approximated by:

$$C_{total}(i) = \begin{cases} 2(f_r(i) + f_w(i))(C_s + C_r) + C_{inv}(i) \\ \qquad + r_{inv}(i)C_r + r_p(i)(C_s + C_r) \\ \qquad + 2\sum_{j=2}^{N}(f_r(j) + f_w(j))(C_s + C_r) & \text{if } i = 1, \\ 2(f_r(i) + f_w(i))(C_s + C_r) + C_{inv}(i) + r_{inv}(i)C_r \\ \qquad + r_p(i)(C_s + C_r) & \text{if } i \neq 1. \end{cases} \quad (2.1)$$

Note the iterated sum term in the equation when $i = 1$. It means (not surprisingly) that a traffic bottleneck at the manager may occur as $N$ becomes large.

Since the centralized manager plays the role of helping other processors locate where a page is, the number of messages for locating a page is a measure of the complexity of the algorithm. When a non-manager processor has a page fault, it sends a message to the manager and gets a reply message from the manager, so the complexity of the algorithm is:

**Theorem 2.1** *The worst-case number of messages to locate a page in the centralized manager algorithm is two.*

Although this algorithm uses only two messages in locating a page, it requires a confirmation message whenever a fault appears on a non-manager processor. Eliminating the *confirmation* operation is the motivation for the following improvement to this algorithm.

### 2.5.2 An Improved Centralized Manager Algorithm

The primary difference between the improved centralized manager algorithm and the previous one is that the synchronization of page ownership has been moved to the individual owners, thus eliminating the *confirmation* operation to the manager. The locking mechanism on each processor now deals not only with multiple local requests, but also with remote requests. The manager still answers the question of where a page owner is, but it no longer synchronizes requests.

To accommodate these changes, the data structure of the manager must change. Specifically, the manager no longer maintains the copy set information, and a page-based lock is no longer needed. The information about the ownership of each page is still kept in a table called `Owner`, but an entry in the `PTable` on each processor now has three fields: *access, lock*, and *copy set*. The copy set field in an entry is *valid* if and only if the processor that holds the page

table is the owner of the page.

The fault handlers and servers for this algorithm are as follows:

**Algorithm 2.2** *CentralManager*

*Read fault handler:*

```
Lock( PTable[ p ].lock );
IF I am manager THEN
    receive page p from owner[ p ];
ELSE
    ask manager for read access to p and a copy of p;
PTable[ p ].access := read;
Unlock( PTable[ p ].lock );
```

*Read server:*

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
    PTable[ p ].copyset
        := PTable[ p ].copyset ∪ {RequestNode};
    PTable[ p ].access := read;
    send p;
    END
ELSE IF I am manager THEN BEGIN
    Lock( ManagerLock );
    forward request to owner[ p ];
    Unlock( ManagerLock );
    END;
Unlock( PTable[ p ].lock );
```

*Write fault handler:*

```
Lock( PTable[ p ].lock );
IF I am manager THEN
    receive page p from owner[ p ];
ELSE
    ask manager for write access to p and p's copyset;
Invalidate( p, PTable[ p ].copyset );
PTable[ p ].access := write;
PTable[ p ].copyset := {};
Unlock( PTable[ p ].lock );
```

*Write server:*

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
```

```
        send p and PTable[ p ].copyset;
        PTable[ p ].access := nil;
        END
    ELSE IF I am manager THEN BEGIN
        Lock( ManagerLock );
        forward request to owner[ p ];
        owner[ p ] := RequestNode;
        Unlock( ManagerLock );
        END;
    Unlock( PTable[ p ].lock );
```

Although the synchronization responsibility of the original manager has moved to individual processors, the functionality of the synchronization remains the same. For example, consider a scenario in which two processors $P_1$ and $P_2$ are trying to write into the same page owned by a third processor $P_3$. If the request from $P_1$ arrives at the manager first, the request will be forwarded to $P_3$. Before the paging is complete, suppose the manager receives a request from $P_2$, then forwards it to $P_1$. Since $P_1$ has not received ownership of the page yet, the request from $P_2$ will be queued until $P_1$ finishes paging. Therefore, both $P_1$ and $P_2$ will receive access to the page in turn.

The cost of a read page fault is:

$$C_{read}(i) = \begin{cases} 2(C_s + C_r) & \text{if } i = 1, \\ 3(C_s + C_r) & \text{if } i \neq 1. \end{cases}$$

Compared with the cost of a read page fault in the monitor-like algorithm, this algorithm saves one send and one receive per fault on the non-manager processors. The cost of a write page fault is similar:

$$C_{write}(i) = \begin{cases} 2(C_s + C_r) + C_v(m) & \text{if } i = 1, \\ 3(C_s + C_r) + C_v(m) & \text{if } i \neq 1 \end{cases}$$

where $m$ is the number of read copies of the faulting page.

The total paging cost on processor $i$ is defined by:

$$C_{total}(i) = \begin{cases} 2(f_r(i) + f_w(i))(C_s + C_r) + C_{inv}(i) \\ \quad + f_v(i)C_r + f_t(i)(C_s + C_r) \\ \quad + \sum_{j=2}^{N}(f_r(j) + f_w(j))(C_s + C_r) & \text{if } i = 1, \\ 2(f_r(i) + f_w(i))(C_s + C_r) + C_{inv}(i) \\ \quad + f_v(i)C_r + f_t(i)(C_s + C_r) & \text{if } i \neq 1. \end{cases} \tag{2.2}$$

The iterated sum term in the equation when $i = 1$ is a factor of 2 less than the one in the monitor-like centralized manager algorithm (Equation 2.1), an obvious improvement.

Decentralizing the synchronization improves the overall performance of the shared virtual memory, but for large $N$ there still might be a bottleneck at the manager processor, because it must respond to every page fault.

## 2.6 Distributed Manager Algorithms

In the centralized manager algorithms described in the previous section, there is only one manager for the whole shared virtual memory. Clearly such a centralized manager can be a potential bottleneck. This section discusses several ways of distributing the managerial task among the individual processors.

### 2.6.1 A Fixed Distributed Manager Algorithm

In a *fixed* distributed manager scheme, every processor is given a predetermined subset of the pages to manage. The primary difficulty in such a scheme is choosing an appropriate mapping from pages to processors. The most straightforward approach is to distribute pages evenly in a fixed manner to all processors. For example, suppose there are $M$ pages in the shared virtual memory, and that $I = \{1, \ldots, M\}$. An appropriate mapping function $H$ could then be defined by:

$$H(p) = p \bmod N \tag{2.3}$$

where $p \in I$ and $N$ is the number of processors. A more general definition is:

$$H(p) = \left(\frac{p}{s}\right) \bmod N \tag{2.4}$$

where $s$ is the number of pages per *segment*. Thus defined, this function distributes manager work by segments.

Another variation is to use a suitable hashing function or to provide a default mapping function that clients may override by supplying their own mapping. In this way, the map could be tailored to the data structure in the application and the expected behavior of concurrent memory references.

With this approach there is one manager per processor, each responsible for the pages specified by the static mapping function $H$. When a fault occurs on page $p$, the faulting processor asks processor $H(p)$ where the true page owner is, and then proceeds as in the centralized manager algorithm.

The total cost for page faults on processor $i$ is given by:

$$\begin{aligned} C_{total}(i) = 2(f_r(i) + f_w(i))(C_s + C_r) + C_{inv}(i) + r_{inv}(i)C_r \\ +r_p(i)(C_s + C_r) + r_{loc}(i)(C_s + C_r). \end{aligned} \tag{2.5}$$

If the distribution function $H(p)$ fits application programs well, then one can assume that:

$$\min_{1 \le i \le N} r_{loc}(i) \simeq \max_{1 \le i \le N} r_{loc}(i).$$

Thus we have the following approximation:

$$r_{loc}(i) \simeq \frac{1}{N}\sum_{j=1}^{N}(f_r(j) + f_w(j)).$$

Note that $C_{total}(i)$ in equation 2.5 is greater than $C_{total}(i)$ in equation 2.1 and 2.2 when $i \neq 1$, but it is much less when $i = 1$. The changing of $C_{total}(i)$ value reflects the transfer of traffic on processor 1 to other processors by the distribution function, thus alleviating the bottleneck.

Our experiments have shown that the fixed distributed manager algorithm is substantially superior to the centralized manager algorithms when a parallel

program exhibits a high rate of page faults. However, it is difficult to find a good static distribution function that fits all applications well. Indeed, for any given function it is always possible to find a pathological case that produces performance no better than the centralized scheme. So, we would like to investigate the possibility of distributing the work of managers *dynamically.*

### 2.6.2 A Broadcast Distributed Manager Algorithm

An obvious way to eliminate the centralized manager is to use a *broadcast* mechanism. With this strategy, each processor manages precisely those pages that it owns, and faulting processors send broadcasts into the network to find the true owner of a page. Thus the `Owner` table is eliminated completely, and the information of ownership is stored in each processor's `PTable`, which, in addition to *access, copy set* and *lock* fields, has an *owner* field.

More precisely, when a read fault occurs, the faulting processor $P$ sends a *broadcast read request*, and the true owner of the page responds by adding $P$ to the page's *copy set* field and sending a copy of the page to $P$. Similarly, when a write fault occurs, the faulting processor sends a *broadcast write request*, and the true owner of the page gives up ownership and sends back the page and its *copy set*. When the requesting processor receives the page and the *copy set*, it invalidates all copies.

The fault handlers and servers for such a naive algorithm are given below:

**Algorithm 2.3** *BroadcastManager*

*Read fault handler:*

```
Lock( PTable[ p ].lock );
broadcast to get p for read;
PTable[ p ].access := read;
Unlock( PTable[ p ].lock );
```

*Read server:*

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
    PTable[ p ].copyset :=
```

```
            PTable[ p ].copyset ∪ [ RequestNode ];
        PTable[ p ].access := read;
        send p;
        END;
    Unlock( PTable[ p ].lock );

Write fault handler:
    Lock( PTable[ p ].lock );
    broadcast to get p for write;
    Invalidate( p, PTable[ p ].copyset );
    PTable[ p ].access := write;
    PTable[ p ].copyset := {};
    PTable[ p ].owner := self;
    Unlock( PTable[ p ].lock );

Write server:
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        send p and PTable[ p ].copyset;
        PTable[ p ].access := nil;
        END;
    Unlock( PTable[ p ].lock );
```

The simplicity of this approach is appealing. Yet, the correctness of the algorithm is not obvious at first. Consider the case in which two write faults to the same page happen simultaneously on two processors $P_1$ and $P_2$. When the owner of the page, $P_3$, receives a broadcast request from $P_1$, it gives up its ownership but $P_1$ has not yet received the message granting ownership. At this point, $P_2$ sends its broadcast request and there is no owner. But this is not a problem because $P_2$'s message is queued on $P_1$ waiting for the lock on the page table entry; after $P_1$ receives ownership, the lock will be released, and $P_2$'s message will then be processed by $P_1$.

A read page fault causes a broadcast request that will be recieved by $N-1$ processors but replied to by only one of them. The cost for a read page fault is:

$$C_{read}(i) = 2C_s + C_r + \bigoplus_{i=1}^{N-1} C_r.$$

The cost for a write page fault is the same, plus the overhead of an invalidation:

$$C_{write}(i) = 2C_s + C_r + C_v(m) + \bigoplus_{i=1}^{N-1} C_r.$$

The total cost of page faults on processor $i$ is given by:

$$C_{total}(i) = (f_r(i) + f_w(i))C_s + C_{inv}(i) + r_{inv}(i)C_r + r_p(i)C_s + \sum_{j=1}^{N}(f_r(j) + f_w(j))C_r. \quad (2.6)$$

Since the iterated sum dominates the total cost, the work on all processors is fairly balanced in this algorithm. For large $N$, of course, this algorithm performs poorly because all processors have to respond to each broadcast request, slowing down the computation speed on all processors.

My experiments show that the cost introduced by the broadcast requests is substantial when $N = 4$. A parallel program with many read and write page faults will not perform well on a shared virtual memory system based cn a broadcast distributed manager algorithm.

### 2.6.3 A Dynamic Distributed Manager Algorithm

The heart of a dynamic distributed manager algorithm is to keep track of the ownership of all pages in each processor's local `PTable`. To do this, the *owner* field is replaced with another field, *probOwner*, whose value can be either *nil* or the "probable" owner of the page. The information that it contains is just a hint; it is not necessarily correct at all times, but if incorrect it will at least provide the beginning of a sequence of processors through which the true owner can be found. Initially, the *probOwner* field of every entry on all processors is set to some default processor that can be considered the initial owner of all pages. It is the job of the page fault handlers and their servers to maintain this field as the program runs.

In this algorithm a page does not have a fixed owner or manager. When a processor has a page fault, it sends a request to the processor indicated by

the *probOwner* field for that page. If that processor is the true owner, it will proceed as in the centralized manager algorithm. If it is not, it will forward the request to the processor indicated by its *probOwner* field. When a processor forwards a request, it need not send a reply to the requesting processor. The protocol of forwarding requests will be discussed in Chapter 4.

The *probOwner* field changes on a write page fault as well as a read page fault. As with the centralized manager algorithms, a read fault results in making a copy of the page, and a write fault results in making a copy, invalidating other copies, and changing the ownership of the page. The *probOwner* field is updated whenever:

- a processor receives an invalidation request,
- a processor relinquishes ownership of the page, which can happen on a read or write page fault, or
- a processor forwards a page fault request.

In the first two cases, the *probOwner* field is changed to the new owner of the page. In the last case, the *probOwner* is changed to the original requesting processor, which will become the true owner in the near future. The algorithm is as follows:

**Algorithm 2.4** *DynamicDistributedManager*

*Read fault handler:*

```
Lock( PTable[ p ].lock );
ask PTable[ p ].probOwner for read access to p;
PTable[ p ].probOwner := self;
PTable[ p ].access := read;
Unlock( PTable[ p ].lock );
```

*Read server:*

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
    PTable[ p ].copyset
        := PTable[ p ].copyset ∪ {RequestNode};
    PTable[ p ].access := read;
    send p and PTable[ p ].copyset;
    PTable[ p ].probOwner := RequestNode;
```

```
        END
    ELSE BEGIN
        forward request to PTable[ p ].probOwner;
        PTable[ p ].probOwner := RequestNode;
        END;
    Unlock( PTable[ p ].lock );
```

*Write fault handler:*

```
    Lock( PTable[ p ].lock );
    ask PTable[ p ].probOwner for write access to page p;
    Invalidate( p, PTable[ p ].copyset );
    PTable[ p ].probOwner := self;
    PTable[ p ].access := write;
    PTable[ p ].copyset := {};
    Unlock( PTable[ p ].lock );
```

*Write server:*

```
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        PTable[ p ].access := nil;
        send p and PTable[ p ].copyset;
        PTable[ p ].probOwner := RequestNode;
        END
    ELSE BEGIN
        forward request to PTable[ p ].probOwner;
        PTable[ p ].probOwner := RequestNode;
        END;
    Unlock( PTable[ p ].lock );
```

*Invalidate server:*

```
    PTable[ p ].access := nil;
    PTable[ p ].probOwner := RequestNode;
```

The two critical questions about this algorithm are whether forwarding requests eventually arrive at the true owner and how many forwarding requests are needed. In order to answer these questions it is convenient to view all the *probOwners* of a page $p$ as a directed graph $G_p = (V, E_p)$ where $V$ is the set of processor numbers $1, \ldots, N$, $|E_p| = N$, and an edge $(i, j) \in E_p$ if and only if the *probOwner* for page $p$ on processor $i$ is $j$.

In the following, I first show some properties of the *probOwner* graph by

assuming that page faults are generated and processed sequentially. In other words, it is assumed that if processor $i$ has a fault on page $p$, then no other processor has a fault on page $p$ until processing the page fault on processor $i$ is complete. I then show the correctness of the concurrent page fault case by reducing an arbitrary concurrent page fault case to a sequential case.

**Lemma 2.1** *If page faults of page p occur sequentially, every probOwner graph* $G_p = (V, E_p)$ *has the following properties:*

1. *there is exactly one node i such that* $(i,i) \in E_p$;
2. *graph* $G'_p = (V, E_p - (i,i))$ *is acyclic; and*
3. *for any node x, there is exactly one path from x to i.*

*Proof:* By induction on the number of page faults on page $p$. Initially, all the *probOwner*s of the processors in $V$ are initialized to a default processor. Obviously, all three properties are satisfied.

After $k$ page faults, the *probOwner* graph $G_p$ satisfies the three properties as shown in Figure 2.6(a). There are two cases when a page fault occurs on processor $j$.

1. If it is a read page fault, the path from $j$ to $i$ is collapsed by the read fault handler and its server in the algorithm such that all the nodes on the path now point to $j$ (Figure 2.6(b)). The resulting graph satisfies 1 since $(i,i)$ is deleted from $E_p$ and $(j,j)$ is added into $E_p$. It satisfies 2 because the subgraphs $g_j, g_u, \ldots, g_v, g_i$ are acyclic and they are not changed. For any node $x \in V$, there was exactly one path to $i$. Suppose the first node in the path (in the node set $\{j, u, \ldots, v, i\}$) is $y$. The edge from $y$ is changed to $(y,j)$, so there is no other path from $y$ to $j$.
2. If it is a write page fault and there is no read-only copy of page $p$, then the resulting graph is the same as the read page fault case. If it is a write page fault and there are $r$ read-only copies on processor $v_1, \ldots, v_r$, in addition to collapsing the path from $j$ to $i$, the invalidation procedure makes nodes $v_1, \ldots, v_r$ point to $j$ (Figure 2.6(c)). The resulting graph satisfies 1, since $(i,i)$ is deleted from the graph and $(j,j)$ is added in. A

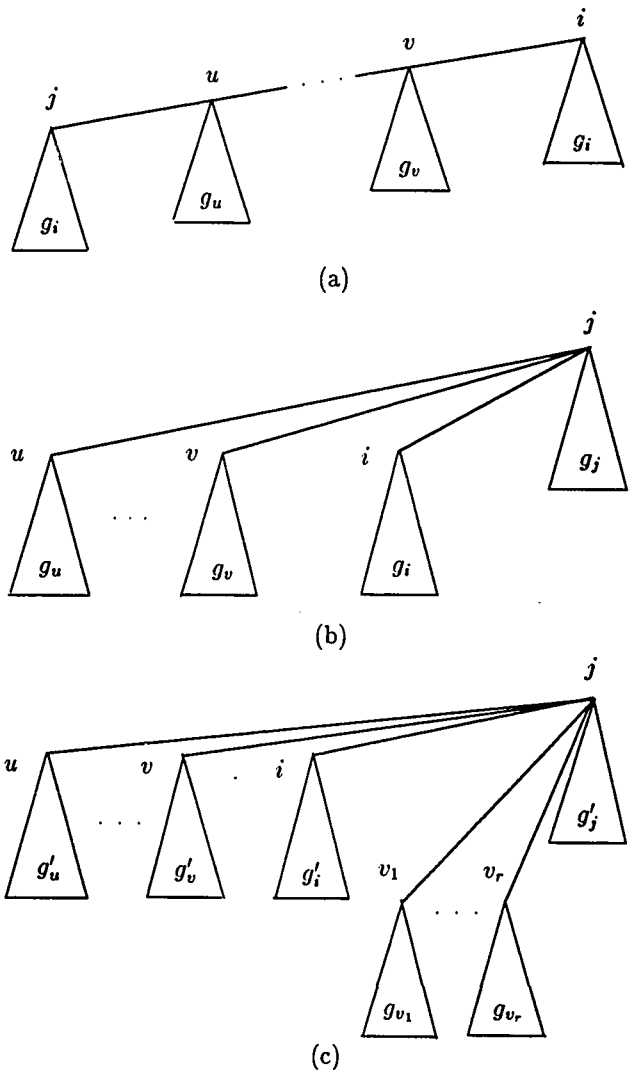


Figure 2.6: Induction of a *probOwner* graph.

subgraph $g_x$ is not equal to $g'_x$ only if there is a node $w$ in $g_x$ such that $w \in \{v_1, \ldots, v_r\}$. However, $g'_x$ is acyclic because making $w$ point to $j$ will not isolate the subgraph $g_w$ from $g_x$. Hence, the subgraphs $g'_j, g'_u, \ldots, g'_v, g'_i$ and $g_{v_1}, \ldots, g_{v_r}$ are acyclic. Thus, the resulting graph satisfies 2. Similar to the read fault case, any node $x \in V$ had exactly one path to $i$. Suppose that the first node in the path (in the node set $\{j, u, \ldots, v, i, v_1, \ldots, v_r\}$) is $y$. The edge from $y$ is changed to $(y, j)$, so there is no other path from $y$ to $j$.

□

Lemma 2.1 shows that any page fault can reach the true owner of the page if the page faults of the same page are processed sequentially. This shows the correctness of the algorithm in the sequential case.

The worst-case number of forwarding messages for the sequential case is given by the following corollary:

**Corollary 2.1** *In the $N$-processor shared virtual memory system, it will take at most $N-1$ messages to locate a page if page faults are processed sequentially.*

*Proof:* By Lemma 2.1, there is only one path to the true owner and there is no cycle in the *probOwner* graph. So, the worst-case occurs when the *probOwner* graph is a linear chain:

$$E_p = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{N-1}, v_N), (v_N, v_N)\}$$

in which case a fault on processor $v_1$ will generate $N-1$ forwarding messages in finding the true owner $v_N$. □

At the other extreme, we can state the following best-case performance (which is better than any of the previous alorithms):

**Lemma 2.2** *There exists a probOwner graph and page fault sequence such that the total number of messages for locating $N$ different owners of the same page is $N$.*

*Proof:* Such a situation exists when the a *prob_owner* graph is the same chain that caused the worst-case performance in Corollary 2.1. □

It is interesting that the worst-case single-fault situation is coincident with the best-case $N$-fault situation. Also, once the worse-case situation occurs, *all* processors know the true owner. The immediate question that now arises is what is the *worst-case* performance for $K$ faults to the same page in the sequential case. The following lemma answers the question:

**Lemma 2.3** *For an $N$-processor shared virtual memory, using the dynamic distributed manager algorithm, the worst-case number of messages for locating $K$ owners of a single page is $O(N + K \log N)$ in the sequential case.*

*Proof:* The algorithm reduces to the *type-0 reversal* find operation for solving the set union-find problem [Tarjan 84][1]. For a *probOwner* graph $G_p = (V, E_p)$, define the node set $V$ to be the set in the set union-find problem and the node $i \in V$ such that $(i, i) \in E_p$ to be the canonical element of the set. A read page fault of page $p$ on processor $j$ is then a type-0 reversal find operation $Find(j, V)$ in which the canonical element of the set is changed to $j$. A write page fault of page $p$ on processor $j$ is a type-0 reversal find operation plus the collapsing (by an invalidation) of the elements in the copy set of the page. Although the collapsing changes the shape of the graph, it does not increase the number in the find operation. According to the proof by Tarjan and Van Leeuwen [Tarjan 84], the worst-case number of messages for locating $K$ owners of a page is $O(N + K \log N)$. □

Note that without changing the ownership on a read page, the algorithm still works correctly, but the worst-case bound will be increased when $N$ is large. In that case, the total number of messages for locating $K$ owners depends on the configuration of the *probOwner* graph. If the graph is a chain, then it can be as bad as $O(KN)$. If the graph is a balanced binary tree, it will be $O(K \log N)$. If the graph is at a state in which every processor knows the owner, it will be $O(K)$.

All the lemmas and corollaries shown above are for the sequential page fault

[1]The reduction to set union-find was motivated by Fowler's analysis on finding objects [Fowler 86], though the reduction methods are different.

case. The problem is that in practice the sequential page fault case is unlikely to happen often, so it is necessary to study the concurrent page fault case. An important property in the algorithm is the atomicity of each fault handler or its server provided by the locking and unlocking mechanism. For convenience, we state it in the following lemma:

**Lemma 2.4** *If a page fault for page p traverses a processor, then other faults for p that need to travers the processor, but haven't yet, cannot be completed until the first fault completes.*

*Proof:* Suppose processing a page fault that occured on processor $i$ has traversed processor $u$. The server of the fault handler atomically set the *probOwner* field in the page table entry on processor $u$ to $i$. The requests for processing other page faults will be forwarded to processor $i$. Since the page table entry on processor $i$ is locked, these requests will be queued on processor $i$ until processing the page for processor $i$ is complete. □

My intention is to show that there exists a sequential page fault processing sequence that matches any given concurrent page fault processing so that the results for the sequential case can apply to the concurrent case. Processing concurrent page faults that occur on processor $v_1, \ldots, v_k$ (it is possible that $v_i = v_j$ when $i \neq j$) is said to be *matched* by a sequential processing of a $K$ page fault sequence $(v1, \ldots, v_k)$ if processing the page fault on processor $v_i$ in the concurrent case traverses the same processors in the same order as in the sequential processing case.

Consider an example in which the owner of a page is $v$ and page faults occur on processor $i$ and processor $j$ concurrently. Suppose that the first common node in the *probOwner* graph the requests for processing both page faults need to traverse is $u$ (Figure 2.7(a)). If the request from processor $i$ traverses processor $u$ first, the algorithm sets the *probOwner* field in the page table entry on processor $u$ to $i$. When the request from processor $j$ arrives at processor $u$, it will be forwarded to processor $i$, but the page table entry on processor $i$ is locked until processing the page fault for processor $i$ is complete. So, the

request from processor $j$ is queued at the page table entry of processor $i$, while the request from processor $i$ is traversing the rest of the path from $u$ to $v$. The *probOwner graph* when the processing is complete is shown in Figure 2.7(b). When the lock on processor $i$ is released, the request from processor $j$ continues. The resulting graph is shown in Figure 2.7(c). Thus, the request from processor $i$ traversed the path $i, \ldots, u, \ldots, v$ and the request from processor $j$ traversed the path $j, \ldots, u, i$. This is equivalent to the case in which the following events occur sequentially:

- a page fault occurs on processor $i$;
- the system processes the page fault;
- a page fault occurs on processor $j$; and
- the system processes the page fault.

The matched sequential page fault processing sequence is therefore $(i, j)$. Obviously, if the request from processor $j$ traverses processor $u$ first, then the matched sequential page fault processing sequence is $(j, i)$.

**Lemma 2.5** *For any concurrent page fault processing, there exists a matched sequential page fault processing sequence.*

*Proof:* By induction on the number of page faults. For one page fault processing, it is obviously true. For $k + 1$ page fault processings, we look at a page fault on processor $i$. By Lemma 2.4, the following is true:

1. $k_1$ page fault processing activities are done before processing the page fault for processor $i$ is complete;
2. there are $k_2$ page fault processing activities after processing the page fault for processor $i$ is complete; and
3. $k_1 + k_2 = k$.

According to the assumption, there is a matched sequential page fault sequence $(u_1, \ldots, u_{k_1})$ for the $k_1$ concurrent page fault processings and there is a matched sequential page fault sequence $(v_1, \ldots, v_{k_2})$ for the $k_2$ concurrent page

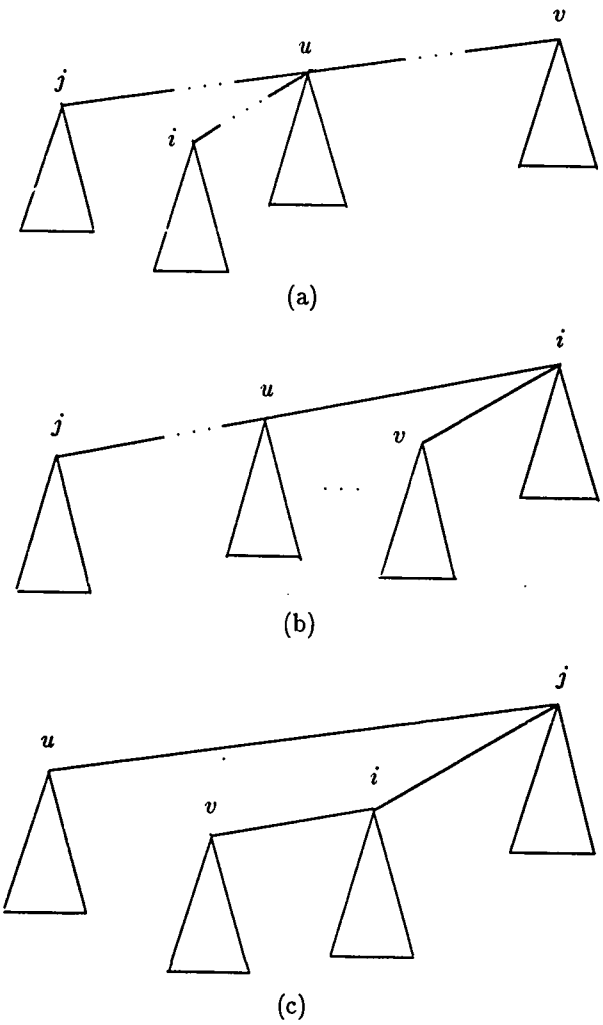


Figure 2.7: Two concurrent page faults.

fault processings. The matched page fault processing sequence is therefore $(u_1, \ldots, u_{k_1}, i, v_1, \ldots, v_{k_2})$. □

This lemma not only enables us to apply all the results for the sequential case to the general case, but also shows that the find operations in the set-union problem can be done in parallel if each find can be broken into small atomic operation in the same manner as this algorithm. The correctness of the algorithm in the general case is described by the following theorem:

**Theorem 2.2** *A page fault on any processor reaches the true owner of the page using at most $N-1$ forwarding request messages.*

*Proof:* By Lemma 2.5 and Corollary 2.1. □

The worst-case performance of $K$ page faults is then given by:

**Theorem 2.3** *For an $N$-processor shared virtual memory, using the dynamic distributed manager algorithm, the worst-case number of messages for locating $K$ owners of a single page is $O(N + K \log N)$.*

*Proof:* By Lemma 2.5 and Lemma 2.3. □

**Corollary 2.2** *Using the dynamic distributed manager algorithm, if $q$ processors are using a page, an upper bound on the total number of messages for locating $K$ owners of the page is $O(p + K \log q)$ if all contending processors are in the $q$ processor set.*

This is an important corollary, since it says that the algorithm does not degrade as more processors are added to the system, but rather degrades (logarithmically) only as more processors contend for the same page.

In the dynamic distributed manager algorithm, only the total cost of page faults on a processor is meaningful:

$$C_{total}(i) = (f_r(i) + f_w(i))(C_s + C_r) + C_{inv}(i) + r_{inv}(i)C_r + r_p(i)C_s + f_f(i)C_s. \tag{2.7}$$

The sum of $f_f(i)$, which is the total number of forwarding messages, is bounded

as follows:

$$\sum_{i=1}^{N} s_{fwd}(i) \leq c(\log N)\sum_{i=1}^{N}(f_r(i) + f_w(i))$$

where $c$ is a constant. The bound is a rather weak one because it ignores the fact that an invalidation operation can collapse all the read copy paths in the graph. Our experiments show that usually few processors are using the same page at the same time, that is, for a given time slot, $p << N$. This means that the right hand side of Equation 2.6.3 is a weak bound. This also means that the dynamic distributed manager algorithm has the potetial to implement a shared virtual memory system on a large-scale multiprocessor system.

Note that in Algorithm 2.4, the read fault handler and its server change the *probOwner* graph in the same way as the write fault handler and its server to the *probOwner* graph except that the write fault handler does invalidation according to the copyset field. The algorithm is designed for making proving theorems easier. For a real implementation, the read fault handler and its server should be slightly changed to get better performance:

*Read fault handler:*

```
Lock( PTable[ p ].lock );
ask PTable[ p ].probOwner for read access to p;
PTable[ p ].probOwner := ReplyNode;
PTable[ p ].access := read;
Unlock( PTable[ p ].lock );
```

*Read server:*

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
    PTable[ p ].copyset
        := PTable[ p ].copyset ∪ {RequestNode};
    PTable[ p ].access := read;
    send p to RequestNode;
    END
ELSE BEGIN
    forward request to PTable[ p ].probOwner;
    PTable[ p ].probOwner := RequestNode;
    END;
Unlock( PTable[ p ].lock );
```

The modified read fault handler and its server still compress the access path from the faulting processor to the owner, but they do not change the ownership of a page but only change its *probOwner* field. The modified algorithm reduces one message for each read page fault, an obvious improvement. For the modified algorithm, the worst case number of messages for locating $K$ owners of a single page is difficult to prove in a clean way because the read fault handler and its server behave the same as a pure forwarding address scheme which can be reduced to the set union-find problem of compressing path with naive linking [Fowler 86], while the write fault handler and its server behave differently. This is precisely why the modified algorithm was not presented in the first place.

The algorithm proposed in this section needs a broadcast or multicast facility only in the invalidation operation. If the invalidation is done by sending individual messages to the copy holders, there is no need to use the broadcast facility at all, while still reaping the benefits of the dynamic distributed manager algorithm.

### 2.6.4 An Improvement by Using Fewer Broadcasts

In the previous algorithm, at initialization or after a broadcast, all processors know the true owner of a page. The following theorem gives the performance for $K$ page faults on different processors in this case:

**Theorem 2.4** *After a broadcast request or a broadcast invalidation, the total number of messages for locating the owner of a page for $K$ page faults on different processors is $2K - 1$.*

*Proof:* This can be shown by the transition of a *probOwner* graph after a broadcast. The first fault uses 1 message to locate a page and after that every fault uses 2 messages. □

This theorem suggests the possibility of further improving the algorithm by enforcing a broadcast message (announcing the true owner of a page) after every $K$ page faults to a page. In this case, a counter is needed in each entry of the

page table, and is maintained by its owner. (It is interesting to note that when $K = 0$ this algorithm is functionally equivalent to the broadcast distributed manager algorithm, and when $K = N - 1$ it is equivalent to the unmodified dynamic distributed manager algorithm.) The algorithm is as follows:

**Algorithm 2.5** *DynamicDistributedBroadcast*

```
Read fault handler:
    Lock( PTable[ p ].lock );
    ask PTable[ p ].probOwner for read access to p
        and a copy of p;
    PTable[ p ].probOwner := ReplyNode;
    PTable[ p ].access := read;
    Unlock( PTable[ p ].lock );

Read server:
    Lock( PTable[ p ].lock );
    IF I am owner THEN BEGIN
        PTable[ p ].copyset
            := PTable[ p ].copyset ∪ {RequestNode};
        PTable[ p ].access := read;
        PTable[ p ].counter := PTable[ p ].counter + 1;
        send p to RequestNode;
        END
    ELSE BEGIN
        forward request to PTable[ p ].probOwner;
        PTable[ p ].probOwner := RequestNode;
        END;
    Unlock( PTable[ p ].lock );

Write fault handler:
    Lock( PTable[ p ].lock );
    ask PTable[ p ].probOwner for write access to p
        and p's copyset;
    Invalidate( p );
    PTable[ p ].probOwner := self;
    PTable[ p ].access := write;
    PTable[ p ].copyset := {};
    Unlock( PTable[ p ].lock );

Write server:
    Lock( PTable[ p ].lock );
```

```
    IF I am owner THEN BEGIN
        PTable[ p ].access := nil;
        send p, PTable[ p ].copyset, and PTable[ p ].counter;
        PTable[ p ].probOwner := RequestNode;
        END
    ELSE BEGIN
        forward request to PTable[ p ].probOwner;
        PTable[ p ].probOwner := RequestNode;
        END;
    Unlock( PTable[ p ].lock );

Invalidate( p ):
    IF ( PTable[ p ].counter > K )
            OR ( Size( PTable[ p ].copyset > L ) THEN
        broadcast invalidation;
    ELSE
        invalidate according to PTable[ p ].copyset;

Invalidate server:
    PTable[ p ].access := nil;
    PTable[ p ].probOwner := RequestNode;
```

Note the counter $L$ used in the invalidation procedure; whether a broadcast invalidation message is sent depends on whether the number of copies of a page reaches $L$. The value $L$ can be adjusted experimentally to improve system performance.

The total cost of page faults $C_{total}(i)$ in this variation of the dynamic distributed manager algorithm is the same as in the original algorithm except for the values of $r_{inv}(i)$ and $s_{fwd}(i)$. The function $r_{inv}(i)$ may be greater than before, but it is not big because a broadcast invalidation can be sent only after every $N$ page faults. It is given by:

$$\sum_{i=1}^{N} r_{inv}(i) \leq \sum_{i=1}^{N} (f_r(i) + 2f_w(i)).$$

In the case of Theorem 2.4, the function $s_{fwd}(i)$ is bounded by a small number:

$$\sum_{i=1}^{N} s_{fwd}(i) \leq \sum_{i=1}^{N} (f_r(i) + f_w(i)) - N$$

because it only shows the performance for $K$ page faults on *distinct* processors. On the average, without considering the cost of the broadcast message, this algorithm takes a little less than 2 messages to locate a page after a broadcast request or broadcast invalidation in this special case.

In order to choose the value of $K$, it is necessary to consider the general case in which the same processor may have any number of page faults. The above section has shown that the worst-case number of messages for locating $K$ owners for a single page is $O(N + K \log N)$, but our intuition says that the performance of $K$ page faults right after a broadcast message should be better because in the starting *probOwner* graph, all the processors know the true owner.

Since it is difficult to find a function that describing the relationship between $K$ and $N$ for the general case, I used two simulation programs to show the relationship in two different cases: approximated worst-case and approximated random-case. The initial *probOwner* graph used in both programs is the graph after a broadcast in which all the processors know the true owner. The programs record the number of messages used for each find (locating an owner) operation.

The first program approximates the worst case by choosing a node with the longest path to the owner at each iteration. Table 2.2 shows the average number of messages for each find operation for $K = N/4$, $K = N/2$, $K = 3N/4$, and $K = N$. For each $N$, the program starts with the same initial graph (the *probOwner* graph after a broadcast message). The table shows that the average number of messages steadily increases as $N$ gets large. Although picking the node with the longest path does not always generate the worst-case *probOwner* graph, my experiments show that the program actually converges when $K$ is very large. For example, the average number of messages becomes stable when $N = 64$ and $K > 1024$. Whether the converged case is the worst case is an open problem.

The second program approximates a random case by choosing a node randomly at each iteration. The average number of messages for each find opera-

| Number of nodes | Average number of messages / find | | | |
|---|---|---|---|---|
| ($N$) | $K = N/4$ | $K = N/2$ | $K = 3N/4$ | $K = N$ |
| 4 | 1 | 1.5 | 1.3 | 1.75 |
| 8 | 1.5 | 1.75 | 2 | 2.13 |
| 16 | 1.75 | 2.13 | 2.42 | 2.63 |
| 32 | 2.13 | 2.63 | 2.83 | 3 |
| 64 | 2.63 | 3 | 3.25 | 3.45 |
| 128 | 3 | 3.45 | 3.71 | 3.88 |
| 256 | 3.45 | 3.88 | 4.14 | 4.35 |
| 512 | 3.88 | 4.35 | 4.62 | 4.8 |
| 1024 | 4.35 | 4.8 | 5.05 | 5.263 |

Table 2.2: Longest-path-first finds

tion is shown in Table 2.3. In order to be objective, the table is produced by running the program four times and computing the average values among all the executions. Comparing Table 2.2 with Table 2.3, note that, on average, random finds spend less number of messages.

To choose an appropriate value of $K$, the two tables should be used together with the information about the average number of contending processors because the performance of the dynamic distributed manager algorithm is only related to such a number rather than the number of processors in the system (Corollary 2.2).

### 2.6.5 Distribution of Copy Sets

Note that in the previous algorithm, the *copy set* of a page is used only for the invalidation operation induced by a write fault. The *location* of the set is unimportant as long as the algorithm can invalidate the read copies of a page correctly. Further note that the *copy set* field of processor $i$ contains $j$ if processor $j$ copied the page from processor $i$, and thus the *copy set* fields for a

| Number of nodes | Average number of messages / find | | | |
|---|---|---|---|---|
| ($N$) | $K = N/4$ | $K = N/2$ | $K = 3N/4$ | $K = N$ |
| 4 | 1 | 1.5 | 1.67 | 1.75 |
| 8 | 1.5 | 1.75 | 1.99 | 2.08 |
| 16 | 1.75 | 1.96 | 2.22 | 2.53 |
| 32 | 1.93 | 2.39 | 2.79 | 2.9 |
| 64 | 2.09 | 2.78 | 2.9 | 3.12 |
| 128 | 2.06 | 2.68 | 2.8 | 3.16 |
| 256 | 2.2 | 2.77 | 3.18 | 3.39 |
| 512 | 2.46 | 3.09 | 3.32 | 3.56 |
| 1024 | 2.34 | 3.08 | 3.34 | 3.64 |

Table 2.3: Random finds

page are subsets of the original *copy set*.

These facts suggest an alternative to the previous algorithms in which the *copy set* data associated with a page is stored as a *tree* of processors rooted at the owner. In fact, the tree is bidirectional, with the edges directed from the root formed by the *copy set* fields, and the edges directed from the leaves formed by *probOwner* fields. The tree is used during faults as follows: A *read* fault collapses the path up the tree through the *probOwner* fields to the owner. A *write* fault invalidates all copies in the tree by inducing a wave of invalidation operations starting at the owner, propagating to the processors in its *copy set*, which in turn send invalidation requests to the processors in their *copy sets*, and so on.

The following algorithm is a modified version of the original dynamic distributed manager algorithm:

**Algorithm 2.6** *DynamicDistributedCopySet*

*Read fault handler:*

```
Lock( PTable[ p ].lock );
ask PTable[ p ].probOwner for read access to p;
```

```
        PTable[ p ].probOwner := ReplyNode;
        PTable[ p ].access := read;
        Unlock( PTable[ p ].lock );

    Read server:
        Lock( PTable[ p ].lock );
        IF PTable[ p ].access ≠ nil THEN BEGIN
            PTable[ p ].copyset
                := PTable[ p ].copyset ∪ {RequestNode};
            PTable[ p ].access := read;
            send p;
            END
        ELSE BEGIN
            forward request to PTable[ p ].probOwner;
            PTable[ p ].probOwner := RequestNode;
            END;
        Unlock( PTable[ p ].lock );

    Write fault handler:
        Lock( PTable[ p ].lock );
        ask PTable[ p ].probOwner for write access to p;
        Invalidate( p, PTable[ p ].copyset );
        PTable[ P ].probOwner := self;
        PTable[ p ].access := write;
        PTable[ p ].copyset := {};
        Unlock( PTable[ p ].lock );

    Write server:
        Lock( PTable[ p ].lock );
        IF I am owner THEN BEGIN
            PTable[ p ].access := nil;
            send p and PTable[ p ].copyset;
            PTable[ p ].probOwner := RequestNode;
            END
        ELSE BEGIN
            forward request to PTable[ p ].probOwner;
            PTable[ p ].probOwner := RequestNode;
            END;
        Unlock( PTable[ p ].lock );

    Invalidate server:
        IF PTable[ p ].access ≠ nil THEN BEGIN
```

```
Invalidate( p, PTable[ p ].copyset );
PTable[ p ].access := nil;
PTable[ p ].probOwner := RequestNode;
PTable[ p ].copyset := {};
END;
```

Since a write page fault needs to find the owner of the page, the lock at the owner synchronizes concurrent write page fault requests to the page. If read faults on some processors occur concurrently, the locks on the processors from which those faulting processors are copying synchronize the possible conflicts of the write fault requests and read fault requests. In this sense, the algorithm is equivalent to the original one.

Distributing *copy sets* in this manner improves system performance for the architectures that do not have a broadcast facility in two important ways. First, the propagation of invalidation messages is usually faster because of its "divide and conquer" effect. If the *copy set* tree is perfectly balanced, the invalidation process will take time proportional to $\log m$ for $m$ read copies. This faster invalidation response shortens the time for a write fault.

Secondly, and perhaps more importantly, a read fault now only needs to find a single processor (not necessarily the owner) that holds a copy of the page. To make this work, recall that a lock at the owner of each page synchronizes concurrent write faults to the page. A similar lock is now needed on processors having read copies of the page to synchronize sending copies of the page in the presence of other read or write faults.

Overall this refinement can be applied to any of the foregoing distributed manager algorithms, but it is particularly useful on a multiprocessor lacking a broadcast facility.

## 2.7 Conclusion

This chapter has studied two classes of algorithms for solving the memory coherence problem—the centralized manager and the distributed manager—

and both of them have many variations.

The centralized algorithm is straightforward and easy to implement, but it may have a traffic bottleneck at the central manager when there are many read and write page faults.

The fixed distributed manager algorithm alleviates the bottleneck, but, on average, a processor still needs to spend about two messages to locate an owner.

The dynamic distributed manager algorithm and its variations seem to have the most desirable overall features. As mentioned in the chapter, the dynamic distributed manager algorithm may only spend one message to locate an owner. On the other hand, when a distributed manager algorithm does not have such a situation, Theorem 2.4 shows that by using fewer broadcasts, the average number of messages for locating a page is a little less than two for a common case. Further refinement can also be done by distributing copy sets.

In general, dynamic distributed manager algorithms will outperform other methods when the number of processors sharing the same page for a short period of time is small, which is the normal case. The good performance of the dynamic distributed manager algorithms shows that it is possible to apply it to an implementation on a large-scaled multiprocessor.

# Chapter 3

# Process Management

The last chapter concentrated on the solutions to the memory coherence problem in implementing a shared virtual memory mapping. One of the important goals of such a mapping is to get processes of a program to execute on different processors in parallel. To do so, the appropriate process manager must be integrated with the memory mapping managers, which requires a simple operating system on top of the shared virtual memory.

Although the interface of such a simple operating system looks much like that of a traditional operating system for a uniprocessor, they are quite different. In a uniprocessor system, all processes share the same processor. In a shared virtual memory system, processes share a number of processors, which requires an effective processor-allocation strategy at compile time and run time.

Most of the work on process scheduling for tightly coupled multiprocessors with physically shared memories has focused on a centralized scheduling mechanism because the communication cost in a tightly coupled multiprocessor is low [Jones 79, Jones 80, Ousterhout 80, Emrath 85, Tuomenoksa 85]. Other work has assumed that the relationship between processes is known and can be used to find a scheduling [Muntz 70, Coffman Jr. 72, Coffman, Jr. 73, Gonzalez 78].

Most of the work on process scheduling for loosely coupled multiprocessors has concentrated on scheduling processes having different address spaces. Since

an efficient process migration implementation is hard to achieve for separate address spaces, most of the process scheduling algorithms are based on the assumption that once a process runs on one processor, it will not be migrated to another.

This chapter discusses the related data structures of processes, process migration, process scheduling mechanisms and dynamic load balancing strategies. The chapter describes a simple, uniform process management design—a distributed process scheduling mechanism with some simple load balancing strategies—that fits in various existing architectures and supports efficient process migration and effective process scheduling. Such a design is not only conceptually interesting but also easy to implement. Since process management is a classical topic, this chapter only focuses on the issues related to a shared virtual memory system on loosely coupled multiprocessors.

For simplicity, the discussion in the chapter assumes that there is only one address space in the whole shared virtul memory system. Once one knows how to deal with one address space, he can easily extend the technique to multiple address spaces. The issues of implementing multiple address spaces are discussed in chapter 4.

## 3.1 Process Control

### 3.1.1 Processes

A process is a concurrent execution unit of a program in the shared virtual memory system. It has an address space, a stack, a process control block (PCB), and a process identifier (PID).

The address space of a process consists of two portions, the shared memory portion and the private memory portion (Figure 3.1). The shared memory portion is shared by all the processes and its memory coherence is maintained by the shared virtual memory mapping managers described in the last chapter.

Memory heap and the stack of the process are usually stored in the shared memory portion. The private memory portion is used to store processor dependent data; it is shared by all processes on the same processor. The division of the two portions is determined at the system initialization stage.

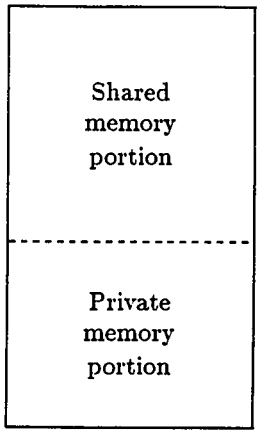


Figure 3.1: Process address space.

Like traditional operating systems, the data structure of a process for the operating system kernel is a PCB that records the information of a process, such as the layout of the process address space, register values, program counter, process state, and statistical information. A process needs at least three states: running, ready, and suspended.

All processes have unique PIDs in a shared virtual memory system. When a process control primitive in the system operates on some processes, it refers to them by PIDs. Since PIDs are used so often in process control primitives, it is important to have an efficient mapping between a PID and its associated PCB.

One way to generate unique PIDs for a shared virtual memory system is to use a semi-centralized PID generator similar to the sequencer proposed by Reed [Reed 79]. At the system initialization stage, the generator can generate

many PID's and buffer them on all the processors so that a process creator can usually find an unused PID without invoking the generator every time. The association between PIDs and PCBs is then established by building a mapping table that is replicated on all processors.

Another way to generate unique PIDs is to use the address of a PCB as its PID. If PCBs are stored in the shared memory, the mapping is done by:

$$\text{PID} = \text{address(PCB)}.$$

If PCB's are stored in the private memory, the (unique) processor number is needed in addition to the address of a PCB:

$$\text{PID} = (\text{processor}, \text{address(PCB)}).$$

This method eliminates the complexity of generating unique PIDs because the PID of a process is automatically generated at the time a piece of storage is allocated for its PCB. They are unique as long as the memory allocation works correctly.

In general, using the address of a PCB as a process PID is the more attractive of these two approaches because of its simplicity and efficiency. The drawback of using the address of a PCB is that a PID needs as many bits as an address in the shared virtual memory system (or the address together with a processor name if PCBs are stored in private memories); whereas the semi-centralized PID generator approach only requires $\log k$ bits where $k$ is the maximum number of processes allowed in the system. Of course, the semi-centralized PID generator approach requires building a PID mapping table, which requires space for the table and also adds complexity in maintaining it.

### 3.1.2 Primitive Operations

The client interface of process control is a set of primitives similar to those in a traditional operating system for a uniprocessor. There are two differences. One is that each process in the shared virtual memory system is *lightweight*, that is,

all processes share the same address space and the creation time of a process is as short as making a few procedure calls [Levin 86]. The other difference is that these lightweight processes can truly run on different processors in parallel and any process can migrate from one processor to another if necessary.

The process management needs two kinds of primitive operations: process control and process synchronization. The following is a small basic set of process control primitives:

- *Create* — Create a process. It is like a procedure call except that it creates a parallel thread. This operation should be no slower than a few procedure calls.
- *join* — Join a number of processes. This operation allows a process to suspend itself until the specified processes terminate.
- *migrate* — Move a process from one processor to another. This operation gives programmers control over processes at run time to allow a manually-controlled process distribution of a parallel program.
- *terminate* — Kill a process. This operation will notify all awaiting processes that execute a join operation.

The first two kinds of primitives, create and join, support the paradigm of **fork** and **join** [Dennis 66, Conway 63, Mitchell 79] or **cobegin** [1] and **coend**. They can also be used directly by programmers or by parallel programming language compilers to partition a problem into small, parallel, executable pieces and to synchronize them in a primitive way.

Synchronization primitives serve two main purposes: mutual exclusion of processes on shared data structures, and ordering of asynchronous events. Any one of the favored mechanisms based on global memory, such as semaphores [Dijkstra 68], monitors [Hoare 74], and eventcounts [Reed 79] may be sufficient. Synchronization mechanisms based on message passing can also be implemented easily in such a system because message passing can be simulated by shared

[1] This was first called **parbegin** [Dijkstra 68].

memory.

### 3.1.3 Process Migration

Process migration is the relocation of a process from one processor to another. Theoretical studies have shown that the performance of a system can be improved by process migrations if the cost of a process migration is not expensive [Stone 77, Bokhari 79, Robinson 79]. Several distributed operating systems have considered process migration [Finkel 80, Rashid 81]. DEMOS/MP actually has a process migration implementation based on message passing, but the cost of a process migration is expensive [Powell 83].

Message passing based systems have a high cost for process migration because of the separate address spaces. When migrating a process, all the operating system resources allocated by the process have to be moved together; the cost is that of moving all the process-related data via the communication link between the two processors. In the case where a process has a few opened ports and files, the pending messages and file access control blocks need to be transfered. Furthermore, the code and the stack of the process have to be moved because there is no easy way of translating the contents of different address spaces efficiently on the fly. To summarize, a process migration on a message passing system needs to:

1. create a PCB on the destination processor,
2. copy the code of the process to the destination processor,
3. allocate a stack on the destination processor and copy the current stack to it, and
4. copy all system data structures related to the process, such as file control blocks and message ports.

Figure 3.2 shows a process migration between separate address spaces.

Process migration in the shared virtual memory system is much simpler because all of the processes share the same address space. If resource control data

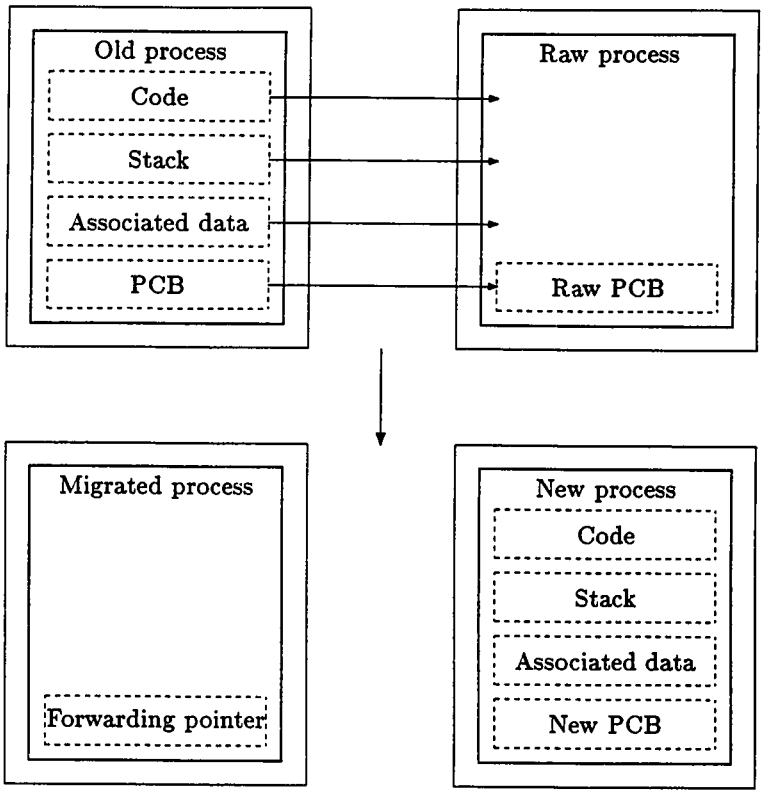


Figure 3.2: A process migration between address spaces.

structures are stored in the shared virtual memory, there is no need to move them because the memory pages keeping the data structures will be paged in as the migrated process uses them. A process migration is then just like a normal context switch except that the context will be restored on a remote processor. Figure 3.3 shows a process migration in the shared virtual memory system. Note that not all the processes need to migrate in the shared virtual memory, so only user processes created by using the process primitive are migratable.

Process migration in the shared virtual memory systems versus that in mes-

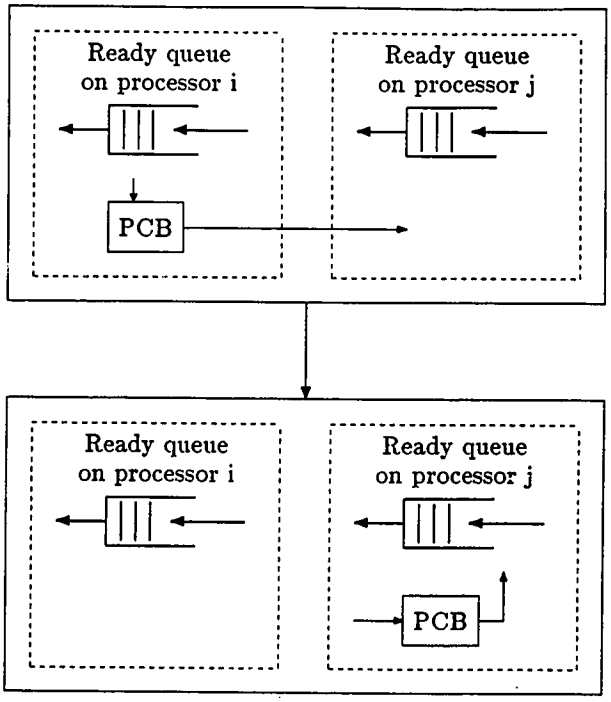


Figure 3.3: A process migration in a shared virtual memory system.

sage passing systems is analogous to lazy evaluation versus eager evaluation [Friedman 76, Henderson 76, Baker 78]. In the shared virtual memory system, a process migration never has to move more than necessary; whereas in message passing systems, one has to move everything eagerly because there is no underlying mechanism to support migration.

### 3.1.4 Process Scheduling

A process schedule is an assignment of which process should run on which processor at each moment of time. The basic objective of process scheduling for the shared virtual memory system is to schedule the processes of a program

so that the program excecutes in minimal time.

Unlike tightly coupled multiprocessor systems, the communication cost in the shared virtual memory system is relatively high; a process migration in the shared virtual memory system may increase the execution time of a computation. For example, when two processes need to write to the same memory page and both of them use it frequently, running the two processes on the same processor may take less time than on different processors because the cost of page moving may dominate the execution time. Process sychronization can also increase the execution time. For example, suppose process $p_1$ and process $p_2$ are running on processor $i$ and $j$ respectively, and $p_1$ is waiting for $p_2$ to terminate. The notification of $p_2$'s termination is a remote operation that requires two messages, a send and a reply. But if $p_1$ and $p_2$ are on the same processor, the cost of a notification is negligible. Therefore, the cost of synchronization between $p_1$ and $p_2$ depends on where they reside.

It is helpful first to consider the spectrum of process scheduling algorithms one may use for the shared virtual memory system. These algorithms may be classified by process scheduling *mechanisms* and process scheduling *disciplines*, as shown in Figure 3.1.

| **Scheduling disciplines** | **Scheduling mechanisms** | |
|---|---|---|
| | centralized | distributed |
| nonpreemptive | *not appropriate* | *not appropriate* |
| preemptive | *yes* | *yes* |

Table 3.1: Spectrum of Process scheduling.

There are two kinds of process scheduling mechanisms: centralized and dis-

tributed. A centralized mechanism has a main scheduler that allocates processors for processes. A distributed process scheduling mechanism has a scheduler on each processor. These schedulers need to negotiate with each other to schedule processes.

A process scheduling discipline can be either *preemptive* or *nonpreemptive*. It is preemptive if a process can be suspended and resumed under some constraint. It is nonpreemptive if once a process begins running, it runs until it suspends itself. The implementation of a nonpreemptive scheduling discipline is fairly straightforward.

It has been shown analytically that preemptive scheduling is more effective than non-preemptive (or processor-sharing) scheduling for multiprocessors [Muntz 70]. With non-preemptive scheduling, a process that has a page fault cannot proceed until it gets the page; preemptive scheduling would allow another process to run when a page fault occurs. The cost of a page fault is usually an order-of-magnitude more than a local context switch, and although it might be roughly the same as a remote context switch (a process migration), it may reduce page thrashing when more than two processes on different processors want the same page.

A preemptive process scheduling mechanism on a uniprocessor consists of four elements: a resource, a ready queue, a waiting queue, and a processing service (CPU) as shown in Figure 3.4. The scheduling algorithm has two policies: *preemptive policy* and *queuing policy*. The resource generates processes and puts them into the ready queue according to the queuing policy, such that the next process to be served is always at the head of the queue. The processing service may suspend the running process, put it into waiting queue, and resumes the process at the head of the ready queue. The processes in the waiting queue will be moved into the ready queue when they are ready to run.

The preemption policy determines when a process should be suspended, normally for one of two reasons: (1) resource request or (2) timeout. A resource request usually includes I/O request, process synchronization, and virtual mem-

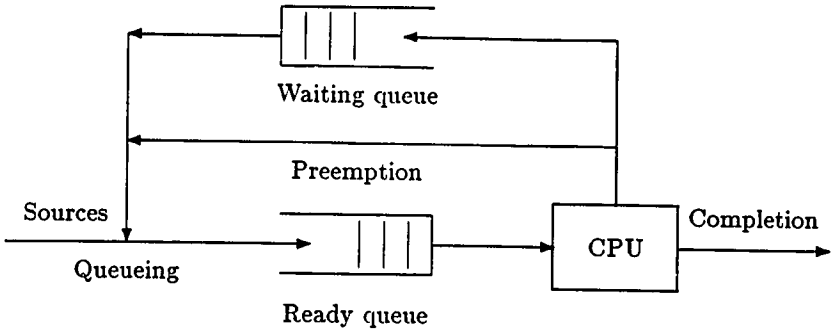


Figure 3.4: Preemptive scheduling on one processor.

ory paging. A timeout gives each process a time quota for each run, after which it will be suspended.

The queuing policy is the essence of a scheduling algorithm. The following is a list of some possible choices:

- dependency
- priority
- first-in-first-out (or round-robin)
- last-in-first-out
- smallest-quota-first
- longest-waiting-first

The *dependency* approach queues processes according to their dependencies such that process $p_1$ proceeds process $p_2$ if $p_2$ depends on $p_1$ or if they are independent. This policy can be used by compilers of parallel programming languages built on top of the shared virtual memory system. The *priority* approach assigns priorities to processes [Lampson 68]; the process with highest prority being put in the front of the queue. Since priority can be changed dynamically, this approach is flexible. *First-in-first-out, last-in-first-out, smallest-quota-first* and *longest-waiting-first* are relatively straightforward.

For a shared virtual memory system, load balancing is important because it is a way to reduce the execution time of a program. Load balancing can be done either at compile time, run time, or both. In compile-time load balancing, programmers or compilers allocate processors for processes statically. In run-time load balancing (also called dynamic load balancing), processors are allocated for ready processes at run-time. Effective run-time load balancing could complement compile-time balancing, especially when the compile-time balancing is not very good. Some problems might have quite different loads based on their input data, and thus need run-time balancing.

## 3.2 Centralized Process Scheduling

### 3.2.1 A Single Ready Queue Mechanism

Figure 3.5 shows a centralized process scheduler for a shared virtual memory system. The main scheduler has a single ready queue. The mechanism is similar to that for uniprocessors. The main difference being that there is more than one processing service in the system. The main scheduler allocates processors for ready processes. This mechanism is suitable for tightly coupled multiprocessors and has been used in many systems [Emrath 85].

The main scheduler has a data structure, *states*, keeping track of the states of all the processors. It is maintained by the local schedulers and the main scheduler. The local scheduler changes the state of a processor to "free" when its running process is suspended for some reason. When the main scheduler allocates a processor for a ready process, the processor state is changed to "busy."

When a process is suspended, its local scheduler will be invoked. When a suspended process becomes ready, it will be inserted into the ready queue according to the queuing policy. The main scheduler is invoked when:

- a processor becomes free,

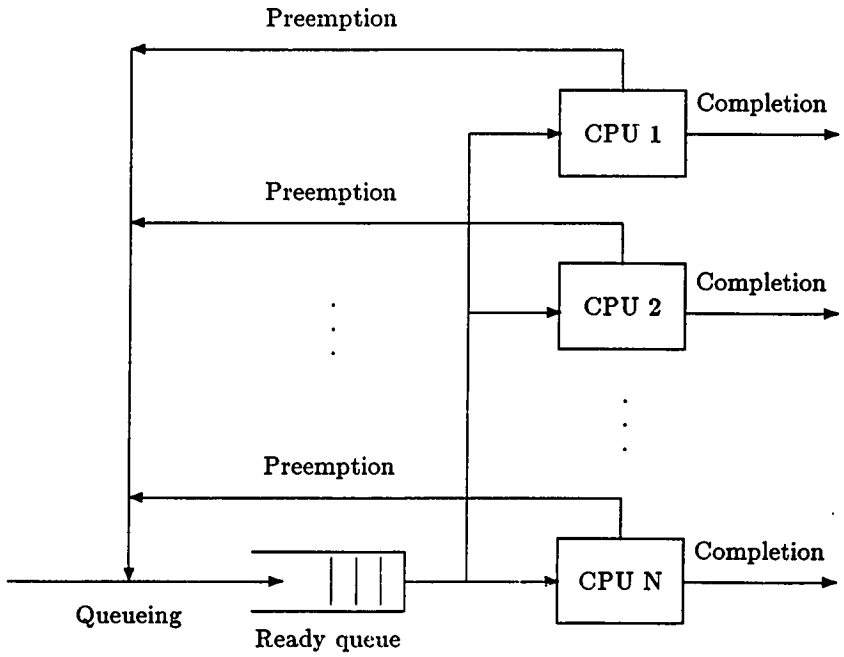


Figure 3.5: Centralized scheduling with one ready queue.

- a process becomes ready, or
- a time quota is filled.

The following is a simple scheduling strategy in which a local scheduler invokes the main scheduler when a processor becomes free.

**Algorithm 3.1** *SingleQueueSchedule*

*Local scheduler:*

```
states[ MyProcessor ] := free;
invoke main scheduler;
```

*Main scheduler:*

```
LOOP BEGIN
    wait for a free processor and non-empty ReadyQueue;
    pid := TakeFrom( ReadyQueue );
    i   := AllocateProcessor( pid );
```

```
IF i ≠ nil THEN
    Invoke( pid, i )
ELSE
    put pid back into ReadyQueue;
END;
```

Processor allocation may fail if there is no free processor. In order to allocate a processor optimally, the main scheduler may need the information about the recources the processors currently have and the resources the invoked process requires in the future. Since finding an optimal processor allocation is known to be NP-complete when the number of processors is greater than one [Karp 72], it is practical to use hueristic process scheduling strategies for large systems.

Here are some strategies for allocating processors:

- round-robin,
- original processor first,
- resource owner first, and
- priority.

*Round-robin* keeps the free processors in a queue and uses a first-in-first-out strategy to choose a free processor. *Original processor first* attempts to find the processor on which the process being invoked originally ran. *Resource owner first* attempts to invoke a process on the processor that has the resources the process needs. The *priority* method assigns each processor a priority that can be changed dynamically, and when invoking a process, the scheduler will choose the processor with the highest priority. Combinations of the strategies above are also worth considering.

As mentioned in Section 3.1.4, there are many possible queuing policies. Since processor allocation strategies are somewhat orthogonal to queuing policies, it is also possible to develop many scheduling algorithms for the centralized scheduling mechanism. The problem of which algorithm is the best for centralized process scheduling remains open.

Note that some of the remote process invocations involve process migrations and some of them do not. If a process is suspended on processor $i$ and invoked

on processor $j$, then the invocation causes a process migration if and only if $i \neq j$. Since there is only one ready queue and it is usually on the same processor as the main scheduler, all the process suspensions and invocations on other processors are remote operations.

### 3.2.2 A Multiple Ready Queue Mechanism

The previous scheduling mechanism can be improved by having multiple ready queues. A main scheduler still has control over the local schedulers in this mechanism, and the main scheduler still uses a main ready queue. But each processor now has not only its own local scheduler but also its own ready queue buffering some processes. Such a mechanism is shown in Figure 3.6.

Although the multiple ready queue mechanism has been theoretically studied [Corbato 62, Habermann 76] and used in uniprocessor systems and tightly coupled multiprocessor systems [Ritchie 78, Emrath 85, Tuomenoksa 85], applying it to shared virtual memory systems requires a somewhat different algorithm to reduce the communication cost. The data structure for processor information is still maintained by both the main scheduler and local schedulers. When the current running process on a processor is suspended, its local scheduler is invoked to get a ready process from its ready queue. If the local scheduler has a ready process in its ready queue, then both suspension and invocation will be local operations. If the local scheduler does not have any ready process to schedule, it will invoke the main scheduler. The ready queue maintained by the main scheduler buffers the processes to be moved to individual processors. The main scheduler may allocate several processes at a time to a processor.

The following is a possible scheduling strategy:

**Algorithm 3.2** *MultipleQueueSchedule*

*Local scheduler:*

```
IF LocalReadyQueue is not empty THEN BEGIN
    pid := TakeFrom( LocalReadyQueue );
    invoke( pid, MyProcessor );
```

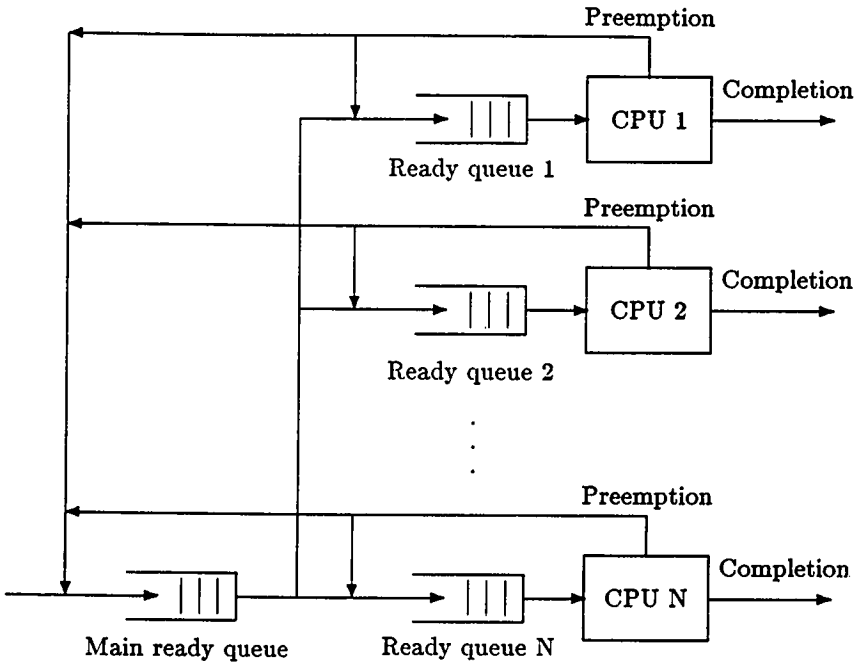


Figure 3.6: Centralized multiple queue scheduling mechanism.

```
        END
    ELSE BEGIN
        states[ MyProcessor ] := free;
        invoke Main scheduler;
        END;

Main scheduler:
    LOOP BEGIN
        wait for a free processor and a non-empty ReadyQueue;
        pid := TakeFrom( ReadyQueue );
        i   := AllocateProcessor( pid );
        IF i ≠ NIL THEN
            Invoke( pid, i )
        ELSE
```

```
        put pid back to ReadyQueue;
    END;
```

The maximum length of a local ready queue reflects the locality of processes; whereas the maximum length of the main ready queue can be considered as a control parameter of the frequency of load balancing. For example, the following is a possible strategy:

- the main scheduler only moves processes to a local ready queue whose length is less than the maximum length;
- the main scheduler is invoked when:
  1. a local ready queue is empty, or
  2. the main ready queue reaches its maximum length.

Both the maximum length of the main ready queue and the maximum length of the local ready queue can be set at the initialization stage. Various queuing policies can also apply to the scheduling mechansim.

An obvious advantage of the multiple ready queue mechanism is that the degree of centralization has been reduced a great deal if the average length of a ready queue is not small. Since the responsibility of process suspensions and invocations has been moved to local schedulers, many operations become local. Furthermore, local schedulings can be done in parallel. Obviously, the scheduling mechanism is much better than the one with a single ready queue.

## 3.3 Distributed Process Scheduling

### 3.3.1 A Distributed Scheduling Mechanism

A distributed process scheduling mechanism is like the multiple ready queue centralized scheduling mechanism, but without centralized control. Each processor has its own scheduler and a ready queue. There is no centralized scheduler or ready queue. The schedulers must negotiate with each other to do

things such as process migration. A distributed process scheduling mechanism is shown in Figure 3.7.

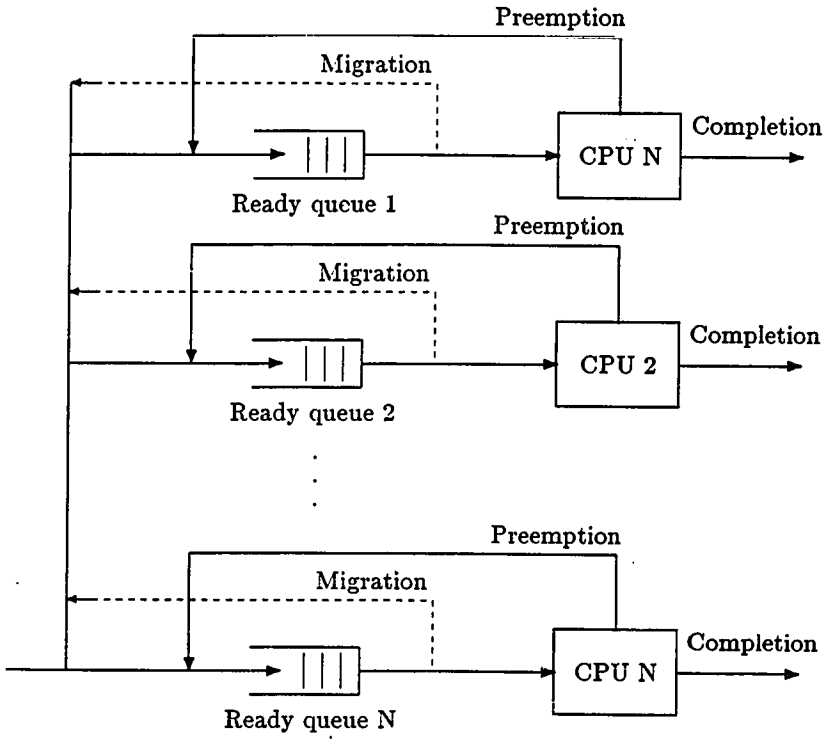


Figure 3.7: Distributed scheduling mechanism.

When a process negotiate with another, it can get an agreement or a rejection. If the schedule given by the distributed process scheduling mechanism is the same as the one given by the centralized process scheduling mechanism with multiple ready queues, the only way the distributed mechanism can beat the centralized one is to let the overhead of the maximal number of rejections be less than the overhead of the main scheduler in the centralized mechanism. This means that the goal is to find an algorithm using the distributed scheduling

mechanism that minimizes the number of rejections on all processors.

The problem is easy if the scheduler has the following information about other processors:

- the number of processes (ready processes and suspended processes),
- the number of ready processes,
- average workload within a specified time quota, and
- accessible memory pages.

The number of processes on each processor is the main factor because it represents the work load of the processor. Usually, processes are suspended by page faults, process synchronization primitives, and other resource requirements. The processes waiting for page faults will be awakened in a short time. The number of ready processes indicates whether there are processes available on the processor that can be migrated to gain parallelism. The accessible memory pages on each processor can be used to optimize decisions on load balancing, since a process migration may induce a great deal of page migration, which is more expensive than no page movement at all.

The information about the number of ready processes is important for minimizing the number of rejections. When two processors negotiate with each other about process migration, one processor must have at least one ready process that can be migrated. Therefore, when a scheduler with no ready process is asking for work, it needs to talk to the processor that has at least one ready process on it. Similarly, when a scheduler with ready processes is looking for help, it needs to find the processor without any ready processes.

With almost no extra effort, the processors can keep each other up to date on their current work loads by adding a few extra bits to the messages transmitted for the other shared virtual memory system operations [Ellis 85]. Usually, a half byte or a byte will be enough to transfer the information. The extra four bits or eight bits can be packed into every message at almost no extra cost.

With such a distributed process scheduling mechanism, it is possible to apply many kinds of scheduling strategies.

### 3.3.2 Active Load Balancing Strategy

Active load balancing is analogous to "data-driven" evaluation in programming languages [Baker 78, Hudak 85]. The main idea is that when a processor has many processes in its ready queue, its scheduler tries to ask another processor for help. If that processor agrees, it accepts the migrated process; otherwise, it sends a rejection message. When the scheduler gets a rejection, it has to try another processor. The schedulers actively try to do less work to balance the work load in the shared virtual memory.

The algorithms for the scheduler and its server are:

**Algorithm 3.3** *ActiveLoadBalance*

```
Scheduler:
    WHILE NeedHelp() DO BEGIN
        pid := TakeFrom( ReadyQueue );
        i := NextProcessor( pid );
        ask processor i to accept process pid;
        END;
    run a process;

Server:
    IF AcceptProcess() THEN
        put the process into ready queue
    ELSE
        send a reject reply;
```

In these algorithms each scheduler controls when to ask for help and the server of the scheduler on each processor controls when to accept processes from other processors.

An obvious algorithm for *NeedHelp* is to put an upper bound on the number of processes on the local processor. When the number of processes exceeds the upper bound, the function returns true. The function *AcceptProcess* can also be simple. It puts a lower bound on the number of processes on its processor; if the number is less than the lower bound, it returns true. This gives good performance when the processes in the application program are well balanced such that each process has similar execution time. However, if processes are

not well balanced or the processor speeds are different, such a straightforward approach does not give good performance.

More complicated algorithms can be developed in favor of different kinds of applications. Instead of putting thresholds on the number of processes, the scheduler and server can put thresholds on the number of ready processes, or on the number of processes and the number of ready processes at the same time. In addition, the memory requirement can be considered. For example, the "ask for help" request includes the information about which memory pages are needed by the migrated process. The function *AcceptProcess* can then determine how many memory pages are needed when the process is accepted. Since the balancing strategies are application dependent, it is hard to find a simple method giving optimal performance for all the applications.

There are many ways to find a processor when a scheduler asks for help. Such a processor must be able to get a true return when executing *AcceptProcess*. This means that the processor that may accept a process has no ready process at the requesting time and has less processes than other processors. The scheduler can find a possible processor by looking in the information table. In order to avoid concurrent requests, the search algorithm can be a function of the local processor number such that when a processor is the possible processor for more than one processor, the schedulers will not choose it at the same time. A round-robin jump search is one straightforward example of how this can be done.

### 3.3.3 Passive Load Balancing Strategy

Passive load balancing is analogous to "demand-driven" evaluation [Friedman 76, Henderson 76]. The idea is to migrate processes passively in the sense that when a scheduler does not have any ready process to run, it sends out a request to another scheduler to migrate a process. If the requested scheduler agrees, it migrates a process to the requesting scheduler; otherwise, it sends a rejection.

When a rejection arrives, the requesting scheduler needs to try another scheduler. The schedulers in the shared virtual memory system are so lazy that they do not want any processes unless they do not have anything to run.

The following is the algorithm for the scheduler and its server:

**Algorithm 3.4** *PassiveLoadBalance*

```
Scheduler:
    LOOP BEGIN
        IF ReadyQueue is not empty THEN
            run a process from ready queue;

        WHILE NeedMoreProcess() DO BEGIN
            i := NextProcessor();
            send a passive request to i;
            END;
        END;

Server:
    proc := FindProcess();
    IF proc ≠ nil THEN
        migrate proc
    ELSE
        send a reject reply;
```

When *NeedMoreProcess* returns true, the scheduler will ask a processor for a process to migrate over to run. The server decides if its local processor will give up a process.

Like the active load balancing algorithm, there are many algorithms for *NeedMoreProcess* and *FindProcess*. The function *NeedMoreProcess* can simply put a lower bound on the number of processes on the local processor. If the number is less than the lower bound, the function returns true. The function *FindProcess* can put upper bounds on the number of processes and the number of ready processes. If both of them are less than their bounds, a process will be returned.

### 3.3.4 Page-Demand Load Balancing Strategy

The page-demand load balancing strategy tries to reduce page thrashing in the shared virtual memory system. When two or more processors try to write to the same page at the same time, the page moves from one processor to another in order to complete the writes. Page thrashing occurs if the write operations to the same page are performed many times and the page has to move back and forth from one processor to another. For general purposes, page thrashing can be defined as a page moving to and from a processor $k$ times within a time period $t$. The values $k$ and $t$ depend on the performance of the target multiprocessor architecture.

Page thrashing can be discovered by using a fixed length list $L$. Each element of $L$ consists of the following fields:

- page number
- counter
- time stamp

The counter can be as small as $\lceil \log k \rceil$ bits and the time stamp needs precision fine enough to distinguish $t$. At the initialization stage, $L$ is empty. The following procedure will return true when page $p$ is thrashing. The procedure is called when a page fault occurs.

**Algorithm 3.5** *Detecter*($p$)

```
search L for the element e with e.PageNumber = p;
IF e = nil THEN BEGIN
    IF length( L ) = M THEN
        delete an element from tail of L;
    get a new element e;
    e.PageNumber := p;
    e.counter := 1;
    e.TimeStamp := CurrentTime();
    END
ELSE IF e.counter = k THEN BEGIN
    e.counter := 1;
    move e to the head of L;
    IF CurrentTime() - e.TimeStamp < t THEN
```

```
            RETURN true
        ELSE BEGIN
            e.TimeStamp := CurrentTime();
            RETURN false;
            END;
        END
    ELSE BEGIN
        e.counter := e.counter + 1;
        move e to the head of L;
        RETURN false;
        END;
```

The list $L$ in the algorithm is like a working set of the local processor. The length of the list, $M$, is then the size of the working set. The algorithm replaces the elements in the list based on a least-recently-used philosophy. $M$ should be small enough so that finding an element in the list is efficient.

When page thrashing is discovered during a page fault, it is possible to migrate the faulting process to the processor that owns the page. This method reduces the page thrashing if the faulting process and the processor owning the page use the page in the future. Such a balancing strategy can be viewed as data driven.

Although this method reduces the page thrashing, it may introduce new page thrashing when the migrated process needs to write to some pages being used by other processes on its original processor. In order to make an optimal decision of whether a process migration should be performed, the processor has to use the information about the future memory page references of all the processes on the faulting processor. Such information is difficult to obtain without a smart compiler. The suitable approaches for the shared virtual memory system are those that do not need the information.

The following are a number of rules based on the number of total processes and the number of ready processes on each processor:

- If the number of the processes on the processor owning the page is less than the one on the faulting processor, then the process should be mi-

grated.

- If the number of the ready processes on the processor owning the page is less than the number of ready processes on the faulting processor, then the process should be migrated.
- If the faulting processor does not have any more ready processes, the faulting process should not be migrated.

The rules above try to consider both maximizing parallelism and avoiding page thrashing without knowing anything about applications. It is possible to develop many other rules by combining the above rules with different weights.

The load balancing strategy can be combined with the active and passive load balancing strategies in favor of different aspects of the shared virtual memory system. The choices of how to combine them can be decided at the system initialization stage.

## 3.4 Conclusions

This chapter has studied the issues in process management for the shared virtual memory system on loosely coupled multiprocessors. Two kinds of process scheduling mechanisms have been studied—centralized and distributed. Both scheduling mechanisms support traditional process control primitives, synchronization primitives, and process migration capability.

The centralized process scheduling mechanism is simple but it has more overhead on loosely coupled multiprocessors. The centralized process scheduling mechanism with multiple ready queues can significantly reduce the communication overhead introduced by loosely coupled multiprocessors.

The distributed process scheduling mechanism studied in this chapter is attractive because it supports distributed scheduling strategies. With approximate information about the processes on each processor, many simple load balancing strategies can be applied to the process scheduling mechanism.

# Chapter 4

# Implementation

The available implementation environments all have limitations. In order to bring a shared virtual memory system into reality, implementors have to overcome these limitations. This chapter focuses on the engineering issues of such an implementation.

## 4.1 Implementation Environment

### 4.1.1 Basic Requirements

There are three basic things that the shared virtual memory system requires of a loosely coupled multiprocessor system:

1. *A reasonably fast communication link.*

   Many commercially available systems meet this requirement. In particular, popular local computer networks such as Ethernet [Metcalfc 76] or token-ring networks [Wilkes 79, Leach 83] satisfy the requirement. Although most commercial local network links support transmission at about 10M bit/second, the real transmission speed between two processors is only about 1M or 2M bit/second in most of the local computer networks using MC68000 based machines because processor speed dominates. For

implementing a shared virtual memory system, the most important aspect of the communication link is the ratio of machine speed to its real communication speed. My implementation experiments show that a processor speed of 1 MIPS with a real communication speed of 1M bit/second is sufficient for many parallel programs.

2. *Homogeneity of processors.*

   While the processors of the system do not need to run at the same speed, they should have the same data format and the same instruction set because processors share both data and instruction code. Whether it is possible to build a shared virtual memory system on a heterogeneous multiprocessor system is an open question.

3. *A memory management unit (MMU) with a page-level access protection mechanism.*

   The page-level protection mechanism allows programs to set the access mode (nil, read-only, or writable) for each page. An illegal memory reference will cause a page fault.

In addition to the above basic requirements, if one plans to implement a shared virtual memory on top of an existing operating system, the operating system should allow:

- sending a packet from one processor to another,
- setting the page-access protection of individual pages, and
- page-fault handlers provided by the programmers to implement a shared virtual memory mapping manager.

Many existing loosely coupled multiprocessors meet these basic requirements.

### 4.1.2 Ideal Environment

Although a shared virtual memory could be implemented on a multiprocessor system meeting the above requirements, several additional capabilities would provide an ideal environment.

In addition to the basic hardware requirements, an ideal system would have a large address space, say 32 to 40 bit wide, on each processor. This is a necessary condition for building a *large* shared virtual memory. An application program with many processes will use a large address space if each process has a relatively large stack.

The ideal system would also be able to distinguish read page faults from write page faults in hardware because the shared virtual memory mapping deal with them differently (see Chapter 2). The cost of distinguishing page faults in software cannot be ignored because it is paid on each page fault. Also, an instruction may require two memory references on the same page. If one of them is a write reference, then the processor should get write access to the page to avoid a read page fault. If the hardware can detect this, many unnecessary read page faults can be eliminated. Most existing hardware does not distinguish between read and write page faults because traditional operating systems do not need to do so.

A large, existing uniprocessor virtual memory system would ease the implementation of a shared virtual memory. If each processor has a large virtual memory, the shared virtual memory implementation might not need a special page replacement algorithm. Implementing a page replacement algorithm is almost as difficult as implementing a traditional virtual memory system.

Ideally, the multiprocessor will have a small page size, as small as 256 bytes. One reason is that memory contention is very closely related to page size; the smaller the page size, the less memory contention. Another reason is that a small page can be moved efficiently from one processor to another. If the page size is smaller than the network packet size, the memory mapping managers can move a page by placing it in one packet together with the information for a request or reply. It has been shown that the moderate page sizes (8–16K bytes) are better than small ones for diskless workstations [Lazoweska 84]. This does not match the small page size requirement for implementing a shared virtual memory system. In order to optimize both, architecture designers may need

to consider using different page sizes for disk paging and for shared virtual memory.

The ideal operating system would have an integrated heavyweight and lightweight process scheduler; then all the processes are scheduled by the same scheduler and they all use the same process synchronization mechanism. Without such a scheduler, an obvious alternative is a two-level process scheduler in which the operating system scheduler schedules all the heavyweight processes and a lightweight scheduler runs within a heavyweight process. The disadvantage of the two-level process scheduler is that it is difficult to overlap synchronous I/O among lightweight processes. When an I/O operation occurs in a lightweight process, the heavyweight process scheduler would suspend the heavyweight process where the lightweight process resides, and start another heavyweight process. The suspended heavyweight process could be invoked only when the I/O operation is complete.

Finally, since the low level implementation of a shared virtual memory mapping manager depends on the communication between processors, the ideal environment would have an efficient transmission protocol modeled on a remote procedure call (RPC) mechanism [Nelson 81, Birrell 83]. The protocol does not need to be as complicated as full RPC but it should be reliable. If the communication supports broadcast or multicast transmission, it would be convenient to have broadcast and multicast in the protocol. The shared virtual memory mapping managers relies on page moving, so an efficient page moving utility would significantly enhance the performance of the shared virtual memory implementation.

Although some of the above capabilities exist in some systems, no existing system contains all of them.

### 4.1.3 Loosely coupled Multiprocessors

This section describes the available types of loosely coupled multiprocessors that are suitable for implementing the shared virtual memory system. In the rest of the chapter, I will discuss in general the algorithms for implementing the system. When the algorithms are architecture-dependent, I will provide specific discussions of them.

The interconnection topology of available loosely coupled multiprocessors falls into two categories: complete connection and point-to-point connection.

The two popular complete connection architectures are *bus* and *ring*. A bus connection architecture has a common communication link connecting all the processors together; only one packet is transfered at a time on the bus. Hardware supports three kinds of communication services: point-to-point, broadcast, and multicast. A typical example of a bus connection multiprocessor is a number of processors connected by an Ethernet [Metcalfe 76].

A ring connection architecture usually uses a ring communication link on which processors pass tokens to each other. Most ring connection architectures only allow a single token to be passed around in a ring; well-known examples are the Cambridge ring [Wilkes 79] and the Apollo Domain [Leach 83]. Some ring connection architectures allow multiple tokens in a ring. Ring connection architectures can provide the same communication services that bus connection architectures do: point-to-point, broadcast, and multicast.

A point-to-point connection architecture has multiple links connecting processors; usually these architectures have a large number of processors. A processor can communicate directly only with its neighbors. If a processor sends a packet to a non-neighbor processor, the packet is stored and forwarded by the processors along the path from the source processor to the destination processor. The communications on different links can be done in parallel, but broadcast and multicast are usually not supported by hardware. Examples of point-to-point connection architectures are Cosmic Cube [Seitz 85] and CHiP

[Snyder 82].

Multiprocessor architectures of different scales are quite different in many aspects. A small-scaled loosely coupled multiprocessor system is usually a number of processors connected by a complete connection communication link. Each processor in the system has a relatively large physical memory and usually has its own secondary storage on which a virtual memory system is implemented.

A large-scaled multiprocessor system is usually designed only for parallel computation. Each processor in the system has a relatively small physical memory and has no direct data path to a secondary storage. Therefore, there is no virtual memory system implementation on the processors.

### 4.1.4 Communication Protocol

The implementation of a shared virtual memory system requires a reliable communication protocol for remote operations. This section discusses the requirements of the remote operations in a shared virtual memory system and informally describes a protocol to handle the remote operations for the system implementation.

There are three kinds of remote operations in a shared virtual memory system:

- implementing the memory mapping managers,
- process control and synchronization, and
- dynamic memory allocation.

A remote operation has the following general form:

*Request*( `processor, request_id, request_type, args` )

where **args** is a record that contains input and output arguments. I call the remote operation mechanism a *simple RPC* because it is a simplified version of the traditional RPC [Nelson 81, Birrell 83]. Syntactically, an RPC and simple RPC are similar. The simple RPC is different from the RPC in two ways. The

first is that the total number of remote procedures is fixed, eliminating the stub compiler and RPC id generator for dynamic RPC server binding. The second is that there is an upper bound on the execution time of a simple RPC; the upper bound is small, usually milliseconds. The simple RPC can simply wait for a reply (the RPC return) and retry the request if it does not get one within the bounded amount of time, so the acknowledgement packets for requests are not necessary.

The simple RPC in a shared virtual memory system has some new requirements. One is to have broadcast or multicast calls if a multiprocessor supports broadcast or multicast in hardware. A broadcast or a multicast request needs several reply schemes:

1. a reply from any receiving processor,
2. replies from all receiving processors, and
3. no reply at all.

The first option is useful for broadcasting page fault requests to locate page owners (see Chapter 2). The second option can be used for implementing invalidation operations. The third option is for broadcasting approximate information.

Another requirement of simple RPCs is a forwarding mechanism that allows a processor to send a request to another processor and get a reply from elsewhere. For example, processor 1 can send a request to processor 2, processor 2 forwards the request to processor 3, and so on until processor $m$ performs the operation and sends a reply back to processor 1. There are no intermediate replies involved in the operation. This mechanism is particularly useful for implementing the dynamic distributed manager algorithm (Chapter 2).

For further optimization, the system needs asynchronous simple RPC requests. A process that sends out an asynchronous request can proceed without waiting for a reply unless the number of outstanding asynchronous requests exceeds an upper bound, $k$. Once a reply to an outstanding asynchronous request is received, the calling process will be able to continue. This mechanism can

usually reduce the response time of a request by a factor of two which is useful for many modules in the system.

The simple RPC protocol should be able to deal with lost packets. Losing packets by hardware is a well-understood problem. Losing packets by software is also possible. A typical case is when a process runs out of physical memory buffers and the incoming packets are thrown away. Such a loss of a request or reply can create deadlocks or other serious errors in a shared virtual memory system, so the simple RPC protocol must have a retransmission mechanism. This simple RPC protocol consists of three parts:

- sending a request,
- receiving a request,
- and receiving a reply.

The following are the algorithms for these three parts:

**Algorithm 4.1** *SendRequest*

1. Send out the request according to the **request_type**, which can be either broadcast, multicast, one-to-one, or forwarding.
2. If the **request_type** indicates that the request is synchronous, suspend the process.
3. If the **request_type** indicates that the request is asynchronous and there are $k$ outstanding asynchronous requests, then suspend the process.
4. Resend an outstanding request if its reply has not arrived within time quota $t$ (where $t$ is greater than the maximum processing time of a request).

In order to resend outstanding requests efficiently, a timeout mechanism may be needed.

All the requests are received by a request dispatcher. The dispatcher dispatches an incoming request to a server process according to its **request_id**. The dispatcher protocol also needs to take care of the retransmission of requests and replies. It is possible that a resent request becomes a duplicate because

the original request may arrive at the same time as the retransmission decision is made. In order to distinguish duplicates from originals, each request needs a unique identifier.

**Algorithm 4.2** *ReceiveRequest*

1. If the request is a duplicated one, then drop the request.
2. If the reply for this request had been sent already, then resend the reply.
3. If the request is not a duplicated one, then dispatch the request to a server according to its `request_id`.

Resending a reply is necessary because the previous reply may have been lost.

All the replies are received by a reply receiver. The reply receiver completes a request and takes care of duplicated replies.

**Algorithm 4.3** *ReceiveReply*

1. If the reply is a duplicate, throw the reply away.
2. If the reply is for an asynchronous request, decrement the outstanding count and wake up the calling process if it was waiting for outstanding asynchronous requests.
3. If the reply is for a synchronous request, pack the returned arguments into the output arguments and wake up the calling process.

In practice, the reply receiver can be combined with the dispatcher into one program.

The retransmission protocol is based on the philosophy of resending replies only when necessary. Such a design is based on the following assumptions:

1. local computation is always correct, and
2. communication may be unreliable, but once a packet is received, its content is always correct.

The protocol is reliable only when these assumptions hold. In practice, the assumptions are reasonable.

## 4.2 Shared Virtual Memory Mapping

### 4.2.1 Implementation Modes

A popular way to protect system kernels is to use different "modes" for different programs. Most available architectures provide at least two modes: system and user. Programs in system mode can use any privileged instructions such as changing mode and prohibiting interrupts; programs in user mode cannot.

In an architecture with the two modes, control passes back and forth between them. Programs in different modes usually have different memories and stacks so that a program in user mode cannot destroy useful data in the system memory space, or adversely affect other user programs. For example, system programs such as interrupt handlers and virtual memory page fault handlers would be in system mode. When an interrupt occurs while the machine is executing a program in user mode, the mode is switched to system mode and the control is switched to its interrupt handler. When the processing is done, the control is returned to user mode.

A shared virtual memory can be implemented in either user mode or system mode. The advantages and disadvantages of either type of implementation are mainly reflected in system performance and implementation effort.

The system performance normally depends on the overhead of processing a page fault and the speed of sending and receiving a packet over the communication link. A user-mode implementation usually has more overhead in both instances than a system-mode implementation. Since programs in user mode cannot access any system memory location, the faulting mechanism requires many context switches. For example, when a fault occurs, the context is usually saved by hardware or software onto the system stack and copied onto the user stack in order to pass the control to a fault handler in user mode. When the control returns from the fault handler, the context on the user stack will be copied back onto the system stack and then restored into hardware. Therefore, a user-mode page fault requires at least four context switches; a system-mode

implementation requires only two.

The low-level operations for sending and receiving a packet are usually in system mode. If this is the case, the operation of transfering a packet from user memory space on one processor to another may require copying the packet back and forth between user memory space and system memory space in order to prevent system calls from getting page faults. In many existing systems, in addition to the cost of switching modes, a program in user mode may need to pay the cost of an extra copy for each send or receive.

While a system-mode implementation may be more efficient, it is usually more difficult to implement than a user-mode implementation. First of all, a system-mode implementation of the memory mapping managers may not allow further page faults once the control is in a page fault handler. So, the system data structures, such as page table entries and buffers for the simple RPC protocol, must be resident before using them. A user-mode implementation usually does not need to worry about the data structures because they can be simply stored in its user-mode virtual memory address space.

Program debugging is another important consideration. Debugging a parallel system is much harder than debugging a sequential system because of the difficulty of developing parallel program debugging tools. Debugging a system-mode implementation may require rebooting systems over and over again, which is particularly annoying when processors are geographically far away from each other. In debugging a user-mode implementation, one does not usually need to reboot the systems. Also, one can easily separate implementation bugs from operating system bugs because the system memory space is protected.

This is not to imply that a user-mode implementation is always easy. If the shared virtual memory system is implemented on top of an existing system, the implementation environment may be so poor that one has to modify some kernel code, in which case the implementer has to understand completely the operating system. A system-mode implementation integrated with an existing system, of course, requires understanding the operating system, but such

an implementation is probably not as chaotic as a user-mode implementation because usually the implementer will not need to modify someone else's code.

In general, it is probably a good idea to start with a user-mode implementation for experimental purposes and then build a system-mode implementation for efficiency.

### 4.2.2 Multiple Address Spaces

There are two basic ways to implement multiple address spaces in a shared virtual memory system: single heavyweight process implementation and multiple heavyweight process implementations.

In a single heavyweight process implementation, there is only one "big" shared virtual memory address space. When multiple applications request their address spaces, the big address space is partitioned into small pieces, one per application. A piece of address space allocated to an application is shared by a number of processors specified by the application. An address space in this case does not always have a starting address 0. Program relocation, however, can provide clients with a transparent interface. One advantage of this approach is its simple implementation. The disadvantage is that the number of shared virtual memory address spaces is limited by the size of the big address space.

In a multiple heavyweight process implementation, each processor has several heavyweight processes, one for implementing each shared virtual memory address space. When initializing a shared virtual memory space among a group of processors, one creates a heavyweight process on each processor in the group. These heavyweight processes implement the same shared virtual memory space and they do not communicate with the heavyweight processes for other shared virtual memory spaces.

If page fault handlers are implemented in user mode, each heavyweight process on each processor has a page table for the corresponding shared virtual memory space. The technique of implementing a single address space can di-

rectly apply to the created heavyweight processes. If page fault handlers are implemented in system mode, each processor only has one page table built at initialization stage. A processor tells which page fault belongs to which shared virtual memory space by looking up the page table. The advantage of this approach is that each address space can be rather large, starting from address 0. The disadvantage is that its implementation is more difficult.

The implementations of both multiple and single heavyweight process can present clients with the same interface, so choosing one over another depends on the individual system on which the shared virtual memory is being implemented. For example, one may choose single heavyweight process implementation on an architecture with a large address space for simplicity.

### 4.2.3 Page Table Compaction

Recall that the data structure used by both page fault handlers and their servers is a page table. The size of a page table is proportional to the size of its shared virtual memory. A page table needs to have $m/s$ entries, where $m$ is the size of a shared virtual memory space in bytes and $s$ is the page size in bytes. For example, if the size of a shared virtual memory space is as large as $2^{32}$ bytes, and $s$ is $2^{10}$ bytes, the page table will have $2^{22}$ (or more than 4 million) entries. This is too large for any implementation since the memory coherence algorithms described in Chapter 2 require each processor to have its own page table.

A page table can be compacted by reducing the number of entries and compressing the size of each entry. The following presents a straightforward way to reduce the number of page table entries and three basic methods to compact a page table entry.

#### Hash Page Table

There is no way to reduce the number of page table entries if every processor uses the whole shared virtual memory address space. Fortunately, this is not

the normal case. A parallel program based on a global memory model normally has a number of processes, at least one on each processor. Each process usually accesses a portion of the shared data structures. This means that when a shared virtual memory system has a number of processors, each processor only accesses part of the address space. If this is true, the hash table technique can be used to compact a page table [Knuth 73].

The basic hash table size can be adjusted according to the configuration of the system. For example, it can be the smallest prime number $q$ such that

$$q > \frac{m}{csN}$$

where $c$ is a parameter to be adjusted and $N$ is the number of processors in the system. Each entry in the hash table is a linked page table entry list.

In order to keep the size of the hash page table small, one can delete a page table entry when the access to the page becomes nil and no process is accessing it. This can be done either when the memory coherence algorithm invalidates a page or when the total size of the page table exceeds some upper bound.

### Page Table Entry

The size of a page table also depends on the size of each entry. In the dynamic distributed manager algorithm, each entry in the page table consists of five fields: probOwner, access, lock, queue, and copy set (see Chapter 2). If the shared virtual memory has a 32 bit address space and there are $N$ processors in the system, then:

- the probOwner field needs $\lceil \log N \rceil$ bits,
- the access field needs two bits because it has three states: write, read, and nil,
- the lock field needs one bit if there is a test-and-set instruction available,
- the queue field needs 32 bits, and
- the copy set field needs $N$ bits if bit vector is used to represent a set [Aho 74].

Therefore, the size of an entry (in bits) is

$$s_e = N + \lceil \log N \rceil + 35.$$

If $N$ is small (e.g. less than 32), the page table entry size is not bad. When the shared virtual memory system is on a large-scaled multiprocessor, this is a serious problem because the size of an entry is linearly proportional to the number of processors in the system.

The probOwner field does not need any compaction because $\lceil \log N \rceil$ bits is small enough. The access field and the lock field do not need any compaction either because they only need three bits together.

The queue field can be compacted by using indices of queue nodes instead of using real pointers. Queue nodes in this case are allocated from a node pool area which is represented as a vector. When a queue is dismissed, the queue nodes will be deallocated. An index of the vector only requires $\lceil \log s_v \rceil$ bits where $s_v$ is the length of the vector. Since each processor has its own node pool, $s_v$ is the maximum number of processes allowed to be waiting for page faults on each processor at any given time. Thus, $s_v$ is bounded by the maximum number of processes on each processor, which is usually small. For example, when $s_v = 1024$, the queue field needs only 10 bits.

The copy set field is a set of processor numbers $S = \{1, \ldots, N\}$. It is used solely by the memory coherence algorithm with the following three operations:

- $Insert(e, S)$ — inserts an element $e$ into a set $S$.
- $Next(e, S)$ — returns nil if all the elements in $S$ are less than $e$; otherwise returns the smallest element $e'$ in $S$ such that $e' > e$.
- $Size(S)$ — returns the number of elements in set $S$.

The speed of these operations is not very important, so the algorithm only concentrates on saving space. Three ways to compact a copy set are: linked bit-vector, neighbor bit-vector, and vaguely-defined set.

### Linked Bit-vector

The linked bit-vector approach represents a copy set as a linked list. The main idea of this approach is to link the meaningful bit-vectors together to save space. Each element in the list has three fields: *link*, *index* and *bit-vector*. The link field points to the next element in the list. The bit-vector field is a zero-based bit vector of $l$ bits. On each processor, there is a zero-based vector called *base* defined as:

$$base[i] = li \quad \text{for } i = 0, \ldots, \frac{N}{l} - 1.$$

The index field is used to index the *base* to tell the starting point of its bit-vector. The definition of the bits in a bit-vector is that $base[index] + i$ is in the copy set if and only if the $i$-th bit of a bit-vector is 1. Figure 4.1 shows a compacted copy set $S = \{3, 995\}$ when $N = 1024$ and $l = 32$.

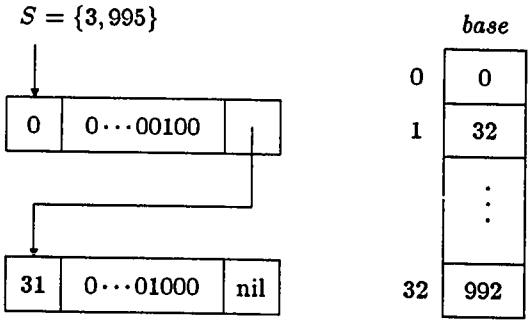


Figure 4.1: An example of linked bit-vector compaction.

Since *Next* and *Size* are trivial, I present only the algorithm for *Insert*:

**Algorithm 4.4** *Insert*$(e, S)$

1. Traverse down to the $k$-th element in the list such that $e > base[index_k]$ and $e \leq base[index_k] + l$.[1]

[1]For simplicity, $index_i$ means the *index* field of the $i$-th element in the list.

2. If such an element does not exist or $i > l$ (which means that $i$ is out of the range of the bit-vector), then make a new element and insert it as the new $k+1$-th element in the list. Set $index_{k+1}$ to an appropriate value such that $e > base[index_{k+1}]$ and $e \leq base[index_{k+1}] + l$.
3. Set the $i$-th bit of the bit-vector to 1 such that $e = base[index_k] + i$.

The *index* field requires $\lceil \log N/l \rceil$ bits. The *bit-vector* size $l$ can be adjusted according to need. The link field in each element can usually be compacted with the compaction method proposed in [Li 86].

The linked bit-vector compaction method works well when the copy set is very sparse, but it does not work well otherwise.

### Neighbor Bit-vector

In many large-scaled multiprocessors, a processor only has a direct connection to a small number of processors, which are usually called neighbors. For example, in a Hypercube of $N$ processors, each processor is directly connected to $\log N$ processors. When a processor sends a packet to a non-neighbor processor, the packet is stored and forwarded by the processors along the path. For multiprocessors of this kind, each processor can only store its neighbor processor's numbers into its copy set. Such a copy set is called *neighbor copy set*.

The neighbor bit-vector approach uses a neighbor copy set $S'$ to compact the information of a copy set $S$ on processor $i$. The neighbor copy set is the set of all neighbors of processor $i$ that are on the shortest paths from $i$ to the elements of the copy set. A more formal definition is that a neighbor $e \in S$ if and only if there exists $k \in S$ such that $e \in P_{k,i}$ where $P_{k,i}$ is the set of the processors (including $k$) that are on the shortest path from processor $k$ to processor $i$.

The following is always true:

$$S' \subseteq S \subset \{1, \ldots, N\}$$

because $e \in S$ represents all the processors for which processor $i$ asks processor $e$ to forward packets. The three operations are not listed because they are straightforward.

$S'$ is represented by a bit-vector of $l$ bits where

$$l = \max_{k=1}^{N} |S_k|.$$

The meaning of the bits in the bit-vector on a processor is defined by an internal table $T_1$ such that $T_1[k] \in S'$ if and only if the $k$-th bit in the bit-vector is 1. There is another internal table called $T_2$ that maps processor numbers to bit numbers in the bit-vector. Table $T_1$ is initialized on each processor according to the neighbor processor numbers. Table $T_2$ is initialized such that

$$T_1[T_2[p]] = p.$$

For example, suppose a shared virtual memory is implemented on an 128-node hypercube multiprocessor. Each processor has seven neighbors which are labelled $0, \ldots, 7$; so each copy set needs seven bits. The hardware routing guarantees that any packet in the multiprocessor traverses at most 7 processors because $7 = \log 128$. Table $T_1$ on each processor implements the mapping between processor numbers and its neighbor labels. Table $T_2$ implements a inverse mapping.

The neighbor copy set definition requires modifications to the memory coherence algorithm such that an invalidation operation always sends a broadcast-like message to a neighbor if it is in the neighbor copy set. When an invalidation is sent to a processor $e \in S'$, it needs to send to all the processors on the shortest path starting from $e$ even if only $e$ has a copy. A partial solution to this problem is to use a status bit-vector $V$ of $l$ bits for each neighbor copy set. The $k$-th bit of $V$ is 1 if and only if $T_1[k]$ represents other processors. If distributing copy sets (described in Chapter 2) is used in the memory coherence algorithm, the probability of setting the bits in $V$ to 1's will be smaller. This claim, however, needs empirical data to justify.

The positive side of this approach is that it only uses $l$ bits for each copy set. In a hypercube multiprocessor, $l = \log N$.

### Vaguely-defined Set

A *vaguely-defined set* is, as the name indicates, a set without precise definition. The idea of a vaguely-defined set was developed because the copy set field is only used by the invalidation operations in the memory coherence program. It is harmless to send a page invalidation request to a processor that does not have the page. This is why the memory coherence algorithms allow a processor to broadcast an invalidation request when the number of elements in a copy set exceeds an upper bound.

The vaguely-defined set approach uses a one-bit tag together with any list representation of a copy set. If the tag bit is 0, the list is valid. If the tag bit is 1, the list is invalid and the copy set contains all the processor numbers. Figure 4.2 shows an example of a vaguely-defined set.

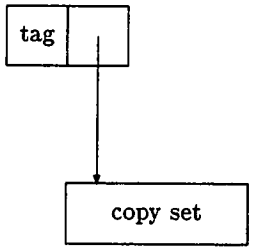


Figure 4.2: An example of vaguely-defined set compaction.

The three operations on the copy set are simple:

**Algorithm 4.5** *Insert*$(e, S)$

1. If the tag bit is 0, and the length of the list of $S$ is less than $l$, put $e$ into the list.
2. If the tag bit is 0, and the length of the list of $S$ is equal to $l$, set the tag bit to 1.

**Algorithm 4.6** $Next(e, S)$

1. If the tag bit is 0, then traverse down the list and return the smallest element $e' \in S$ such that $e' > e$.
2. If the tag bit is 1, then return $e + 1$ if $e < N$.

**Algorithm 4.7** $Size(S)$

1. If the tag bit is 0, then return the length of the list.
2. If the tag bit is 1, then return $N$.

This approach does not restrict the data structure of the list. The simplest data structure might be a list of processor numbers. The processor number field in this case needs $\lceil \log N \rceil$ bits and the list can be compacted by the method proposed in [Li 86] so that normally only two bits are needed for each pointer field. The ideas of the linked bit-vector can be also applied to the vaguely-defined set approach.

All three approaches have advantages and disadvantages. The linked bit-vector approach is suitable for the shared virtual memory system in which copy sets are sparse, but it is worse than using bit-vector when they are not sparse. The neighbor bit-vector approach works well for the multiprocessor system in which each processor has a small number of neighbors; it does not do much good when the number is large. The vaguely-defined set approach is simple and should be good for the case in which memory pages are either shared by all the processors, or very few processors. It can also be combined with other compaction approaches.

By using the hashing technique and compacting page entries, one can have a rather practical page table representation for a large shared virtual memory address space. But the total storage needed for a page table may still be too large to fit into physical memory all the time. In this case, one needs to get help from secondary storage or other processors.

### 4.2.4 Page Replacement

Since the size of a physical memory on a processor is usually much less than the size of a shared virtual memory address space, the implementation of a shared virtual memory system must have page replacements between a physical memory and other storage devices.

It is well understood that there is no ideal solution for page replacement because an ideal algorithm requires data about future memory references which are impossible to predict. Therefore, this section investigates what should be done to make a better page replacement algorithm. The section studies how to find a page for replacement and investigates how to replace it. Three cases are considered: page replacement for a multiprocessor in which each processor has a private or shared disk, integration of shared virtual memory page replacement with an existing virtual memory system, and page replacement among physical memories.

#### Page Replacement Priority

When a page fault occurs and there is no unused physical memory page available, the shared virtual memory system invokes the page replacement algorithm to find a memory page and save it elsewhere to make a space for the new page to put in. The question is which page should be chosen for replacement. Obviously, the page replacement algorithm of a traditional virtual memory system cannot be applied here because a shared virtual memory system has more kinds of memory pages than a traditional one. Different pages have different replacement priorities even if they have the same last reference time.

The replacement priority of a page can be computed by its page-type priority and its last reference time. The relationship between the two factors can be adjusted by the system designers. A possible way is to weight last reference time by:

$$prio_{replace} = prio_{type} w + (t - t_{last}) \tag{4.1}$$

where $prio_{replace}$ is the replacement priority of a page, $prio_{type}$ is the page-type priority assigned by the system designers, $w$ is a weight which is usually the same for all page types but can be different for different page types, $t$ is the current time, and $t_{last}$ is the last reference time of the page. The Least-Recently-Used (LRU) page replacement policy can be viewed as a special case of the page replacement priority, one which does not have the term of $prio_{type}w$. Page replacement priority ensures that LRU is used for each page type while the priorities are preserved among different page types for a range of last reference time. Adjusting the value of $w$ for each page type can control such a range.

There are five kinds of memory pages in a shared virtual memory address space: *writable, owned read-only, read-only, nil access*, and *unused*. A writable page is obviously owned by the processor. An owned read-only page is also owned by the processor but it is read-only. A read-only page is not owned by the processor, but the processor knows who owns the page. A nil access page is a memory page invalidated by the memory coherence algorithm. An unused page is a free memory page. Table 4.1 shows a possible page type priority assignment for different pages in a shared virtual memory space.

| Page type | Replacement priority |
|---|---|
| writable | 1 |
| owned read-only | 2 |
| read-only | 3 |
| nil access | $\infty$ |
| unused | $\infty$ |

Table 4.1: A page-type priority assignment

Nil access (or invalidated) memory pages are treated the same as unused memory pages—they all have the highest priority for replacement. This is because if any program wants to use a nil access page, a page fault will be generated anyway. Writable memory pages and owned read-only memory pages

should normally have lower replacement priorities than read-only pages because some processors may want to make read-only pages from them. Writable pages have lower replacement priority than owned read-only pages because each writable page only has one copy in the whole shared virtual memory system. The page-type priority assignment can be affected by the architecture of the target multiprocessor and the memory coherence algorithm used. Optimal decisions require empirical data on the particular implementation environment.

For a single address space implementation, the shared virtual memory system maintains a number of page sets, each for the pages with the same page-type priority. Each set is ordered by its last reference time such that the system can efficiently find the least recently used page in it. For convenience, a set is denoted by $S_{prio}$ where *prio* is the page-type priority. For example, the unused pages and nil access pages are in $S_\infty$. Assuming that there are $k$ page-type priorities in addition to $S_\infty$, the following algorithm can be used to find a page for replacement for a single address space implementation:

**Algorithm 4.8** *FindPage*

1. If $S_\infty$ is not empty, delete a page $p$ from $S_\infty$, and return $p$.
2. Find the highest replacement priority page $p$ by using equation

$$prio_{replace} = prio_{type} w + (t - t_{last})$$

from the page-type set $S_i$, $i = 1, \ldots, k$, and return $p$.

In this algorithm the priority computation is performed $k$ times, one for each set. The algorithm is guaranteed to find the page with the highest page replacement priority.

For a multiple address space implementation, finding a page for replacement is more complicated. Recall that the page replacement problem in the traditional virtual memory system is solved by the concept of working set. Peter Denning has clearly defined the working set of the traditional systems [Denning 80]:

*"The working set is usually defined as a collection of recently referenced segments (or pages) of a program's virtual address space."*

The concept of "recently referenced" is not applicable to the shared virtual memory because nil access pages are not in the working set although they might be referenced recently. The working set in a shared virtual memory system is defined as a collection of low replacement priority pages of an address space.

Furthermore, instead of using the virtual time of each heavyweight process to bound the size of a working set, the working set size for each address space is fixed [Levy 82]. This strategy ensures that each address space has at least some number of pages in memory and they can never be replaced by the page faults in other address spaces. The memory pages that are not in any working set can be replaced by the page faults in any address spaces. Therefore, the shared virtual memory system needs a global data structure to keep track of these pages and a data structure for each working set.

The global data structure is a collection of page sets, one for each page-type priority except that there should be only one $S_\infty$ in the whole system. The working set data structure is the same as the single address space implementation. Since the size of each working set is fixed, when a new memory page is added to a working set, the page with the highest replacement priority will be deleted from the working set and put into the global data structure.

The following is an algorithm for finding a page in a multiple address space implementation:

**Algorithm 4.9** *FindPage*

1. If $S_\infty$ is not empty, delete a page $p$ from $S_\infty$, and return $p$.
2. Find the highest replacement priority page $p$ by using equation

$$prio_{replace} = prio_{type} w + (t - t_{last})$$

from $S_i$, in the global data structure, $i = 1, \ldots, k$; if $p$ is found, return p.

3. Find the highest replacement priority page $p$ by using equation

$$prio_{replace} = prio_{type}w + (t - t_{last})$$

from $S_i$, in the working set of the faulting process, $i = 1, \ldots, k$, and return p.

This algorithm is more general than that for the single address space implementation. The one for single address space implementation simply does not have working sets.

Unfortunately, it is impossible to compute replacement priorities using Equation 4.1 unless there is special hardware support. The only popular hardware support in most available machines is to use a usage bit per page frame and keep the bits aside in a special area of the primary memory space. To live with the existing hardware, one needs to approximate the computation of page replacement priorities.

A simple way to approximate page replacement priorities is to use the idea of widely-used CLOCK replacement algorithm [Belady 66, Easton 76, Carr 81]. The original CLOCK algorithm is to use a roving pointer scanning through the page frames of main memory, skipping used frames and resetting their usage bits. The first page frame with its usage bit off is chosen for replacement. The roving pointer acts as the hand of a clock.

For the page replacement priority scheme, one can apply the idea of the CLOCK algorithm to each page set instead of to the whole main memory. Each page set is represented by a circular list with a roving pointer. When the procedure *FindPage* tries to find the highest replacement priority page from a page set $S_i$, it scans through the pages in the set starting from the roving pointer. The first page with its usage bit off is chosen as the highest replacement priority page in $S_i$. If all bits are set, take the page that the roving pointer points to. After executing *FindPage*, the system resets all the usage bits. When a page is added into a page set, it is always inserted before the roving pointer so that it has the lowest replacement priority in the next search.

To preserve the page-type priority, each page set has a counter indicating consecutive deletions from the set. Each page set also has an upper bound on the number of consecutive deletions allowed. A counter will be set to 0 if it reaches its upper bound. *FindPage* can use the following equation to crudely approximate Equation 4.1:

$$prio_{replace} = prio_{type} - counter \tag{4.2}$$

for the highest replacement priority page in each page set.

There are clearly other possible approximation methods. For example, the working set in the shared virtual memory system can be redefined by using the parameter of time instead of a fixed size. These design choices are left for the system designers.

### Replacement on Local Disks

This section considers page replacement in a shared virtual memory system implemented on a multiprocessor in which each processor has its own disk or some other kind of secondary storage. A typical example is a network of workstations. For convenience, we assume that each processor has an infinitely large secondary storage space.

The shared virtual memory system on this architecture has two kinds of memory page faults in a physical memory: disk page fault and shared virtual memory page fault. A disk page fault causes a page to move from disk to memory. A shared virtual memory page fault can be either a read page fault or a write page fault which may cause the shared virtual memory mapping managers to bring a page into memory from another processor.

When a memory page is chosen for replacement, the page is either thrown away or saved on disk. In a traditional virtual memory system, a page is written back to disk if it is dirty; otherwise, it will be thrown away. In a shared virtual memory system, a page will be thrown away if it is not dirty; but a page might

be thrown away even if it is dirty. Table 4.2 shows a replacement strategy for a dirty page.

| Page type | Replacement strategy |
|---|---|
| writable | *backup* |
| owned read-only | *backup or throw away* |
| read-only | *throw away* |
| nil access | *throw away* |
| unused | *throw away* |

Table 4.2: A replacement strategy for a dirty page

Whether a read-only page should be thrown away depends on the cost of transferring a page via the communication link and the cost of moving a page between memory and the disk because a read-only page can be brought back from its owner via the communication link. The cost of a read-only page replacement by using the disk is:

$$C_{replace} = C_{write} + \rho C_{read} \tag{4.3}$$

where $C_{replace}$ is the average cost of a page replacement, $C_{write}$ is the cost of writing a memory page to disk, $C_{read}$ is the cost of reading a memory page from disk, and $\rho$ is the probability of the next fault on the page. According to the principles of localities of programs, $\rho$ is close to 1. The cost of a read-only page replacement by using the network (throwing away) is:

$$C_{replace} = \rho C_{network} \tag{4.4}$$

where $C_{network}$ is the cost of moving a page from one processor to another. In a network of workstations, a number of diskless processors usually share one disk server. If this is the case, $C_{read}$ is about the same as $C_{network}$ if the system uses the dynamic distributed manager algorithm because the current processor knows the true owner of the page. It is clear that a read-only page should

be thrown away in this case. Even if the disk is not shared, a read-only page should be thrown away as long as $C_{network}$ is less than $2C_{write}$.

The replacement of an owned read-only page is a little more complicated. In order to throw the page away, its ownership must migrate. Such an operation costs a remote operation to change the remote page table. The operation may also increase the time for locating the owner of the page when another processor has a fault on the page. If the increased cost in the future for locating the owner is ignored, the cost of an owned read-only page replacement by using the network (throwing away) is:

$$C_{replace} = C_{rpc} + \rho C_{network} \tag{4.5}$$

where $C_{rpc}$ is the cost of a simple RPC. Although in many existing high performance system implementations [Birrell 83, Cheriton 84], $C_{rpc}$ is less than $C_{write}$, while $C_{network}$ is about the same as $C_{read}$, it is not a good idea to throw an owned read-only page away because there is no guarantee that there is a read-only page on at least one other processor after read-only pages are thrown away.

To throw a writable page away, ownership of the page must migrate. Since the page is writable, there is no read-only page anywhere in the system; so an ownership migration includes moving a network page in addition to a simple RPC. The cost can be expressed by:

$$C_{replace} = C_{rpc} + (1 + \rho)C_{network}. \tag{4.6}$$

This is greater than Equation 4.5. Furthermore, other processors have a higher probability of spending more time locating the owner of the page if they have faults on the page because the new owner is the only processor from which they can make read-only pages. It is probably a good idea to save a writable page onto disk for replacement, although this is still an architecture-dependent problem.

### Replacement on Existing Systems

When a shared virtual memory system is implemented on top of a traditional operating system, one may need to consider how to integrate a shared virtual memory page replacement with the existing virtual memory system page replacement. In this kind of implementation environment, each processor has a virtual memory implementation. For simplicity, we assume that the virtual memory address space on each processor is large enough.

The simplest way to integrate a shared virtual memory page replacement with an existing virtual memory system is to put both the shared virtual memory address space and its page table into a traditional virtual memory space without doing any page replacement work. Both the page table and the shared virtual memory space will be paged in and out according to the page replacement algorithm of the existing virtual memory system. In most existing systems, the algorithm is a form of approximate LRU.

While this approach is simple, it has a number of disadvantages. First of all, the page table and the shared virtual memory space are treated in the same way; it is possible that the page table uses too many or too few physical memory pages. Secondly, page replacement based only on last-reference time is probably not the best method, as discussed in Section 4.2.4. An obvious example is that nil access pages may not be chosen for replacement because they are referenced recently. Thirdly, the replacement strategy is not fair; for example, a nil access page replacement may cause the system to save the page onto disk. Another problem is that the page table may need a large address space.

To improve the shared virtual memory implementation on top of an existing virtual memory system, one may consider altering the existing virtual memory system. To solve the problem of using too many pages or too few pages for the page table, one can use a call to enforce sweeping out a memory page. If it is possible to make a call to bring in a given page, other tricks can improve

performance.

The fairness problem of page replacement can be partially solved by changing the attributes of memory page frames. When a memory page is invalidated by the memory coherence algorithm, the page becomes nil access and it will be marked "not dirty" and "never referenced". It also should be removed from the working set of the current address space. If all this can be done, the existing virtual memory page replacement algorithm will be able to treat it as a page with higher replacement priority. For read-only, owned read-only and writable pages, there is no obvious way to solve the problem other than changing the existing virtual memory page replacement algorithm, which is difficult.

A solution to the page replacement for page table is to put the page table into the system virtual memory address space. When the system builders decide to have a system-mode implementation, this comes naturally.

The above modifications may enhance the performance of the shared virtual memory performance substantially when an application requires a large address space. But to make the whole system work, one would need intricate knowledge of the existing system.

### Replacement Among Memories

When a shared virtual memory system is implemented on a multiprocessor in which each processor has a small memory (either physical or virtual), the page replacement algorithm will differ from that which has been discussed. An example architecture in this category is a hypercube multiprocessor; each processor has a relatively small physical memory and there is no direct connection to secondary storage. In this architecture, when a page is chosen for replacement and needs to be saved, the only way to replace the page is to save it into the memory of another processor. Memory pages do not need to be marked "dirty" because every page is dirty. The pages in $S_\infty$ will be thrown away. A read-only page will also be thrown away because it can be found from its owner. An owned read-only page requires an ownership migration. A writable page needs

to have both page saving and ownership migration.

The following is an algorithm for replacing an owned read-only page:

**Algorithm 4.10** *ReadOwnerMigrate*($p$, Hint)

1. $i$ = *FindProcessor*(Hint).
2. Send a request for ownership migration of page $p$ to processor $i$.
3. If the request is accepted and there is a request for the page, send the page to processor $i$; set the prob_owner of page $p$ to $i$ and return.
4. If the request is accepted, then set the prob_owner of page $p$ to $i$ and return.
5. If the request is rejected, call *ReadOwnerMigrate*($p$, Hint).

**Algorithm 4.11** *Server*

1. If there is a read-only copy of $p$, accept the request, make the current processor to be the owner of $p$ and return.
2. $p'$ = *FindDisposablePage*().
3. If $p'$ = nil, send a reject reply and return.
4. Accept the request and send a request for the page, change received page $p$ to $p'$, change the virtual memory mapping, make the current processor to be the owner of $p$, set the page to be read-only and return.

If the destination processor has a read-only copy of $p$, this algorithm can probably avoid moving a page. But it does not always avoid page moving, because read-only pages can be thrown away.

The algorithm for replacing a writable page differs slightly. When the server accepts a request for ownership migration and the page itself, it will set the page to be writable instead of read-only. When a writable page is replaced, everything needs to be moved to another processor; there is no alternative.

Procedure *FindDisposablePage* tries to find a disposable page. A disposable page is either a page in $S_\infty$ or a read-only page. The procedure first tries to find a page in $S_\infty$. If $S_\infty$ is empty, the procedure tries to find a read-only

page with the highest replacement priority from $S_r$. If such a page does not exist, the procedure returns nil.

Procedure *FindProcessor* returns a possible destination processor. Its main goal is to find a processor in the least expensive manner. Ideally, the procedure finds a processor with

1. a read-only copy of the replaced page if the request is to replace an owned read-only page,
2. an unused or nil access page (in $S_\infty$), or
3. a read-only page that has the highest replacement priority in the union of $S_r$'s on all the processors.

To achieve this, the procedure needs the up-to-date information about $|S_\infty|$ and $S_r$ of each processor. Unfortunately, this is not available.

A reasonable heuristic method is to maintain a global table `NodeInfo` on each processor. `NodeInfo` is a vector of $N$ records. Each record has two fields: *nils* and *reads* where nils is the latest information about $|S_\infty|$ and reads is the latest information about $|S_r|$. `NodeInfo` is initialized at the initialization stage and maintained by procedure *FindProcessor* and the replacement algorithm. The argument `Hint` provides the procedure with information about the copy set of page $p$ and a roving pointer indicating the last processor being tried. The following is an algorithm for *FindProcessor*:

**Algorithm 4.12** *FindProcessor*(`Hint`)

1. $i = Next(0, \texttt{Hint.copyset})$; delete $i$ from `Hint.copyset`; if $i \neq$ nil, return $i$.
2. Find $i$ such that $i$ has the maximum value of `NodeInfo[` $j$ `].nils` where $j = 1, \ldots, N$; if `NodeInfo[` $i$ `].nils` $\neq 0$, return $i$.
3. Find $i$ such that $i$ has the maximum value of `NodeInfo[` $j$ `].reads` where $j = 1, \ldots, N$; if `NodeInfo[` $i$ `].reads` $\neq 0$, return $i$.
4. Advance `Hint.rover` by one (modulo $N$) and return `Hint.rover`.

This procedure tries to find a processor with a read-only copy first. If it fails, it tries to find a processor with a disposable page. If it still fails, it returns a processor number in a round-robin style. When processor $i$ sends a request to processor $j$, `NodeInfo[ i ]` will be packed into the request. Similarly, processor $j$ packs `NodeInfo[ j ]` into the reply. Upon receiving a request or reply, a processor will unpack the information and update the corresponding record in its local `NodeInfo`. This way it does not introduce any additional packets in the replacement strategy and prevents *FindProcessor* from making incorrect guesses. Note that all the procedures in the algorithm are atomic operations implemented by some locking mechanism.

The main concern with Algorithm 4.2.4 is whether the page replacement algorithm terminates and under what condition it does. The following theorem answers the question:

**Theorem 4.1** *The page replacement algorithm will terminate if the size of the shared virtual memory address space, $m$, is bounded by:*

$$m \leq \sum_{i=1}^{N} m_i - \max_{i=1}^{N} m_i - N$$

*where $m_i$ is the size of the memory space on processor $i$.*

*Proof:* By contradiction. Assume processor $i$ has a page fault that causes the page replacement algorithm not to terminate. When this happens, the requesting processor is rejected by all other processors. Let us count how many pages there are in the shared virtual memory system to make this happen. A processor needs to move a page to another processor only when the page for replacement is owned by the processor itself, so there is at least one page charged to the shared virtual memory space from this processor. Since all the procedures in the algorithm are atomic, there is at most one read-only page being made on a processor. Therefore, processor $j$ rejects a replacement request when

1. its memory has $m_j - 1$ pages owned by itself and a read-only being made for a process on processor $j$, or

2. its memory has $m_j$ pages owned by processor $j$ and there is no read-only page being made.

Therefore, at least $m_j - 1$ pages are charged to the shared virtual memory space from processor $j$. Suppose the requesting processor number is $x$, there are at least

$$\sum_{i=1}^{N} m_i - m_x - N$$

pages charged to the shared virtual memory space. Thus, the smallest shared virtual memory space that can possibly make the algorithm terminate is

$$m > \sum_{i=1}^{N} m_i - \max_{i=1}^{N} m_i - N$$

which is false. □

In order to enlarge the shared virtual memory space in a system, a processor can certainly try to find a disposable page in its memory when it failed $N - 1$ times. The disadvantage of the algorithm is the unfairness of page replacement on destination processors. For one, it uses *FindDisposablePage* instead of using *FindPage*. Using *FindPage* may introduce two problems. One is that the maximum safe size of the shared virtual memory is reduced to

$$m \leq \max_{i=1}^{N} m_i.$$

The other problem is that when a destination processor is allowed to replace a page owned by itself, a nested ownership migration may be generated. In order to avoid a deadlock, the replacement algorithm has to remember which processors are on the requesting path so that the last requesting processor will not try to generate a loop.

This page replacement algorithm can also apply to the multiprocessors in which each processor has a large address space but does not have a large virtual memory space. Thus, it is possible to implement a shared virtual memory almost as large as the sum of all the virtual memory spaces in the system.

### 4.2.5 Integration of Memory Coherence Algorithms

As mentioned in Chapter 2, there are many memory coherence algorithms. I have shown that the dynamic distributed manager algorithm is good when the contending processor set is small. The fixed distributed manager algorithm, on the other hand, is good when the contending processor set is large. This section investigates the applications of different memory coherence algorithms to different kinds of pages.

The first issue in integrating memory coherence algorithms is to decide the granularity of the shared virtual memory space for the different algorithms. The smallest memory unit that uses the same memory coherence algorithm can be a page, a segment, or any predefined number of pages. Such a unit is called a *section*. A simple implementation of the idea uses a table that contains the page type information for each section. The page type for each section can be initialized at the initialization stage. When a user program allocates a piece of memory, it can change the page types of the section that enclosed the allocated piece of memory.

For every page fault, its page fault handler dispatches control to the appropriate memory coherence algorithm according to its page type. The page fault servers can be modified in such a way that they combine the functionality of both the dynamic distributed manager algorithm and the fixed distributed manager algorithm.

This approach can exploit the advantages of different memory coherence algorithms. The disadvantage is that it uses extra space for the page type table and pays the cost of dispatching.

## 4.3 Process Management

### 4.3.1 Process control primitives

The speed of process control primitives is important for a shared virtual memory implementation. For example, if a parallel program creates $k$ processes on one processor, runs them on $N$ processors, and joins them together on one processor, the speed of the program can be roughly expressed by:

$$C = \sum_{i=1}^{k}(C_{create}(i) + C_{migrate}(i) + C_{notify}(i)) + \frac{\sum_{i=1}^{k} C_{execute}(i)}{\min(k, N)} + C_{overhead}$$

where $C_{create}(i)$ is the creation time of process $i$, $C_{migrate}(i)$ is the migraration time of process $i$, $C_{notify}(i)$ is the notification time of process $i$, $C_{execute}(i)$ is the execution time of process $i$, and $C_{overhead}$ is the overhead of parallel computation which may be dependent on $N$. If the granularity of the parallelism in the program is fine, it is possible that

$$C_{create}(i) + C_{migrate}(i) + C_{notify}(i) \geq C_{execute}(i), \quad \text{for } i = 1, \ldots, k.$$

In this case, the speed of the parallel execution of the program is slower than the sequential execution of the program. So, $C_{create}(i)$, $C_{migrate}(i)$ and $C_{notify}(i)$ can set a lower bound on the granularity of the parallelism in the program.

Although in the shared virtual memory system, $C_{migrate}(i)$ and $C_{notify}(i)$ may dominate $C_{create}(i)$ in the programs in which processes need to migrate, fast process creation is still desirable for the programs in which processes do not need to migrate. The trick used in many systems for light weight process creation is to maintain a free list of preinitialized Process Control Blocks (PCB) [Birrell 84]. A process creation operation binds necessary values to a PCB and puts the PCB into ready queue. Binding usually includes such values as PC value, arguments, and the data pointer register value. Since the stack of the process and other information are preinitialized, the cost of a process creation can be no slower than a few procedure calls.

Process migration is a simple RPC that moves a process from one processor to another. In order to make the operation fast, one may consider making process migration an asynchronous remote operation, that is, sending a request to another processor and returning without waiting for its reply. A process migration sometimes can be done by creating a process remotely if the programmer (or the compiler) knows that the process will be started on another processor. Usually creating a process requires setting one or two registers such as stack pointer (SP) register and data block (DB) register, but a process migration needs to transfer all registers. A process creation may never need to allocate a stack if there are preinitialized PCBs available, but a process migration needs to move some pages in its stack because the memory pages of the stack have been allocated on the source processor. So, creating a process remotely can avoid the overhead of moving many register values and some portion of the process stack.

When a process terminates, it usually checks to see if there are awaiting processes that have performed join operations. If there are, the termination operation will perform a process notification, an internal operation that can be either local or remote. The remote process notification can be asynchronous because it does not matter which waiting process gets notified first. If there is more than one waiting process on a processor, the notifications can be merged to a single RPC. The reclamation of the PCB of a terminated process is usually done when its processor is free or when there is no raw PCB available for creating new processes. The termination operation is like a lazy person who finishes eating and does not intend to clean his dishes; but it does save time.

### 4.3.2 Process Synchronization

A process synchronization mechanism is a set of primitives with which clients control concurrent programs. The primitives should be transparent to processes. A process should be able to use synchronization primitives to synchro-

nize with any process without knowing which processor it is on. Any one of the currently favored mechanisms, such as semaphores [Dijkstra 68], monitors [Hoare 74], or eventcounts [Reed 79], will probably be sufficient for applications.

In a shared virtual memory system, it is possible to implement a process synchronization mechanism based on either global memory or message passing. This section uses an eventcount implementation as an example of implementing a mechanism efficiently.

An eventcount synchronization mechanism has four basic operations:

- *Init*(*ec*) — initializes an eventcount.
- *Read*(*ec*) — returns the value of the eventcount.
- *Await*(*ec*, *value*) — suspends the calling process itself until the value of the eventcount reaches the value specified.
- *Advance*(*ec*) — increments the value of the eventcount by one and wakes up awaiting processes.

After an eventcount is initialized, any process can use it without knowing where it resides.

### Global Memory Implementation

In an eventcount mechanism based on global memory, the data structure of an eventcount is stored in the shared virtual memory space. Since the shared virtual memory mapping managers guarantees that the memory is coherent, a reference to the data structure of an eventcount will be correct if each primitive operation is atomic, that is, all the operations are performed in a mutually exclusive way. Reed has given algorithms for implementing the primitives without using process queues [Reed 79]. Although these algorithms present the ideas cleanly, in real implementations, process queues are usually used.

With a process queue for each eventcount, an atomic operation implementation can be done on available architectures by using a test-and-set instruction or by prohibiting interrupts. Using a test-and-set instruction on a lock in the

shared virtual memory space is not a good idea because the instruction can cause a write page fault and a page movement when the page containing the lock is on another processor. When all the processors are trying to use test-and-set instructions on the same lock, the cost can be substantial.

Prohibiting interrupts may look like an easy way to solve the problem, but it is not. One needs to make sure that the memory references to any data structures during each operation do not generate any page faults. So, before prohibiting interrupts, all the related memory pages should be wired into memory. An efficient way to wire a page in a shared virtual memory system is to use a memory page map. Each bit in the map is associated with a memory page. Before a page in the memory is moved to another processor or disk, its associated bit is checked. If the bit is 0, the page can be moved; otherwise, the moving operation is delayed. The size of this map is not so big because it only needs to take care of the physical memory. For a 4M byte memory with 1k bytes per page, 512 bytes will be sufficient. The detailed algorithm for implementing the four primitives is omitted because it is an easy matter once the problem of atomicity is solved.

The advantage of this approach is that one does not need to worry about the transparency of eventcounts and the consistency of their data structures. By simply relying on the shared virtual memory mapping managers, more than one processor can concurrently read an eventcount while no processor is performing *Advance*. Although the approach is simple and attractive, a reference to an eventcount may cause a page fault. A page fault and a page movement usually cost more than a simple RPC. One the other hand, when a processor perfoms a number of primitive operations on the same eventcount, probably only the first one is a remote operation.

### Message Passing Implementation

In a message passing implementation, the data structures of eventcounts are stored in the private memory portion instead of the shared virtual memory

space. The primitive operations are simple RPCs that consistently access the data structures.

For this implementation, one must first solve the transparency problem. When an eventcount primitive operation executes on processor $i$, it needs to know where the eventcount is. If it is on the same processor, the operation will be local; otherwise a simple RPC should be used or the eventcount should be migrated to processor $i$.

Locating an eventcount can be done by a mapping function $M(ec, i)$ that returns a pair $(processor, local\_address)$ that tells where the eventcount is. The implementation of the function depends on how the data structures of eventcounts are arranged: statically or dynamically. A static arrangement can be either centralized or distributed. Dynamic arrangement is obviously distributed.

A simple, static, centralized arrangement is to put all the eventcounts on one processor, say process 1. In this case, the mapping function is:

$$M(ec, i) = (1, address(ec)).$$

The obvious disadvantage of this arrangement is that eventcount operations may be inefficient when many processors are trying to access eventcounts at the same time. Even if the processes are accessing different eventcounts, they all have to be done on processor 1. Another disadvantage is that many *Init* calls will be remote operations. If every processor is assumed to have the same probability of using *Init* and there are $x$ *Init* calls, then the number of remote operations is:

$$N_r = \frac{(N-1)}{N}x.$$

When $N$ is large, most of the *Init* calls are remote operations.

A static, distributed arrangement is to distribute eventcounts on different processors. Each processor reserves a space called *ec_space*, which is divided into $N$ equally sized subspaces. All the *ec_space*'s have the same base address in their private memory portions. It is convenient to make the size of a subspace

the power of 2. When an *Init* is executed on processor $i$, the eventcount is initialized in the $i$-th subspace on processor $i$. Such an arrangement is shown in Figure 4.3.

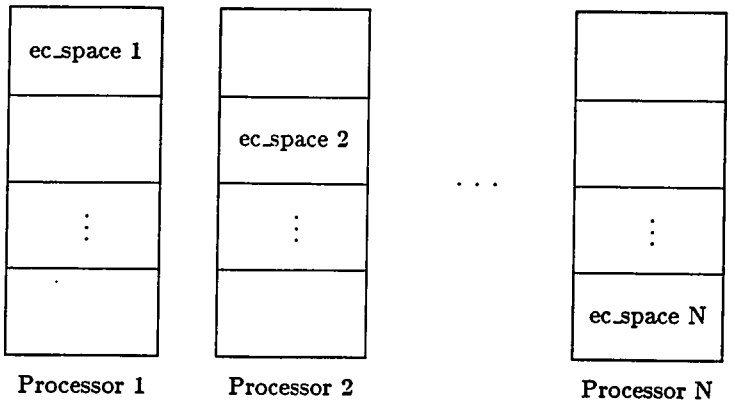


Figure 4.3: Eventcounts in private memories.

If $s$ is the size of the subspace and $b$ the base address of the *ec_space*, the mapping function can be defined by:

$$M(ec, i) = \Big(\frac{(address(ec) - b)}{s}, (address(ec) - b) \bmod s\Big).$$

After subtracting from the base address of the *ec_space*, the last $\log s$ bits of the eventcount will be the offset within the subspace and the $\lceil \log N \rceil$ bits before them will be the processor number. The size of the *ec_space* can be small because another *ec_space* can be allocated when the *Init* operation runs out of a subspace. All the initialization operations in this approach are local operations, so the mapping function is efficient.

This approach can be further optimized if every processor uses all the subspaces rather than only the $i$-th subspace on processor $i$. When a process on processor $i$ performs an eventcount operation on an eventcount residing on processor $j$, the operation saves the value of the eventcount into the place cor-

responding to its *ec_space* in the $j$-th subspace on processor $i$. Those values, though they may not be up-to-date, can then be used in the *Await* calls to reduce remote operations. If the eventcount value in the corresponding subspace is not less than the value specified in *Await*, then there is no need to get the value from where the eventcount resides.

The static, distributed arrangement is not as simple as the static, centralized one but it can solve the potential problem of inefficiency. The disadvantage is that once an eventcount is initialized, it stays. When a remote processor performs many operations on the same eventcount, they all have to be remote operations.

A dynamic arrangement allows eventcounts to migrate from one processor to another. The work involved is essentially the same as that of the memory coherence algorithm if the system allows concurrent reads while there is no write (see Chapter 2). In order to access a migrated eventcount, a forwarding pointer mechanism may be needed [Fowler 86], which, of course, requires a more complicated mapping function.

The message passing implementation of an eventcount mechanism also has the same atomic execution problem as the global memory implementation, but it does not need to wire memory pages. In this sense, a message passing implementation is simpler. In general, a message passing implementation can exploit the efficiency provided by the simple RPC mechanism, and the atomic problem is easy, but message passing introduces the complexity of the transparency problem. The static, centralized arrangement is relatively simple but it may introduce inefficiency when many processors are executing eventcount primitives. The static, distributed arrangement can solve the problem to some degree, but it requires a more complicated data structure design. The dynamic arrangement may provide the best performance but the implementation is much more difficult.

## 4.4 Dynamic Memory Allocation

When a process wants a piece of memory in a shared virtual memory address space, it must explicitly allocate it. Similarly, the allocated piece of memory should be freed explicitly when it is no longer needed.

Dynamic memory allocation for a uniprocessor environment is a well-known problem and has been studied since the beginning of operating systems development. Some simple memory allocation algorithms, such as "boundary tag" and "buddy system", were shown to be efficient in theory [Knuth 73] and are widely used in practice.

Since, from a programmer's point of view, a shared virtual memory address space on a loosely coupled multiprocessor is, in many ways, the same as a single address space on a uniprocessor, many dynamic memory allocation algorithms for a uniprocessor environment can be used directly with a simple centralized control.

This section shows how to apply a dynamic memory allocation algorithm —boundary tag—to a shared virtual memory address space and discusses the drawbacks of the direct application; it also proposes some modifications to improve the algorithm for the shared virtual memory environment. Although the discussion follows one example, it presents a general way of modifying an existing memory allocation algorithm to fit into a shared virtual memory system.

### 4.4.1 Boundary Tag

The boundary tag algorithm was designed by Knuth in 1962 and was described in his book in detail [Knuth 73]. The idea of the algorithm is to maintain a free list of memory blocks, `Avail`, so that a "first fit" or "best fit" strategy can be used to allocate a memory block. Such a list is doubly linked so that the algorithm can search blocks in any direction. A tag is put at one end of each memory block. It has four fields: state, size, forward link, and backward

link. State is a bit indicating whether the block is allocated. Size indicates the size of the memory block. The two links are for the doubly linked list Avail. To improve readability, the algorithm uses four functions STATE( addr ), SIZE( addr ), FLINK( addr ), and BLINK( addr ) to return the four fields of memory block starting at addr.

The algorithm below is taken from Knuth's book [Knuth 73] with some modifications using the conventions here. The algorithm consists of three procedures: *Initialize*, *Allocate*, and *Free*. The following is an algorithm for initialization:

**Algorithm 4.13** *Initialize*($addr, n$)

```
Avail := addr;
Rover := addr;
STATE( addr ) := FREE;
SIZE( addr )  := n;
FLINK( addr ) := addr;
BLINK( addr ) := addr;
```

It gives initial values to the free list head Avail and roving pointer Rover, and initializes the whole memory as a free memory block.

*Allocate* tries to allocate a piece of memory by sequential search following the forwarding pointers starting from the roving pointer. The algorithm essentially performs the "first fit" strategy:

**Algorithm 4.14** *Allocate*($n$)

```
IF Avail = 0 THEN RETURN fail;

addr := Rover;
WHILE SIZE( addr ) < n DO BEGIN
    IF STATE( addr + SIZE( addr ) ) = FREE THEN
        Collapse( addr, addr + SIZE( addr ) );
    ELSE
        addr := FLINK( addr );
    IF addr = Rover THEN RETURN fail;
    END;

Rover := FLINK( addr );
```

```
k := SIZE( addr ) - n;
IF k < c THEN BEGIN
    Rover := FLINK( Rover );
    BLINK( Rover ) := BLINK( addr );
    FLINK( BLINK( addr ) ) := Rover;
    END
ELSE BEGIN
    new_addr := addr + n;
    SIZE( new_addr ) := k;
    STATE( new_addr ) := FREE;
    SIZE( addr ) := n;
    END;

STATE( addr ) := ALLOCATED;
RETURN addr;
```

Note that when a block is not large enough, the algorithm merges the current block with its adjacent memory block. Since there is only one tag at one end of a memory block, there is no way to do all collapsing in *Free*. If the current block is larger than the caller needs, the algorithm will split it into two blocks, one for current allocation and the other for future allocation.

*Free* tries to collapse the freed block with its adjacent memory block. The freed block will be put into the free list.

**Algorithm 4.15** *Free(addr)*

```
STATE( addr ) := FREE;
IF STATE( addr + SIZE( addr ) ) = FREE THEN
    Collapse( addr, addr + SIZE( addr ) );
FLINK( addr ) := Avail;
BLINK( addr ) := BLINK( Avail );
BLINK( Avail ) := addr;
```

Of course, it is possible to insert the freed block at the roving pointer instead of at `Avail`.

For the shared virtual memory system, the algorithm can be used if the three operations can be done mutually exclusively. A monitor system will work just fine [Hoare 74]. Thus, correctness is not a problem.

Performance, however, is a problem. Since a tag is a part of a memory block,

any reference to a tag field may cause a page fault. When the *Allocate* procedure searches for an available memory block, it may have as many page faults as the number of blocks traversed. When collapsing two blocks or splitting a block, the procedure may also cause page faults. The upper bound of the number of page faults for *Allocate* is expressed by:

$$N_{traverse} + 2N_{collapse} + N_{split} + 4$$

where $N_{traverse}$ is the number of blocks traversed while searching, $N_{collapse}$ is the number of blocks collapsed, and $N_{split}$ is the number of blocks split. The constant 4 comes from modifying the tag of the current block, accessing `Avail`, setting the forward link of the previous block, and setting the backward link of the next block. A collapse operation may cause two page faults because it needs to modify the tag of the adjacent block and the link of the next block of the adjacent block. Similarly, *Free* may cause up to six page faults: changing the tag of the current block, changing the forward link of the previous block, accessing to `Avail`, changing the backward link of the next block, and collapsing two adjacent blocks.

The true number of page faults for *Allocate* and *Free* are hard to calculate because they depend on the number of processors in the system and where the memory blocks are used.

### 4.4.2 One-level Centralized Memory Management

One way to reduce the number of page faults in the direct method is to have a processor as the central memory allocator. All the memory allocations and deallocations are done on the allocator. The operations on processors other than the allocator need to use the remote operation mechanism to perform the operation. Since the allocation information is maintained in a monitor style, the correctness is clear. By doing this alone, it reduces the page faults caused by the references to `Avail`.

In order to reduce the number of page faults caused by the references to the tags associated with memory blocks, the tags are stored in a separate place on the allocator processor rather than at the beginning of the memory blocks. When the allocator accesses the tags, there should not be any read or write shared virtual memory page faults because they are on the same processor.

A hash table can be used as the data structure for the tags. For a given starting memory block address $x$, a hash function $TAG(x)$ will return the tag of the memory block. A tag here is a record with four fields: state, size, flink, and blink. The modification of the algorithm to use this mechanism is rather straightforward. In *Initialize*, the hash table is initialized before doing anything else. In *Allocate* and *Free*, a reference to any field of a tag requires using the hashing mechanism to get the tag first.

With this centralized memory management approach, neither *Allocate* nor *Free* will generate any shared virtual memory page fault. Obviously, it is superior to the direct method. But an *Allocate* or *Free* called from any processor other than the allocator is still a remote operation.

### 4.4.3 Two-level Centralized Memory Management

Two-level centralized memory management reduces the number of remote operations in memory allocations.

A straightforward two-level memory management is to let each processor take care of most local memory allocations. There is still a processor acting as the central memory allocator on which the memory allocation algorithm is exactly the same as before. But the memory allocation algorithm on other processors is different. Instead of making a remote call to the allocator each time, a non-allocator processor will allocate a big chunk of memory and maintain it by a local memory allocation algorithm, so that most allocations and deallocations will be local.

The local memory allocation algorithm can be anything as long as it main-

tains the big chunks of memory. For example, the boundary tag algorithm can be used again. There is no need to put the tags of memory blocks of each allocated big chunk of memory into a separate place because once a big chunk of memory is allocated, memory references inside the chunk will not cause any shared virtual memory page faults.

The local memory allocation algorithm has its own free list `Avail` and roving pointer `Rover`. Initially, the free list is empty. The first memory allocation operation will result in requesting the central memory allocator for a big chunk of memory. The proper size of memory is then allocated to the original request and the rest of the big chunk will be put into the free list. The memory allocation operations then allocate memory from the free list until the list is empty or the list does not contain any pieces that are big enough. In which case, it asks the central memory allocator again for another big chunk of memory. The alogrithm is almost the same as the unmodified boundary tag algorithm:

**Algorithm 4.16** *Allocate*($n$)

```
IF Avail = 0 THEN RETURN RemoteAllocate( n );

addr := Rover;
WHILE SIZE( addr ) < n DO BEGIN
    page := ( addr + SIZE( addr ) ) / page_size;
    IF ( ptable[ page ].access = writable )
        AND ( STATE( addr + SIZE( addr ) ) = FREE ) THEN
        Collapse( addr, addr + SIZE( addr ) );
    ELSE
        addr := FLINK( addr );
    IF addr = Rover THEN RETURN RemoteAllocate( n );
    END;

Rover := FLINK( addr );
k := SIZE( addr ) - n;
IF k < c THEN BEGIN
    Rover := FLINK( Rover );
    BLINK( Rover ) := BLINK( addr );
    FLINK( BLINK( addr ) ) := Rover;
    END
ELSE BEGIN
```

```
        new_addr := addr + n;
        SIZE( new_addr ) := k;
        STATE( new_addr ) := FREE;
        SIZE( addr ) := n;
        END;

    STATE( addr ) := ALLOCATED;
    RETURN addr;
```

The procedure *Remote_Allocate*($n$) askes the central allocator for a big chunk of memory, allocates $n$ bytes from it, and leaves the rest in the local free list. Note that memory blocks are collapsed only when the tag page of the next memory block is owned by the processor, so *Allocate* does not cause any shared virtual memory page faults. *Free* might be slow, but there might be locality of reference in it.

The size of a big chunk of memory can be chosen by the local memory algorithm. A convenient way is to divide the whole shared virtual memory address space into segments. The size of a big chunk can be determined by:

$$\left\lceil \frac{n}{segment\_size} \right\rceil segment\_size$$

where $n$ is the size of memory required by a local memory allocation operation. The size of a segment should be much less than $1/N$ of the size of the whole address space; otherwise some processors may never be able to do a memory allocation.

The deallocation algorithm is troublesome because the central memory allocator needs to reclaim memory blocks from local free lists. There are mainly two methods: eager and lazy. The eager method means that local memory deallocation procedure *Free* returns memory blocks to the central memory allocator from time to time. For example, a possible solution is to use `Max_size` to remember the maximum block size in the free list. `Max_size` may be updated in *Free* and *Collapse*. When `Max_size` exceeds the upper bound `Bound`, the block is returned to the central memory allocator. The following is the algorithm:

**Algorithm 4.17** *Free( addr )*

```
STATE( addr ) := FREE;
Max_size := MAX( Max_size, SIZE( addr ) );
page := ( addr + SIZE( addr ) ) / page_size;
IF ( ptable[ page ].access = writable )
    AND ( STATE( addr + SIZE( addr ) ) = FREE ) THEN
    Collapse( addr, addr + SIZE( addr ) );
IF Max_size > Bound THEN RemoteFree( addr );
FLINK( addr ) := Avail;
BLINK( addr ) := BLINK( Avail );
BLINK( Avail ) := addr;
```

Note that the procedure *RemoteFree* may generate up to two memory blocks because the central memory allocator only wants memory blocks starting and ending at segment boundaries.

In the lazy method, the local memory deallocation does not return any memory blocks to the central memory allocator until the central memory allocator sends requests. The algorithm can be the same as above except that it does not call *RemoteFree*. In this approach, when the central memory allocator runs out of memory blocks, it asks the local memory allocators for help. For example, the central memory allocator asks the local memory allocator one by one. When a local memory allocator receives a request, it checks to see if the largest memory block is larger than the upper bound. If it is, the memory block is given back to the central memory allocator. Obviously, the lazy method is better than the eager method if the central memory allocator rarely reclaims memory blocks because the procedure *Free* does not have a remote operation.

All the local memory management operations require mutually exclusions because there may be more than one process on each processor. If it is important to avoid mutual exclusion, the two level memory allocation mechanism can be based on processes rather than processors. In order to do so, there should be a free list and a roving pointer for each process. They can be stored in the PCB of a process. The disadvantage of this approach is that the number of local memory allocators is the same as the number of application processes. If

the number is rather large, the shared virtual memory system may run out of space easily [Knuth 73].

In general, the two-level centralized memory management mechanism benefits the shared virtual memory system because it reduces the number of remote operations. But the method introduces some complexities in memory deallocation and may waste some memory space. When the shared virtual memory system has a large address space, this is not a problem.

## 4.5 Further Optimizations

### 4.5.1 Eliminating Unnecessary Read Page Faults

A read fault of page $p$ is unnecessary if there follow be a write fault of $p$ and the distance between the two faults are within $k$ instructions, where $k$ is very small.

An instruction may generate an unnecessary read page fault when the memory references of its source and destination are on the same page. For example, instruction

```
MOVE (R1),(R2)
```

causes a read page fault by reading location `(R1)`. If location `(R2)` is on the same page as `(R1)`, a write page fault will be generated right after the control is returned from the read page fault handler.

It is trivial to eliminate the one-instruction unnecessary read page faults if there is a mechanism to indicate the destination address of the faulting instruction in hardware. A simple way is to add the following line at the beginning of the read page fault handler:

```
IF destination-page = faulting-page THEN RETURN.
```

The **destination-page** is the destination page number provided by hardware. The **faulting-page** is the faulting page number.

When the mechanism does not exist in hardware, which is true in most available systems, the only reasonable way is to compute the destination page of the faulting instruction in the fault handlers. The `destination-page` then becomes a procedure that returns the page number of the destination page of the faulting instruction. The procedure needs to interpret the faulting instruction according to the context saved. The cost of such a procedure depends on the instruction set of the target processor.

One can apply the elimination technique for one-instruction unnecessary read fault case to the $k$-instruction case if it is possible to figure out all the write fault pages within the $k$ instruction range. Whether it is worth doing so is an open problem.

### 4.5.2 Preventing Thrashing

As mentioned in Chapter 3, the shared virtual memory system may have page thrashing. The page-demand load balancing strategy in Section 3.3.4 uses a detection algorithm to discover page thrashing and reduces page thrashing by migrating processes. This section discusses how to prevent a shared virtual memory from thrashing without doing process migrations.

Recall that traditional virtual memory systems used to have the memory thrashing problem until the *working set* concept was developed [Denning 80]. The working set idea is based on the "principle of locality" of sequential programs [Denning 72] and its implementation significantly improves the performance of the multiprogramming environment on a uniprocessor.

The working set concept does not prevent thrashing on shared virtual memory systems. There is only one heavyweight process for each application program, so the shared virtual memory system mainly considers paging between lightweight processes. Another reason is that the degree of data sharing is closely related to the parallelism of a program. If the processes in a parallel program share data and the system implementation inhibits the processes from

sharing according to the traditional virtual memory working set idea, parallelism may be restricted.

In addition to the page-demand load balancing strategy mentioned in Chapter 3, a concept called *working time* in the memory coherence mapping helps prevent a shared virtual memory system from thrashing. The working time of a page on a processor is the time between receiving and relinquishing the ownership of the page. By using the page thrashing detection mechanism described in Section 3.3.4, the shared virtual memory system can figure out whether a requesting page is thrashing or not. If the page is thrashing, the shared virtual memory system can delay the requests for relinquishing the ownership until the working time reaches a lower bound. If the page is not thrashing, the ownership is relinquished immediately. Such an algorithm is not difficult to implement when the page-demand process scheduling strategy is used because the thrashing detection mechanism already exists.

Another way to prevent a shared virtual memory from thrashing is to modify the dynamic memory allocation algorithm such that every allocation operation always allocates a piece of memory starting and ending at page boundaries. In other words, the minimum unit of memory allocation is a page. This simple method effectively eliminates the page thrashings caused by frequently accessing variables that used to be in the same page. For example, if two variables $v_1$ and $v_2$ are in the same page $p$ and process 1 accesses $v_1$ many times while process 2 is accessing $v_2$, page $p$ will start to thrash. However, if the variables are in different pages, this thrashing will not occur. Thus, one may allocate different pages for the variables accessed by different processes. This method does not restrict the programmer to putting the different variables into the same page. A programmer can always use a record to hold many variables and allocate a piece of memory for the record. Another alternative is to provide two kinds of allocation calls, one for allocating memory in pages and another for allocating memory in bytes.

So far, we have seen three ways to prevent a shared virtual memory from

thrashing:

- page-demand load balancing strategy which migrates processes to reduce memory page thrashing,
- page working time which guarantees that each processor holds a page for a certain period of time when the page is thrashing, and
- using distinct pages for different variables.

Although these methods are proposed for the implementation of a shared virtual memory system, it is probably a good idea to use empirical data to justify them first.

### 4.5.3 Indirect Memory Reference

Allocating memory in pages can only reduce the memory page thrashing when variables share the same page. When processes frequently access the elements of vectors and arrays, memory page thrashing may occur. In this case, allocating memory in pages does not help. This section presents a strategy of trading memory space for efficiency—indirect memory reference.

Let's start with vectors. An indirect memory reference vector consists of two parts, an address vector and a data domain. The address vector is a normal vector with the same number of elements except that each element is the address of the corresponding data element in the data domain. The address vector uses a consecutive piece of memory. The data domain of an indirect memory reference vector does not require a consecutive piece of memory. An application program decides how to allocate memory for the data domain.

Normally, allocation of memory for the data domain of an indirect memory reference vector depends on how to split the data structures in the application programs. If a process mainly accesses $k$ elements in a vector, the data domain may allocate memory $k$ elements at a time so that the vector does not have unnecessary shared memory pages among processes. Figure 4.4 shows the mapping between the address vector $a$ and the data pages $d_1$ and $d_2$, which

are on two different pages. The algorithm for initializing an indirect memory reference vector is too simple to list here.

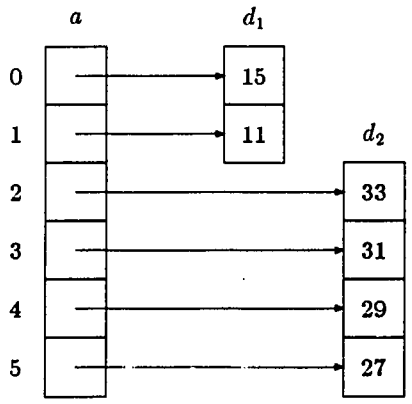


Figure 4.4: An indirect memory reference vector.

Since an array can be constructed by vectors, it is easy to build an indirect memory reference array by indirect memory reference vectors. It is also possible to construct indirect memory reference records by using an address vector and a data domain.

The tradeoff of indirect memory reference data structures is clear. In terms of space, this approach requires an additional space for address vectors. When the size of a data element is not small, the space for an address vector is not large. In terms of time, this approach needs one more memory reference for each reference to an element in an indirect memory reference data structure than in a traditional data structure, but this approach reduces the memory contention which may cause memory page thrashing. The gain of reducing memory contention is probably much more than the extra cost of using an address vector because the cost of a memory reference is usually not comparable with the computation cost related to a data element. Therefore, the package for building indirect memory reference data structures is a good thing to have

in a library for application programmers to optimize their parallel programs.

## 4.6 Conclusions

This chapter has studied many engineering issues related to the shared virtual memory implementation. The discussions on implementation environments have raised some system design requirements that benefit implementing shared virtual memory systems. On the other hand, the minimum system requirements show that it is possible to implement a shared virtual memory system on most existing systems. A simple RPC protocol has been informally presented for the more efficient implementation of the shared virtual memory mapping managers and process control primitives.

This chapter has shown how to implement shared virtual memory mapping managers in different modes and how to support multiple address spaces. Several page table compaction methods have been proposed and compared. For a large-scale multiprocessor with a small memory on each processor, page table compaction is necessary. Page replacement algorithms have been presented in detail. The discussion on memory replacement covers how to design a page replacement algorithm for a shared virtual memory system and how to integrate a shared virtual memory system with an existing virtual memory system. The page replacement algorithm without using secondary storage is useful for implementing a shared virtual memory system on a large-scale multiprocessor. The algorithm enables system designers to build a shared virtual memory almost as large as the sum of all the physical (or virtual) memory spaces.

Implementation issues of process control primitives that are not in Chapter 3 have been studied here. Different approaches to implementing synchronization mechanisms are given by showing how to implement an eventcount mechanism.

It has been shown through an example that it is not appropriate to move a uniprocessor dynamic allocation algorithm to the shared virtual memory system without modifications. The ideas for efficiently implementing dynamic

allocation algorithms have been presented by modifying a well-known sequential allocation algorithm, the boundary tag algorithm. Those ideas apply to other existing algorithms, but these applications are not discussed.

Finally, I proposed optimization methods to improve system performance. Eliminating unnecessary read page faults can be done by software, whereas it is a trivial problem if hardware support exists. The methods for preventing page thrashing are rather simple and fit with the page demand load balancing strategy. The idea of indirect memory reference is an application program optimization technique. It can be used by compiler designers if a parallel programming language is ever built for a shared virtual memory system in the future.

Many problems remain open. In particular, since I have had no practical experience implementing a shared virtual memory system on point-to-point connection multiprocessors, the implementation issues on those architectures are probably not covered adequately.

# Chapter 5

# IVY: A Prototype

This chapter describes IVY—an Integrated shared Virtual memory system developed at Yale. Since Chapter 4 has presented and compared many of the engineering issues of implementing a shared virtual memory system, this chapter only describes the implementation of IVY with a few comments on design decisions.

## 5.1 Overview

IVY is a shared virtual memory system developed for experimental purposes. It has evolved through several stages. At the beginning of the research (in the spring of 1984), the idea of shared virtual memory system was premature. During the summer of 1984, many people at DECSRC offered helpful suggestions. Butler Lampson suggested first looking at the memory reference behavior of some programs using shared memory, because it was not clear if a shared virtual memory implementation on a loosely coupled multiprocessor would be able to deal with many practical parallel programs.

Following Lampson's suggestion, I looked at the assembly code of some procedures for partial differential equations and found that a shared virtual memory implementation was plausible. Instead of trying to verify this theoret-

ically, I implemented a simple shared virtual memory based on the centralized distributed manager algorithm. IVY I, which took a few months to implement, took the fullest possible advantage of the Apollo operating system, Aegis, without modifying its kernel code. The system performance of the implementation was not satisfactory.

After modifying some of the kernel code of Aegis, I built IVY II based on the centralized distributed manager algorithm. It performed well enough to run some real experiments. While implementing IVY II, I also developed and implemented other memory coherence algorithms: the improved centralized manager algorithm, the dynamic distributed manager algorithm, and the fixed distributed manager algorithm.

The most recent version of IVY II is a shared virtual memory system implemented on top of the modified Aegis operating system. Ivy consists of 5 modules, namely, simple RPC, memory mapping, process management, memory allocation, and initialization. The hierachy of the system is shown in Figure 5.1.

The three top modules in the hierachy form the IVY client interface. Each consists of a set of primitives that can be used by application programs. The primitives are together in a library file. There are four library files of this kind, one for each memory coherence algorithm. One can produce an IVY image by compiling a program and binding it with the desired library. The IVY image file can be executed on any number of nodes in the network.

Starting IVY is simple. A startup program initializes IVY on the nodes listed in a specification file which can be either edited by hand or generated by a program (nodes that are usually idle for long periods of time are usually chosen for running IVY). IVY's initialization usually takes less than a minite because it can be done in parallel on all the nodes.

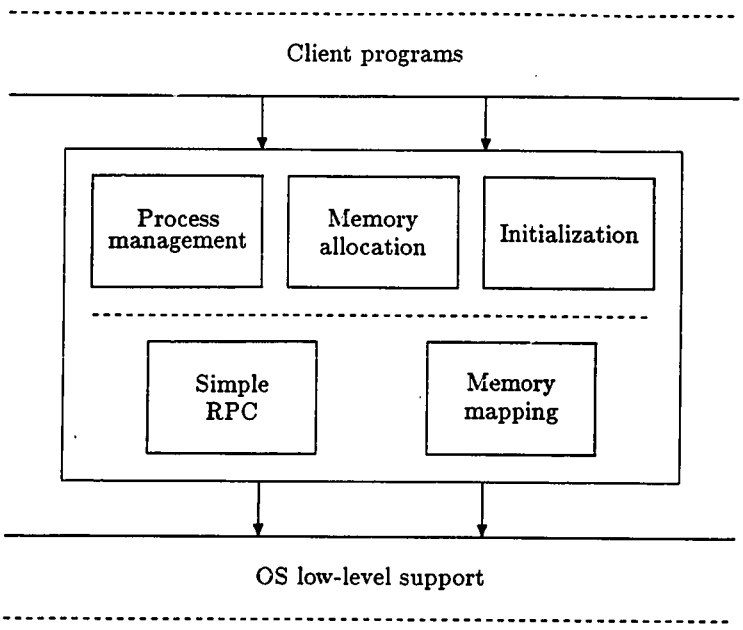


Figure 5.1: IVY hierachy.

## 5.2 The Apollo DOMAIN Environment

IVY was developed in the Apollo DOMAIN environment [Apollo 81, Leach 82, Leach 83]. DOMAIN is an integrated environment of personal workstations and server computers connected by a 12M bit/sec baseband, single token ring network. The DOMAIN environment is based on an object-oriented single-level store system that presents a flat space of objects addressed by unique identifiers (UIDs). A program accesses any object by presenting its UID and asking for it to be mapped into the program's address space; subsequently, it is accessed with ordinary machine instructions. The whole main memory on a node acts as an object cache based on demand paging. The DOMAIN environment allows nodes to share objects (not memories). The true single-level store implementation is a feature that makes the DOMAIN environment

unique among commercial workstations.

The DOMAIN virtual memory architecture presents a virtual address space of $2^{24}$ (16M) bytes in which half of the space is reserved for the operating system; user-mode virtual memory space is about 8M bytes. Each user address space is divided into 1K byte pages. For a page to be usable it must be *mapped* to an object (disk file). The smallest amount of virtual address space that can be mapped is a segment of 32 memory pages.

The Memory Management Unit (MMU) of the MC68000 architecture provides each memory page frame with a set of rights (nil, execute, read, write). They are checked at each memory reference. A memory reference to a page with an incorrect access right results in a hardware exception.

Although the ring network supports data communication at the rate of 12M bit/sec, the real communication speed between the memories of two MC68000 processor nodes is about 1M bit/sec [Leach 83]. My experiments confirmed this rate.

The operating system of the DOMAIN environment, Aegis, supports multiple heavyweight processes (it is expected that later Apollo will later support lightweight user processes). Each process has its own virtual memory address space. A program in user mode can have its own fault handlers processing hardware exceptions. A hardware exception in this case switches from its system mode to user mode and proper context savings are also involved. The basic process synchronization mechanism in Aegis is the *eventcount* [Reed 79].

IVY is developed on an Apollo ring in which most nodes use a Motorola MC68000 or MC68010 microprocessor [Motorola 84] with either 1.5M or 2M bytes of main memory. In order to make the implementation environment consistent, IVY only operates on the MC68000 processor nodes with private disks. The basic hardware support of the Apollo architecture has made the implementation of a shared virtual memory system possible.

## 5.3 The Simple RPC

The simple RPC mechanism handles all the remote operations of other modules. The performance and the reliability of the shared virtual memory system depend heavily on this mechanism. My experience shows that different implementations yield rather different results.

### 5.3.1 Basic Mechanism

The simple RPC mechanism in IVY is based on sending and receiving packets and the faulting (or exception) handling mechanism.

The mechanism for sending and receiving packets in the Aegis operating system is the lowest level means that programs in user mode can use to communicate between processes on different nodes. A send operation sends a packet to a socket of a user process address space of another node. One can also send a broadcast packet to the sockets with the same socket number on all the nodes. When a packet is sent, it will be copied into a system memory buffer first on the sending processor and the low-level kernel code will send it onto the network. If the socket on the destination processor is opened, the packet will be picked up and put into the specified socket and advance its associated eventcount. A receive operation always awaits the eventcount of a socket, so if the process is doing a receive before the packet arrives, it will be suspended until the eventcount is advanced. In this case, the packet is moved from the buffer to a specified area in the user space.

The existing faulting mechanism in the Aegis operating system has two kinds of faults: synchronous and asynchronous. Synchronous faults are generated by the process itself. For example, an illegal instruction or access violation are synchronous. Asynchronous faults are produced from outside of the process. For example, one process can send a "trace fault" to another process by setting its trace bit [Motorola 84].

The implementation of the simple RPC module in IVY I used the existing

Aegis faulting mechanism in a straightforward way. Since the Aegis operating system supports only one thread for each user process (or address space), the implementation used a *helper* process on each processor to receive all the incoming packets. The lightweight processes of the shared virtual memory system were implemented in another heavyweight process called the *main* process. The helper process was simple; it waits for an incoming packet at all times and sends a trace fault to its main process if a new one is received.

In this mechanism, a synchronous simple RPC has five steps:

1. The lightweight process in the main process on processor 1 sends a request to processor 2 and suspends itself.
2. The helper process on processor 2 receives the request in the area shared with its main process, and sends a trace fault to the main process.
3. The main process on processor 2 dispatches the request to a server process, and the server process processes the request and sends a reply packet back to processor 1.
4. The helper process on processor 1 receives the reply in the area shared with its main process, and sends a trace fault to the main process.
5. The main process on processor 1 receives the fault in its trace fault handler, processes the reply, and resumes the requesting lightweight process.

The area that the helper process shares with its main process can be rather large because the sharing is done by mapping the same file into the two address spaces.

The implementation of this mechanism is clean, but its performance is inadequate. The helper process pretends to be a lightweight process but it is not. The cost of a synchronous simple RPC is roughly expressed by:

$$C_{rpc} = 2C_s + 2C_r + 4C_{light} + C_{processing} + 4C_{heavy} + 2C_{trace} \qquad (5.1)$$

where $C_s$ is the cost of sending a packet, $C_r$ is the cost of receiving a packet, $C_{processing}$ is the cost of processing a packet, $C_{light}$ is the cost of a lightweight process context switch, $C_{heavy}$ is the cost of a heavyweight process context switch,

and $C_{trace}$ is the overhead of the trace fault mechanism. The first four terms in Equation 5.1 are obviously necessary. However, the four heavyweight process context switches are from the two receive operations by the helper processes and the two trace faults, and can cost as much as 10 milliseconds. A simple RPC doing nothing takes about 21 milliseconds. Clearly, such performance is not satisfactory.

An improved version of the simple RPC mechanism uses two *ad hoc* tricks to reduce overhead. The first trick is to enlarge the shared area between the helper process and the main process to hold more information. Thus, processing a simple RPC request is done in the helper process instead of in the main process. This eliminates one trace fault and one heavyweight process context switch. The second trick is to process all the simple RPC requests in the helper process without using lightweight server processes. This eliminates two lightweight process context switches.

Thus, in the improved mechanism, a simple RPC has four steps:

1. The lightweight process in the main process on processor 1 sends a request to processor 2 and suspends itself.
2. The helper process on processor 2 receives the request, processes the request, and sends a reply to processor 1.
3. The helper process on processor 1 receives the reply in the area shared with its main process, and sends a trace fault to the main process.
4. The main process on processor 1 receives the fault in its trace fault handler, processes the reply and resumes the requesting lightweight process.

Although this version is conceptually simpler than the first version, it is more complicated in implementation because there are more shared data structures between the helper process and the main process. A simple RPC doing nothing takes about 16 milliseconds with this new strategy, which is still slow.

After trying the above implementation, I decided to modify the Aegis kernel to further improve performance. I call this implementation IVY II. The goal was to let the Aegis operating system generate a trace fault to a process

when a packet arrived so that the simple RPC processing could be done in the trace fault handler of the main process. In order to do this, the following modifications were made to the Aegis operating system:

- The data structure of the socket was changed to hold information about raising trace faults.
- A new system call was added to intialize the socket so that the system will raise a trace fault when a packet arrives in the socket.
- The interrupt handler of receiving a packet raised a trace fault if the socket was marked as raising trace faults.

These modifications eliminated the helper process from the simple RPC mechanism.

The implementation in IVY II does not have any heavyweight process context switches and the implementation is rather clean. A simple RPC doing nothing takes about 10 milliseconds. The performance of the simple RPC mechanism is still poor compared with a good RPC implementation [Birrell 83] because it has many mode switches, implicit context switches, and extra memory copying. This is the limitation of a user-mode implementation, but the resulting system is still adequate to demosstrate the feasibility of a shared virtual memory.

### 5.3.2 Protocol Implementation

The protocol of the simple RPC mechanism follows the description given in Chapter 4 except that the simple RPCs are processed in fault handlers instead of server processes. In the data structure for implementing the protocol in IVY, there are two types of channels: outgoing and incoming. An outgoing channel is associated with a receiving processor and records the information about the outgoing requests to the processor. Similarly, an incoming channel is associated with a requesting processor and records the information about the incoming requests from the processor.

When a request is sent through an outgoing channel, it gets a transaction ID (TID) from the channel. Therefore, a request can be uniquely identified by its requesting processor number and its TID. An outgoing channel has six fields:

- *current tid*—the tid of the last request,
- *request pid*—the PID of the requesting process,
- *request*—pointer to the last request packet,
- *reply set*—the set of the processors that need to send replies,
- *acked tid*—last acknowledged request tid, and
- *queue*—for the awaiting processes.

The current tid field is used to generate a TID and is incremented by one every time a new request is sent. When a reply to the last request is received, its processor number is deleted from the reply set. The request is completed only when the reply set is empty.

The simple RPC mechanism only allows one outstanding asynchronous request for each outgoing channel. This means that each processor allows one outstanding asynchronous request to the same destination processor, or $N$ outstanding asynchronous requests to different processors.

An incoming channel has three fields:

- *received tid*—the tid of the last request received,
- *reply tid*—the tid of the last reply sent out, and
- *reply*—pointer to the last reply packet.

The incoming channel always keeps its last reply packet because if the reply is lost, reprocessing the request and generating a new reply would be incorrect unless no state was changed. The data representation of a reply set is the same as a copy set so that a broadcast invalidation request can simply copy its copy set to the reply set field.

All the packet buffers are allocated from a buffer pool by using a straightforward round-robin algorithm. If the buffer pool runs out of space, then another

buffer pool is allocated. The initial buffer pool has 128 buffers. In all my experiments, IVY has not run out of buffers.

IVY uses different channels for broadcast requests and forwarding requests. It is possible to use the same channels as the normal one-to-one requests, but the implementation is a little more complicated. Retransmission checking is done in a null process, which checks all the outgoing channels every half second when there is nothing to do.

## 5.4 Shared Virtual Memory Mapping

Although the Apollo MMU hardware supports a page-level protection mechanism, the Aegis operating system does not offer any system call to set the page access rights. The only way a user program can change these access rights in its address space is to map a file (object) into some area with a given access right. The smallest unit of mapping is a segment of 32 pages (or 32K bytes).

The memory mapping managers in IVY I was implemented based on mapping objects and on the user-mode fault handling mechanism provided by the Aegis operating system. Thus, the size of a memory synchronization unit is a segment. In order to avoid moving the whole segment across the network for every page fault, IVY uses a distinct object for each segment. When a processor has a read fault on a page, the owner processor of the segment containing the page changes the access right to the segment to read-only and the faulting processor maps the object into the address space. The pages in the segment will be paged in on demand. Similarly, a write page fault results in mapping in the associated object with writable access. The pages in the segment are paged in on demand.

Although the object-memory mapping and the demand paging facility can prevent the unnecessary movement of a great amount of data, memory contention is a problem when processes in a parallel program modify a lot of shared data concurrently. The overhead of mapping and unmapping an object is also

large because the Aegis operating system was not designed for this purpose.

In order to improve the performance, I modified the Aegis kernel to add system calls that can set access rights directly. The modifications include:

- changing the virtual memory table to maintain the information about the access right for each page,
- changing all the programs affected by the virtual memory table so that they will work correctly with the new virtual memory table, and
- adding new system calls to set access rights for the pages in user address spaces.

These modifications were non-trivial.

With the modifications to the Aegis operating system, the IVY II implementation can use the page size (1K bytes) as its memory synchronization unit size. Since the data portion of a packet can be as long as 1K bytes and the header portion can be as long as 512 bytes, the protocol information of the simple RPC can be put into the header portion and the page can be put into the data portion. Moving a page usually does not require any additional messages.

The Apollo user address space is divided into two portions. The shared virtual memory address space is in the high portion and the private memory is in the low portion. The data structure of the page table is a vector of records and each record is a table entry. The page table entries are not compacted and the whole table is stored in the private memory. The page replacement of the page table relies on the existing virtual memory page replacement mechanism in the Aegis operating system.

There is no special page replacement algorithm for the shared virtual memory address space. As discussed in Chapter 4, page replacement between main memory and disk is unfair. Further improvement of page replacement was not implemented in IVY. This optimization should be done in the future.

## 5.5 Process Management

The process management module consists of all the operations for process control, process migration, and process synchronization. Although some operations have been through several implementation stages, further improvements can still be done. See Chapter 4 for more design choices.

### 5.5.1 Processes and Process Scheduling

All the processes in IVY are lightweight. The program code of a process is stored in its private memory; therefore, there is no need to build a dynamic loader. The stack of a process is allocated from the shared memory portion. Each process has a process control block (PCB) that contains necessary information like process state, stack, context, and other process control-dependent information. The PCBs are stored in the private memory of the address space. Therefore, the PID of a process is simply represented as a pair—processor number and the address of its PCB.

The process state field in a PCB has four attributes: *raw, ready, migratible,* and *migrated*. The raw attribute is set if and only if the PCB is initialized and ready for a process creation. The ready attribute is set if the process is running or ready to run. The migratible attribute is set if the process can be migrated to another processor. Application programs can modify this field by using a primitive so that a migratible process can become non-migratible or vice versa at run time. When a process is migrated, a forwarding pointer is put into its PCB and the migrated attribute is set. The PCBs of migrated processes are never collected.

At the initialization stage, a fixed number of PCBs are created and initialized so that process creation requires very little binding work. The dead PCBs generated by process terminations are not collected until no free PCB is available for a process creation.

Since IVY is implemented in user mode, all the lightweight processes are

implemented in a heavyweight process. Hence, there is no disk I/O overlap among lightweight processes. Nevertheless, lightweight processes have overlaps on synchronous simple RPC operations.

The process scheduling mechanism is built to be simple. Each processor has a local ready queue using a last-in-first-out policy. Processes do not have priorities; the process dispatcher is invoked only when a process is suspended. The process dispatcher always picks up the process in the front of the ready queue. If there is no ready process available, the dispatcher runs a system process called the null process.

The null process implements the passive load balancing algorithm. It normally waits on two low level eventcounts, one for timeout and another for new ready processes. The null process is invoked when either of them is advanced. When a timeout event occurs, the null process will run the passive load balancing algorithm (see Chapter 3). The eventcount for new ready processes can be advanced only when

- a process is migrated to the current processor,
- a remote resume operation is performed, or
- a remote notification operation results in waking up a process.

Of course, when a new ready process is available, the null process will suspend itself. The dispatcher will the do another schedule.

While implementing the process scheduling algorithm, I learned that, for many application programs, the algorithm will not work well if the number of ready processes on each processor is used as the only criterion for migrating processes. A better way is to use the number of processes (including both ready and suspended) controlled by an upper bound and a lower bound as a criterion.

### 5.5.2 Process migration

Only a ready, migratible process can migrate from one processor to another. Since PCBs are stored in the private memory portion of the address space, a

process migration must

- send the PCB of the process to the destination processor and put it into a PCB,
- copy the current page of the process's stack to the destination processor and transfer the ownership of the page,
- transfer the ownership of all the pages in the upper portion of the stack to the destination processor, and
- put the PCB in the ready queue on the destination processor.

The reason for moving the current page of the process's stack is to avoid a page fault in the process dispatcher (Figure 5.2). If a stack page fault occurs in the dispatcher, the fault is a system-mode fault which results in saving the current context to the system stack and then copying to the user stack in order to pass control to user mode. Copying to the user stack generates another fault. Since this is an infinite loop, a fault like this will cause the system stack to run out and eventually crash the heavyweight process.

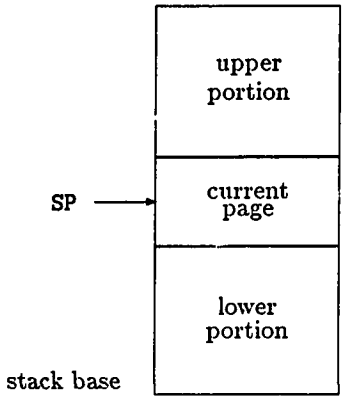


Figure 5.2: A process stack.

The upper portion of the stack need not move to the destination processor because its content is meaningless. Ownership transfer is inexpensive because

it only requires setting the protection bits of the page frames. There is no need to do anything with the lower portion of the stack because the stack can grow without having further page faults after the current page and the upper portion of the stack become writable.

### 5.5.3 Eventcount Implementation

Eventcount [Reed 79] is the process synchronization mechanism in IVY. Eventcounts were implemented in IVY primarily because the Aegis operating system uses eventcounts as its synchronization mechanism.

There are two levels of eventcounts in the Aegis operating system, one for system mode and the other for user processes. These mechanisms only work among the processes within a node. Although it is possible to extend the user process eventcount to the network-wide case, doing so requires modifications to the operating system. Therefore, I decided to implement an eventcount mechanism only for the processes in the shared virtual memory address space.

The first eventcount implementation was based on message passing, as discussed in Chapter 4. This implementation was designed with IVY I because moving a page in IVY costs too much. But, since the data structures of eventcounts are stored in the private memory, the implementation was not clean.

The eventcount implementation in IVY II is based on shared virtual memory. Curiosity about the locality of eventcount operations motivated this implementation. The atomic operation is implemented by wiring memory pages and using test-and-set instructions. It turns out that the shared virtual memory implementation is much cleaner than that based on message-passing; furthermore, the performance is better when there is more than one process on each processor because eventcount primitives become local operations when the eventcount data structure has been paged into the local processor.

The queue nodes for the eventcount data structure are allocated from the same page of the eventcount itself unless it runs out of space in the page, in

which case, allocation is done in a new page. The queue node pages for the same eventcount are linked together. This mechanism increases the locality of the eventcount data structure. In most cases, no additional paging is needed for queue node allocation.

## 5.6 Memory Allocation

IVY has a simple memory allocation module that uses a "first fit" algorithm with one-level centralized control. The processor that the user directly contacts will be appointed to the centralized memory manager. The algorithm is almost exactly the same as that described in Chapter 4. The only memory allocation optimization in IVY is to allocate each piece of memory to the boundary of a page.

Both allocate and free are atomic operations. IVY uses a binary lock on each processor for memory allocation purposes. At the beginning of each memory management primitive, a test-and-set operation is performed on the lock. A failed process will be put into a queue and will be awakened by an unlock operation on the lock which is done at every end of each primitive.

The two-level memory management was not implemented, though it is expected to have better performance.

## 5.7 Programming in IVY Environment

Programmers can use any programming language in the Apollo DOMAIN to write parallel programs as long as they can interact with the procedure calls in the Apollo DOMAIN Pascal in which IVY is implemented. Since all the languages in the Apollo DOMAIN are designed for sequential programming, the programmer has to program parallel constructs explicitly with the primitives provided by IVY. This section addresses several issues of writing parallel programs in Pascal.

## Shared Data Structures

The Apollo DOMAIN Pascal compiler implements all the common language features with some extensions to serve system programming. Since Pascal is a strongly typed language, the programmer has to declare the data types of shared data structures in order to put data into the shared virtual memory.

To make the programming task easier, IVY uses the WITH statement to treat the data structures in the shared virtual memory syntactically as normal data structures. For example, a program has two shared data structures foo and bar. The programming convention in IVY requires the grouping of all shared data structures into one record:

```
TYPE
    fooType = ARRAY[ 1..1024 ] OF CHAR;
    barType = Integer;

    sharedType = RECORD
        foo: fooType;
        bar: barType;
        END;

    sharedPointer = ^sharedType;
```

In the main program, there is a variable sharedDataPointer defined to be the pointer to the shared record, so that every processor can simply share all the data structures by using the shared record pointer. The main program of the parallel program allocates a piece of memory for the shared data structure:

```
    ...
VAR sharedDataPointer: sharedPointer;

    ...
Allocate( sharedDataPointer, Sizeof( sharedDataPointer^ ) );
    ...
```

To use the shared data structures, a procedure uses WITH to enclose the piece of program:

```
PROCEDURE baz;
```

```
BEGIN

    WITH sharedDataPointer^ DO BEGIN

        ...
        x := foo[ 10 ];
        ...

    END;
END;
```

Note that the program references the shared data structure `foo` exactly as it does a normal variable.

## Parallel Constructs

Parallel constructs are created by using the primitives provided by the process management module. The programmer chooses how to schedule processes when calling an initialization procedure at the beginning of her program. There are two options: manual scheduling and system scheduling. If system scheduling is used, the programmer only needs to create and terminate processes. But if manual scheduling is chosen, the programmer needs to take care of process migration as well.

It is the programmer's responsibility to program process synchronization. The methodology of such programming is the same as that of "conventional" concurrent programming developed since the 1960s. The programmer only needs to keep in mind that she is writing a concurrent program that runs on a traditional operating system on a uniprocessor. Although there is no parallel programming language, such a primitive environment has proven to be convenient enough to write benchmark programs.

### Debugging

IVY does not have any special debugging tools. Debugging programs is usually done on a single processor. Since an IVY image file can run on any number of processors, there is no need to have a simulator. My experience with IVY shows that if a program follows IVY parallel programming conventions, debugging on a single processor is usually successful. After debugging on a single processor, the programmer should debug her program on two and then three processors. My experience indicates that if a program can run on three processors correctly, there are few bugs left.

## 5.8 Experience

As mentioned early in this chapter, the algorithms in the implementation of IVY are not complicated. In fact, there are only 4669 lines of code; most of the programs are written in Pascal and a few of them are written in assembly code. This number includes the code for implementing different memory coherence algorithms and debugging options. It does not include the modifications to the Aegis operating system. However, the task of making the system work correctly is far more subtle than it may appear.

While debugging, I usually used two examples: consumer-producer and parallel Jacobi algorithm. Even for the consumer-producer example, I encountered bugs that appeared only when running the system on more than two processors. When I inserted some statements to print out related states or save those states to print out later, the bugs disappeared because the timing of communications and interrupts were different. Since MIMD multiprocessors are asynchronous, there were also bugs that were not always reproducible. Lacking information and tools, the only way to deal with these kind of bugs is to think.

The main complexity in debugging such a system is that one needs to consider possibilities in a multiple dimensional space. Humans are trained to think

sequentially; this is especially true for those of us who have written sequential programs for many years. A methodology for parallel program engineering and debugging tools for parallel programs are highly desirable.

## 5.9 Remarks

My experience implementing IVY shows that, although it is possible to implement a shared virtual memory without modifying an existing system like the Aegis operating system, it is necessary to modify the existing system to get acceptable performance. While the performance of IVY II is acceptable, there is much room for improvement. The implementation has shown that a user-mode implementation has a lot of overhead and that a system-mode implementation ought to provide a substantial improvement. A well-tuned system-mode implementation should improve the performance of the simple RPC and page moving by a factor of at least two according to the performance comparison with some well tuned system like the V kernel [Cheriton 84, Zwaenepoel 85]. I/O overlaps among the lightweight processes do not exist in IVY. An integrated heavyweight and lightweight process scheduler is highly desirable. The disk I/O overlap may also greatly improve IVY's performance. The page replacements for both page tables and the shared virtual memory space are not fair. To achieve a fair page replacement, one needs to implement the algorithm mentioned in Chapter 4.

# Chapter 6

# Experiments

As stated at the beginning of this disertation, my thesis is: Shared virtual memory on a loosely coupled multiprocessor can achieve orders-of-magnitude speedups over a uniprocessor for many parallel programs, and it is practical to implement on existing architectures. A reasonable way to justify the thesis and to compare some design choices is to run parallel programs on the prototype shared virtual memory system described in the previous chapter. In this chapter, I present three kinds of statistical data for the experiments performed on the prototype: speedups, memory coherence algorithm comparison, and shared virtual memory reference miss ratio. The experimental results strongly support my thesis.

## 6.1 Applications

Given the difficulties of finding practical parallel programs, the only reasonable way to do experiments is to select a set of application programs from different fields as a benchmark suite. In choosing benchmark programs, I used the following two features as criteria:

- *Reasonably fine granularity of parallelism.*

Parallel programs with rather coarse granularity can obviously perform well in the shared virtual memory system.

- *Side-effects in shared data structures.*

  There are parallel functional programs that do not have any side-effects in their data structures at run time[Hudak 86a]. The shared virtual memory system is clearly a big win in these applications, so I did not choose them as benchmarks.

The main goal in using these criteria is to avoid weighing the experiments in favor of the shared virtual memory system by picking problems that suit the system well. A better test is to select problems that do not suit the system well.

The benchmark set in the experiments consists of six parallel programs that are written in Pascal. All of them are transformed manually from sequential algorithms into parallel ones in a straightforward way.

## Linear Equation Solver

This is a parallel Jacobi algorithm for solving linear equations. The algorithm is transformed from the traditional, sequential Jacobi algorithm that solves the linear equation $Ax = b$ where $A$ is an $n$ by $n$ matrix. In each iteration, $x^{(k+1)}$ is obtained by

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)/a_{ii}.$$

The parallel algorithm creates a number of processes to partition the problem by the number of rows of matrix $A$. All the processes are synchronized at each iteration by using an eventcount. The data structures $A, x$, and $b$ are stored linearly in the shared virtual memory, and the processes access them freely without regard to their location. Such a program is much simpler than that which results from the usual message-passing style program, because the

programmer does not have to perform data movement explicitly at each iteration.

## 3D PDE Solver

This is a parallel Jacobi algorithm for solving three dimensional partial differential equations (PDEs). The algorithm and its transformation are the same as the linear equation solver except that in the equation $Ax = b$, $A$ is a special sparse matrix "coded into the program" rather than stored in the shared virtual memory. The vectors $x$ and $b$ are stored in the shared virtual memory, and the processes access them freely without regard to their location. Since matrix $A$ does not need space in the shared virtual memory, the program executes differently from the linear equation solver.

## Sorting

The parallel sorting algorithm implements block split-merge sort; more specifically, a variation of the block odd-even based merge-split algorithm described in [Bitton 84]. The sorted data is a vector of records that contain random strings.

At the beginning, the program divides the vector into $2N$ blocks for $N$ processors, and creates $N$ processes, one for each processor. Each process sorts two blocks by using a quicksort algorithm [Hoare 62]. This internal sorting is naturally done in parallel. Each process then does an odd-even block merge-split sort $2N - 1$ times. The vector is stored in the shared virtual memory, and the spawned processes access it freely. Because the data movement is implicit, the parallel transformation is straightforward.

## Dot-product

The dot-product program computes

$$S = \sum_{i=1}^{n} x_i y_i.$$

A number of processes are created to partition the problem. Process $i$ computes a sum

$$S_i = \sum_{j=l_i}^{u_i} x_j y_j.$$

$S$ is obtained by suming up the sums produced by the individual processes:

$$S = \sum_{i=1}^{m} S_i$$

where $m$ is the number of processes. Both vector $x$ and $y$ are stored in the shared virtual memory in a random manner, under the assumption that $x$ and $y$ are not fully distributed before doing the computation. The main reason for choosing this example is to show the weak side of the shared virtual memory system; dot-product does little computation but requires a lot of data movement.

## Traveling Salesman Problem

The traveling salesman problem is to find a tour that visits each city once with the minimum cost. The cities are represented as the nodes in an undirected graph. Each edge in the graph is assigned a random weight. The cost of a tour is the sum of the weights of the edges on the tour.

The program is a parallel branch-and-bound algorithm that finds a tour with the minimum cost. The branch-and-bound strategy used in the program is a simplified version of the one proposed by Heid and Karp [Heid 70]. At each step, an 1-tree (a variation of the minimum spanning tree) of the remaining graph is computed. The sum of the cost of the subtour and the 1-tree is compared with the cost of the current least upper bound. If the cost is less than the upper bound, it will replace the upper bound and the subtour is still valid; otherwise, the subtour will be thrown away. The available branches, the graph, and the least upper bound are stored in the shared virtual memory. The program creates a process for each processor that performs the branch-and-bound algorithm on a branch obtained from the shared virtual memory. All the processes run in parallel until the tour is found. Such a program is easy

to write because the spawned processes do not require explicit data movement among processors.

### Matrix Multiply

The matrix multiply computes $C = AB$ where $A, B$ and $C$ are square matrices. A number of processes are created to partition the problem by the number of columns of matrix $B$. All the matrices are stored in the shared virtual memory. The program assumes that matrix $A$ and $B$ are on one processor at the beginning and they will be paged to other processors on demand. This program has a lot of computation and little data movement.

## 6.2 Speedups

The speedup of a program is the ratio of the execution time of the program on a single processor to that on the shared virtual memory system. The execution time of a program is the elapsed time from program start to program end, which is measured by the clock in the system. The execution time does not include the time of initializing data structures in the program because the initialization has little to do with the algorithm itself. For instance, the initialization of the merge-split sort program initializes an unsorted vector of records with random strings in their key fields. The time spent on the initialization depends on the generation of random strings; a complicated random string generating algorithm can well consume a lot of time. Indeed, if this initialization is included in the execution time of the program, and such an initialization is performed in parallel, it is possible to get a better speedup than the ideal speedup, since ideally this parallel algorithm does not yield a linear speedup.

In order to obtain a fair speedup measurement, all the programs in the experiments partition their problems by creating a certain number of processes according to the number of processors used. As a result of such a parameterized

partitioning, any program does its best for any given number of processors. To help observe the system performance of the prototype shared virtual memory system, all the speedup curves are compared against the ideal speedups on a single processor with the same parallel algorithm.

Figure 6.1 shows the speedup curve for the parallel Jacobi algorithm solving a linear equation $Ax = b$ where $A$ is a $26^3$ by $26^3$ matrix. The dashed line in the figure is the linear speedup curve. The parallel 3-D PDE solver has an almost identical result (Figure 6.2) in which matrix $A$ is $50^3$ by $50^3$. The dashed line in the figure is also the linear speedup curve. Note that both programs experience super-linear speedup.

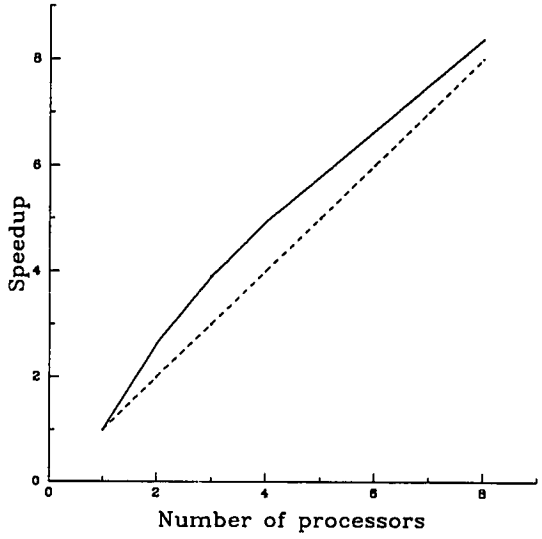


Figure 6.1: Speedups of solving a linear equation

At first glance, these results seem impossible because the fundamental law of parallel computation says that a parallel solution utilizing $p$ processors can improve the best sequential solution by at most a factor of $p$. Something must be interacting in either the programs or the shared virtual memory implementa-

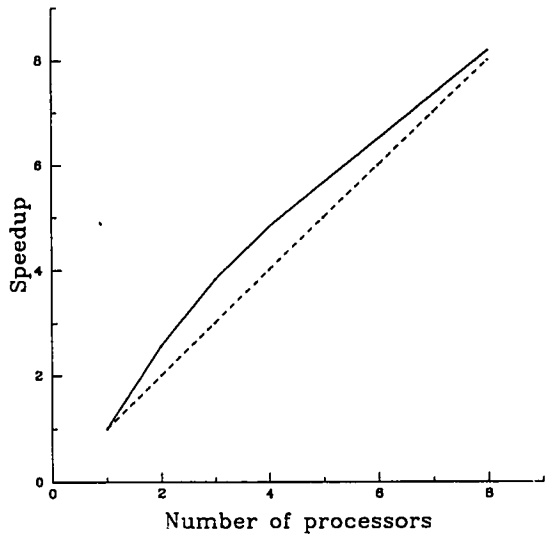


Figure 6.2: Speedups of a 3-D PDE where $n = 50$

tion. Since the algorithm in both programs is a straightforward transformation from the sequential Jacobi algorithm and all the processes are synchronized at each iteration, the algorithm cannot yield super-linear speedup. So, the speedup must be in the shared virtual memory implementation.

The shared virtual memory system can provide super-linear speedups because the fundamental law of parallel computation assumes that every processor has an infinitely large memory, which is not true in practice. For instance, in the parallel 3-D PDE example above, the data structure for the problem is greater than the size of physical memory on a single processor, so when the program is run on one processor there is a large amount of paging between the physical memory and disk.

Figure 6.3 shows the disk paging in the one processor case and the two processor case. The solid line in the figure shows the number of disk I/O pages when the program runs on one processor. The dashed line and dotted line shows

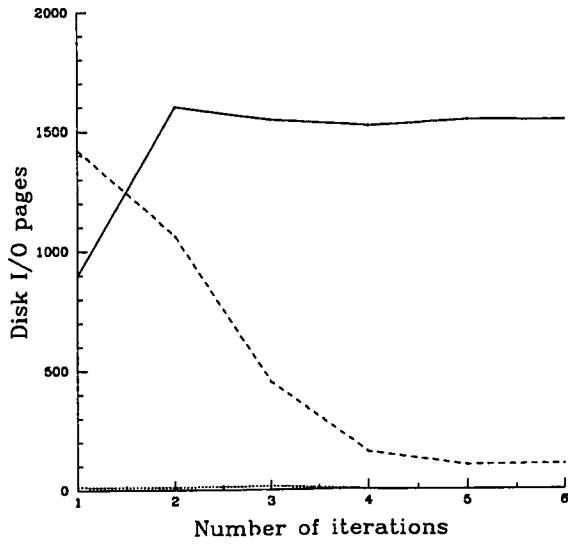


Figure 6.3: Disk paging on one processor and two processors

the numbers of disk I/O pages when the program runs on two processors. The two curves are so different because the program initializes its data structures only on one processor. The dashed line indicates the number of disk I/O pages on the processor with initialized data (processor 1) and the dotted line (which can hardly be seen) indicates that on the processor without initialized data (processor 2). Since the virtual memory paging in the Aegis operating system performs an approximated LRU strategy and the pages that moved to processor 2 are recently used on processor 1, processor 1 had to page out some pages that it needs later, causing more disk I/O page movement. This also explains why the number of disk I/O pages on processor decreases after the first few iterations.

The shared virtual memory, on the other hand, distributes the data structure into individual physical memories whose cumulative size is large enough to inhibit disk paging. It is clear from this example alone that the shared virtual

memory can indeed exploit the combined physical memories of a multiprocessor system.

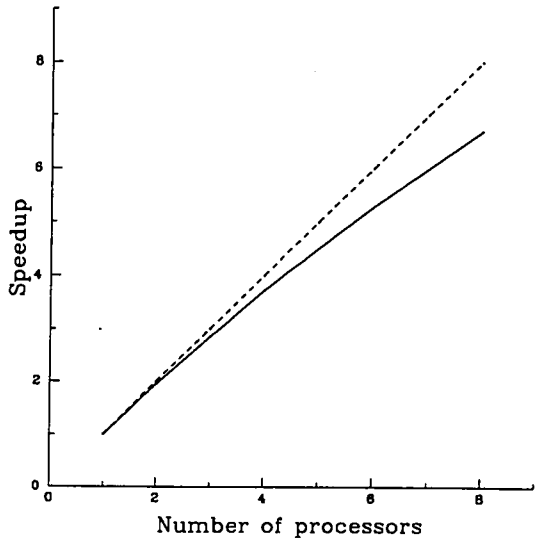


Figure 6.4: Speedups of a 3-D PDE where $n = 40$

Figure 6.4 shows another speedup curve for the 3-D PDE program, but now $n = 40$, in which case the data structure of the problem is not larger than the physical memory on a processor. This curve is what we see in most parallel computation papers. The curve is similar to that generated by similar experiments on CM*, a pioneer shared memory multiprocessor [Fuller 78, Jones 80, Deminet 82]. Indeed, the shared virtual memory system is as good as the best curve in the published experiments on CM* for the same program; but the efforts and costs of the two approaches are dramatically different. In fact, the best curve in CM* was obtained by keeping the private program code and stack in the local memory on each processor. The main reason that the performance of this program is so good in the shared virtual memory system is that the program exhibits a high degree of locality. While the shared virtual memory

system pays the cost of local memory references, CM* pays the cost of remote memory references aross its Kmaps.

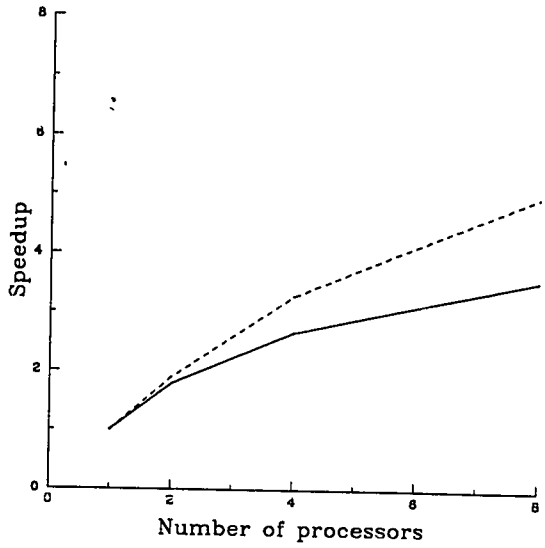


Figure 6.5: Speedup of merge-split sort

Parallel sorting on a loosely coupled multiprocessor is generally difficult. The speedup curve of the parallel merge-split sort of 200K elements shown in Figure 6.5 is not very good. In theory, even with no communication costs, this algorithm does not yield linear speedup. In order to give a fair comparison, the dashed line in the figure shows the speedup curve when the costs of all memory references are the same. Recall that the program uses the best strategy for any given number of processors. For example, there is one merge-split sorting when running the program on one processor, there are 4 blocks when running the program on two processors, and so on. Using a fixed number of blocks for any number of processors would result in a better speedup, but such an approach is not reasonable.

Figure 6.6 shows the speedup curve of the parallel dot-product program in

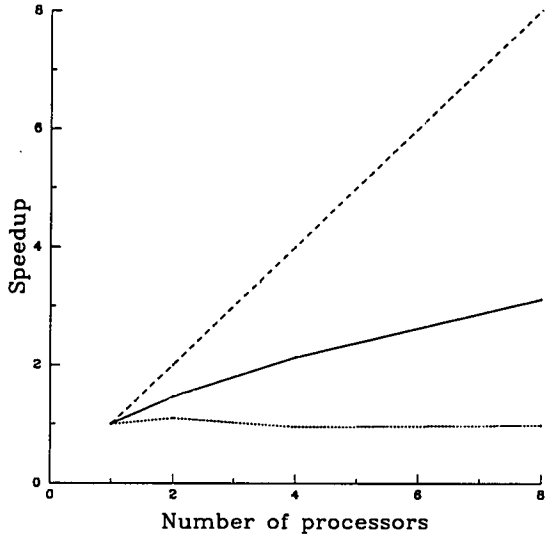


Figure 6.6: Speedup of dot-product

which each vector has 128K elements. It is included here so as not to paint too bright a picture. To be fair, the program assumes that the two vectors have a random distribution on each processor. Even with such an assumption, the speedup curve is not good, as indicated by the solid line in Figure 6.6. If the two vectors are located on one processor, there is no speedup at all, as indicated by the dotted curve in Figure 6.6, because the ratio of the communication cost to the computation cost in this program is large. For programs like dot-product, it is not appropriate to use a shared virtual memory system.

Figure 6.7 shows the speedup curve of the matrix multiplication program for $C = AB$ where both $A$ and $B$ are 128 by 128 square matrices. This example shows the good side of the shared virtual memory system. The speedup curve is close to linear because the program exhibits a high degree of localized computation.

The speedup curve of the travelling salesman problem is shown in Figure 6.8.

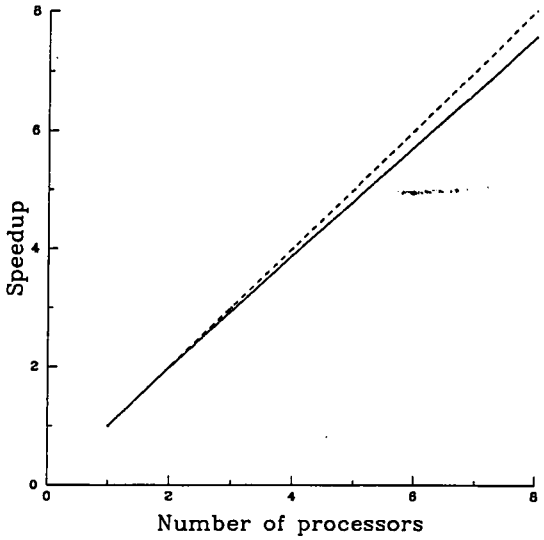


Figure 6.7: Speedup of matrix multiplication

Since the algorithm is a parallel branch-and-bound, there are anomalies [Lai 84]. It is possible that the program gets super-linear speedup or no speedup at all. In this example, it happens to have super-linear speedup. The reason it has super-linear speedup is that the program can get a smaller upper bound earlier than the sequential branch-and-bound program, so some branches are cut off in the parallel program while they are not cut off in the sequential one. Whether the program gets a better speedup also depends on the weights assigned to the edges. One can imagine an extreme case in which all the weights are similar. Such a case has all tours with similar costs. A sequential branch-and-bound should take exponential time to find the best tour whereas a parallel branch-and-bound should have a speedup close to linear. In this example, the weights are obtained by generating random numbers in the range $[10, 1000]$.

In general, the experimental results show that a shared virtual memory implementation is indeed practical, even on a very loosely coupled architecture

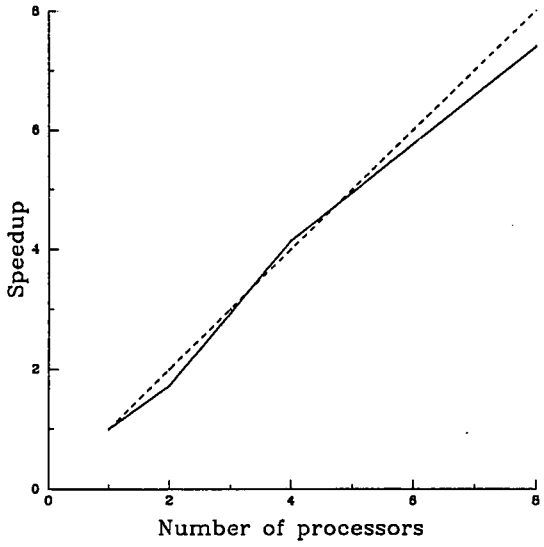


Figure 6.8: Speedup of traveling salesman problem

such as the Apollo ring.

## 6.3 Memory coherence algorithms

The most natural way to evaluate memory coherence algorithms is by comparing their system performance. In other words, algorithm $a_1$ is better than algorithm $a_2$ if the system based on $a_1$ can finish parallel programs faster than the one based on $a_2$.

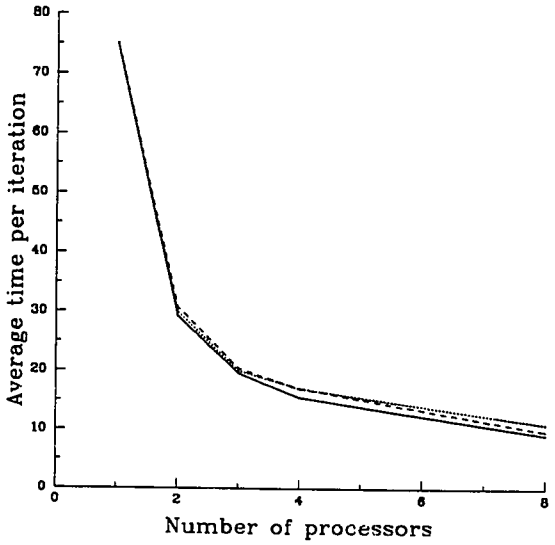


Figure 6.9: Performance with different algorithms

Figure 6.9 shows the average time per iteration of the 3-D PDE program running on up to eight processors using three different algorithms: the dynamic distributed manager algorithm (solid curve), the improved centralized manager algorithm (dotted curve), and the fixed distributed manager algorithm (dashed curve). The dynamic distributed manager algorithm performs better than the other two.

Although using system performance to compare memory coherence algorithms is a good approach, the differences between algorithms are not explicit

when a parallel program does not have a lot of page faults. An alternative approach is to measure the total number of messages used in an algorithm. For example, the number of forwarding requests can be used as a criteria for comparing algorithms. In order to do so, each processor at run time records the statistical information into a file. The information includes the number of read page faults, the number of write page faults, process migration information, memory page distribution information, and simple RPC information. The information about disk paging and network traffic is obtained from the Aegis operating system.

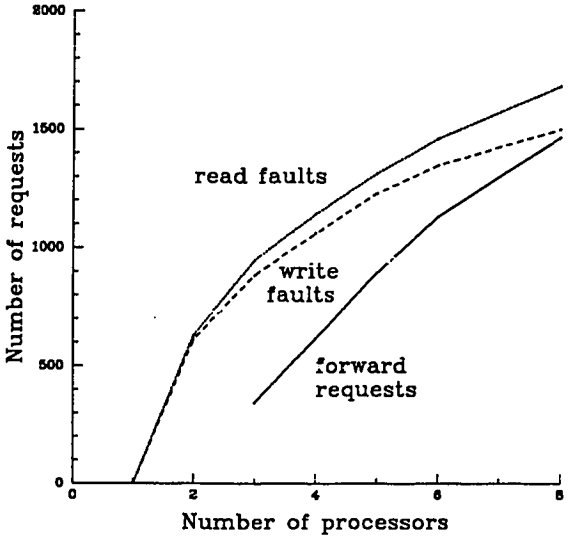


Figure 6.10: Improved centralized manager algorithm

Figures 6.10, 6.11 and 6.12 show the number of forwarding requests for locating true pages during the first six iterations of the 3D PDE program using the improved centralized manager algorithm, the fixed distributed manager algorithm, and the dynamic distributed manager algorithm, respectively.

In the fixed distributed manager algorithm, the manager mapping function

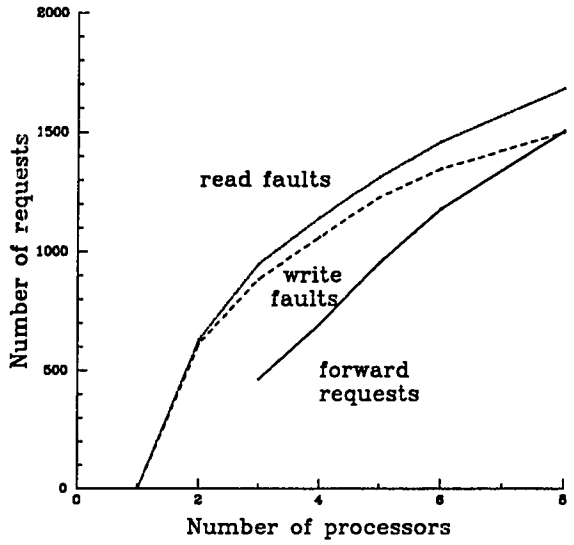


Figure 6.11: Fixed distributed manager algorithm

is

$$H(p) = p \bmod N$$

where $p$ is a page number and $N$ is the number of processors. The curve of the forwarding requests of the fixed distributed manager algorithm is similar to that of the improved centralized manager algorithm because both algorithms need a forwarding request to locate the owner of the page for almost every page fault that occured on a non-manager processor. Since the workload of the fixed distributed manager algorithm is a little better than that of the improved centralized manager algorithm, the performance of the former is a little better than the latter as the number of processors increases.

The figures show that the overhead of the dynamic distributed manager algorithm is much less than that of the other two algorithms. This confirms the theoretical result that the dynamic distributed manager algorithm outperforms the other two because the *prob_owner* fields usually give correct hints (thus the

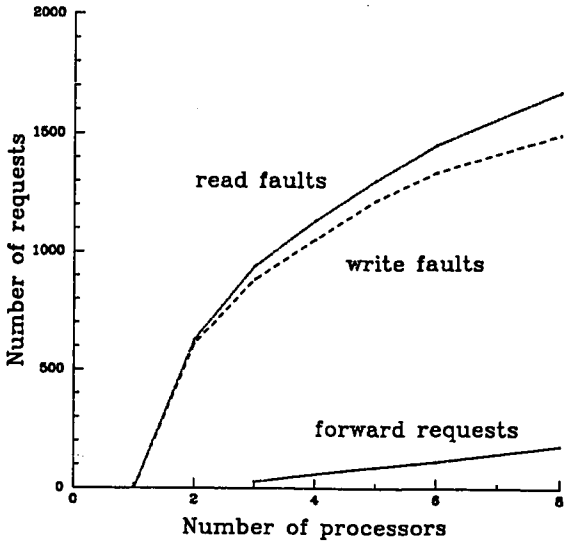


Figure 6.12: Dynamic distributed manager algorithm

number of forward requests is very small), and within a short period of time the number of processors sharing a page is small.

## 6.4 Miss Ratios

The memory reference miss ratio of a program execution in a shared virtual memory system is the ratio of the number of the shared virtual memory references that cause page faults to the number of the shared virtual memory references that do not cause page faults. Normally, miss ratios are measured by trace driven simulations [Smith 82, Smith 85]. For a shared virtual memory system, memory reference traces are difficult to get because recording memory reference traces will change the timing of the parallel execution of a program which results in changing the miss ratio itself. The miss ratios obtained here are calculated by counting the number of shared virtual memory references and

the total page faults for each execution of a program. Counting the number of shared virtual memory references is done by reading the assembly code of each program. Note that disk page faults are not included because they are architecture-dependent.

The miss ratio of a program not only reflects the system performance of a shared virtual memory system, but also indicates the granularity of parallelism in the program. In order to compare the localities of different benchmark programs, the miss ratios of the 3D PDE program, the sort program, and the matrix multiply program are put into one figure in which the miss ratio dimension is from 0 to 0.001. If the miss ratio of a program is 0.001, it means that, on average, the locality of the program can at least guarantee that there are as many as 1000 memory references per page fault, or that each word in the page is referenced 4 times.

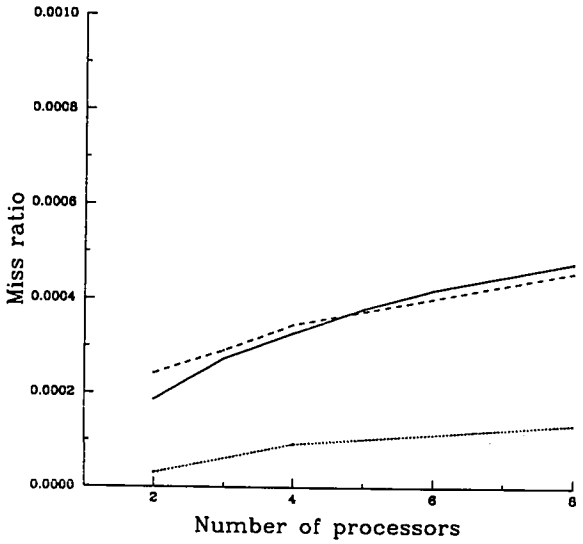


Figure 6.13: Miss ratios

In Figure 6.13, the solid line shows the miss ratio of the first six iterations

of the 3D PDE program in which the matrix is $50^3$ by $50^3$. The miss ratio curve is a function of the number of processors. The miss ratio is obtained by counting the number of memory references and the number of page faults during each iteration. The miss ratio curve goes up as the number of processors increases. The low miss ratio indicates that the program has fairly coarse grain parallelism and has *locality* in the shared virtual memory environment.

The dashed line in Figure 6.13 shows the miss ratio of the merge-split sorting program. At the internal sorting stage, the program obviously has a high degree of locality and at the merge-split stage, one of the two merging blocks is always local.

The dotted line in Figure 6.13 shows the miss ratio of the matrix multiply program. The miss ratio is very low because two matrices are read only and they are only paged in at the beginning of the execution. Apparently, the program has a high degree of locality.

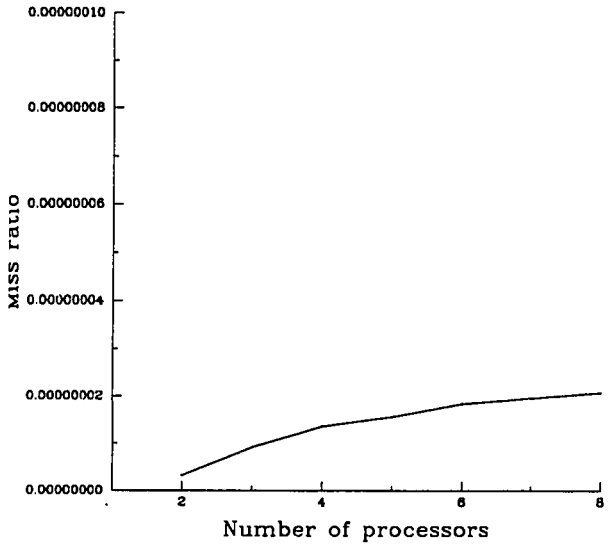


Figure 6.14: Miss ratio of a linear equation solver

Figure 6.14 shows the average miss ratio of the first eight iterations of the linear equation solver in which the matrix $A$ is $26^3$ by $26^3$. Since it is stored in the shared virtual memory (unlike the 3D PDE program in which the matrix is coded in program), there are more shared virtual memory references than those in the 3D PDE program. The miss ratio is so low that the figure has to use the limits that are 1/10000 of the one in Figure 6.13.

The miss ratio of the dot-product program is not interesting because the two vectors in the program are only referenced once. The miss ratio of the traveling salesman problem is difficult to obtain because the number of shared virtual memory references is not fixed.

## 6.5 Remarks

The experimental results shown in this chapter strongly support the idea of the shared virtual memory on loosely coupled multiprocessors. Although the experiments were run on a not well-tuned system implemented on a very loosely coupled multiprocessor, it is possible to get super-linear speedups without even paying special hardware cost.

Given the time and the implementation environment, the next experiments I would like to run would measure the performance on more than eight processors. Although building a shared virtual memory system on more than eight processors is perhaps not very important for a very loosely coupled multiprocessor system like a network of workstations, the experimental results would show the limits of a shared virtual memory system.

Next, I would like to run experiments comparing the different scheduling strategies described in Chapter 3 and the page replacement algorithms described in Chapter 4. Furthermore, I would like to run experiments to compare different page sizes in a shared virtual memory system. Although these experiments are not crucial to my thesis, they will fully justify other ideas proposed in the dissertation.

# Chapter 7

# Final Thoughts

I have presented strong evidence in support of my thesis that shared virtual memory on a loosely coupled multiprocessors can achieve orders-of-magnitude speedups over a uniprocessor for many parallel programs, and that it is practical to implement on existing architectures. But this dissertation only begins the research in building the shared virtual memory. Many research problems remain. This chapter addresses some of them.

## 7.1 Generality

The prototype shared virtual memory has demonstrated that the idea of shared virtual memory is practical for loosely coupled multiprocessors. A natural question is whether the concept can be applied to tightly coupled multiprocessors or a network of tightly coupled multiprocessors.

For a tightly coupled multiprocessor based on an interconnection network, a shared virtual memory implementation may still prove both feasible and desirable. For example, in tightly coupled multiprocessors such as CM*, Butterfly, and RP3 [Jones 80, Larus 84, Pfister 85], the cost of a remote memory reference is more expensive than a local memory reference although any processor can reference any memory location. A straightforward implementation of virtual

memory on a multiprocessor of this kind would result in losing the localities of parallel programs that the shared virtual memory system can offer. This is exactly why the experiment of the 3D PDE program on IVY has a similar curve to that of the best curve in the experiment on CM*.

To implement a shared virtual memory on a tightly coupled multiprocessor, one may want to make the page size very small and have special hardware do paging to solve the memory coherence problem. So far, for $N$ memory units, all the practical interconnection networks have a $\log N$ delay for a remote memory reference. This delay is a big concern if one wants to build a multiprocessor with a massive number of processors [Snyder 86]. Since the shared virtual memory system offers localities for many parallel programs, the average delay of a memory reference may be reduced by a large factor.

The techniques of the shared virtual memory system proposed in this thesis can probably apply directly to a network of tightly coupled multiprocessors. For example, a multiprocessor workstation such as a Firefly provides a true shared memory and solves the memory coherence problem through the use of snoopy caches [Thacker 86]. When Fireflies are connected by a communication link, it would be possible to use the design methods proposed in this thesis to build a shared virtual memory among a number of Fireflies. Such a shared virtual memory implementation would allow a parallel program to run on more than one multiprocessor while there is still a shared memory space. Process scheduling would play an important role in the system because a program can run faster if the processes that are closely related to each other run on the same multiprocessor workstation.

The idea of building a shared virtual memory on a network of multiprocessors suggests a radically different viewpoint of parallel architectures in which one can build a shared memory based massively parallel architecture by using loosely coupled links and software instead of using high cost interconnection networks.

## 7.2 Parallel Programming Language

In the prototype shared virtual memory system IVY, there is no parallel programming language implementation. For this reason, the programmer needs to manually write parallel constructs. She must allocate memory for shared data structures and create processes by using the primitives provided by the system. A parallel programming language would make the programming environment much more convenient.

In order to see whether a shared virtual memory system can be used as the base from which one implements parallel programming languages, let us briefly examine the implementation of several recently proposed parallel programming languages.

Linda is a parallel programming language in which a small set of primitives are used to access a global tuple space [Gelernter 85]. Implementing the global tuple space on top of a shared virtual memory system would greatly simplify the implementation designs. Indeed, it would be sufficient to implement only atomic local primitives since the shared virtual memory solves the memory coherence problem. If tuples are carefully allocated in the global tuple space, even the first use of a read operation may not require any data movement because other operations used before may have paged in its data. I would expect that the system performance of such an implementation would be better than a straightforward implementation.

Multilisp is an extension of Lisp (more specifically, the Lisp dialect Scheme) with additional operators and additional semantics to deal with parallel execution [Halstead Jr 85]. The shared virtual memory perfectly fits the langauge design because Multilisp assumes a shared memory and all the processes created by *pcall* and *future* share the same address space. Multilisp has been implemented on Concert [Anderson 82, Halstead Jr 84], a multiprocessor that has a physical shared memory. If one wants to implement a Multilisp on a loosely coupled multiprocessor, the shared virtual memory would present an environ-

ment just as convenient as any tightly coupled multiprocessors with physical shared memories. Whether it is possible to gain enough parallelism for most parallel programs is an open question.

ParAlfl is a parallel functional language augmented with features that allow programs to be mapped to specific multiprocessor topologies [Hudak 85]. Again, ParAlfl assumes a shared memory among processors. The shared virtual memory fits this langauge extremely well because programs in this langauge do not have side-effects which are the causes of memory contention. I would expect that such a language would show the best side of the shared virtual memory system. Furthermore, implementing a mechanism to map functions automatically onto multiprocessors would be much simpler than using any existing message passing means because the shared virtual memory provides the compiler with a single address space so that passing data structures is no longer necessary.

From all the implementation issues I can see, the shared virtual memory seems to fit parallel programming languages well in both performance and the effort needed to implement them.

## 7.3 Granularity

The page size in the prototype shared virtual memory implementation IVY is 1024 bytes. Experiments have shown that such a granularity is suitable for a large class of parallel programs. The informal analysis in Chapter 2 indicates that the page size in the shared virtual memory system should be as small as the MMU allows in existing systems. If one designs an architecture, one needs to decide, given a communication architecture and a transmission rate, what the optimal page size is for a shared virtual memory system.

This problem must be addressed in both theory and practice. So far, no one has practical experience with different page sizes. Since the shared virtual memory system is on multiprocessors, it might be difficult to simulate. A plausible way to collect empirical data is to use a system whose page size is as

small as or smaller than 256 bytes. With some work, the shared virtual memory implementation could parameterize its page size to be an integral multiple of the original page size. This kind of implementation could serve the experiments with different page sizes.

Recent studies of page sizes for diskless workstations show that moderate page sizes (8-16K bytes) are better than small ones [Lazoweska 84]. The best page size for disk paging may not match the best page size for shared virtual memory. In order to optimize both, architecture designers may need to consider using different page sizes for disk paging and shared virtual memory.

## 7.4 Reliability

Throughout the thesis, I have not addressed the reliability issue. Although the simple RPC protocol proposed in Chapter 4 allows processors to lose packets, it does not allow the shared virtual memory system to have faulty or fail-stop (or crash only) processors during its execution. This restriction is not a big problem for reproducible applications such as scientific computing problems because one can always recreate a fresh shared virtual memory system and redo the computation. It would be a problem for applications that are not reproducible.

The faulty processor problem is a form of the Byzantine Generals Problem [Pease 80]. It is not clear whether it is possible to apply the theoretical results of the problem to the shared virtual memory system. The fail-stop recovery problem is simpler but it requires a lot of work.

## 7.5 Other Applications

I have only tested a small set of parallel programs, which raises the question of whether the shared virtual memory system is good for other applications or not. The parallel programs in the benchmark set have a fairly fine granularity of par-

allelism. They all have side-effects to their shared data structures at run time. Parallel application programs with similar parallel granularities and similar ratios of the cost of communication to the cost of computation should perform as well as the benchmark programs if they run on a shared virtual memory system with the system performance similar to or better than that in the prototype system. Clearly, the parallel applications that have fewer side-effects to their shared data structures and have coarser granularity of parallelism would perform better.

If the fail-stop recovery problem can be solved, some distributed computing applications might be well suited to the shared virtual memory system. An interesting example is a distributed database system. One can imagine that many things in implementing such a system would be greatly simplified because the data consistency problem would be well taken care of by the shared virtual memory system. Furthermore, some expensive database operations such as *Join* could be parallelized because the shared virtual memory system supports parallel computation.

# Bibliography


[Aho 74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman.
*The Design and Analysis of Computer Algorithms.*
Addison-Wesley Publishing Company, 1974.

[Anderson 82] Thomas L. Anderson.
*The Design of A Multiprocessor Development System.*
Technical Report MIT/LCS/TR-279, Massachusetts Institute of Technology, September 1982.

[Annaratone 86] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M.S. Lam, O. Menzilcioglu, K. Sarocky, and J.A. Webb.
Warp Architecture and Implementation.
In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 346–356, 1986.

[Apollo 81] Apollo.
*Apollo DOMAIN Architecture.*
Apollo Computer Inc., Chelmsford, Mass., 1981.

[Archibald 85] J. Archibald and J. Baer.
*An Evaluation of Cache Coherence Solutions in Shared-bus Multiprocessors.*
Technical Report 85-10-05, University of Washington, October 1985.

[Baker 78] Henry G. Baker.
*Actor Systems for Real-time Computation.*
PhD thesis, M.I.T., March 1978.
Technical Report TR-197.

[Barnes 68] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes.
The ILLIAC IV Computer.
*IEEE Transactions on Computers*, C-17:746–757, August 1968.

[Belady 66] L.A. Belady.
A Study of Replacement Algorithms for Virtual Storage Computers.
*IBM Systems Journal*, 5(2):78–101, 1966.

[Birrell 83] A.D. Birrell and B.J. Nelson.
*Implementing Remote Procedure Calls.*
Technical Report CSL-83-7, Xerox PARC, December 1983.

[Birrell 84] Andrew D. Birrell.
Private communications, 1984.

[Bitton 84] D. Bitton, D.J. DeWitt, D.K. Hsaio, and J. Menon.
A Taxonomy of Parallel Sorting.
*ACM Computing Surveys*, 16(3):287–318, September 1984.

[Bobrow 72] D.G. Bobrow, J.D. Burchfiel, D.L. Murphy, and R.S. Tomlinson.
TENEX, a paged time-shaing system for the PDP-10.
*Communications of the ACM*, 15(3):135–143, March 1972.

[Bokhari 79] S.H. Bokhari.
Dual Processor Scheduling with Dynamic Reassignment.
*IEEE Transactions on Software Engineering*, SE-5(4), October 1979.

[Carr 81] R.W. Carr and J.L. Hennessy.
WSClock — A Simple and Efficient Algorithm for Virtual Memory Management.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981.

[Carriero 86a] N. Carriero and D. Gelernter.
The S/Net's Linda Kernel.

*ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[Carriero 86b] Nicholas Carriero.
Private communications, 1986.

[Censier 78] L.M. Censier and P. Feautrier.
A New Solution to Coherence Problems in Multicache Systems.
*IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[Cheriton 84] David R. Cheriton.
The V kernel: A Software Base for Distributed Systems.
*IEEE Software*, 1(2):19–43, 1984.

[Cheriton 86] D.R. Cheriton and M. Stumm.
The Multi-Satellite Star: Structuring Parallel Computations for A Workstation Cluster.
Presentation at Yale by M. Stumm, 1986.

[Coffman Jr. 72] E.G. Coffman Jr. and R.L. Grahan.
Optimal Scheduling For Two-processor systems.
*Acta Informatica*, 1(3):200–213, 1972.

[Coffman, Jr. 73] E.G. Coffman, Jr. and P.J. Denning.
*Operating Systems Theory.*
Prentice-hall, Inc., Englewood Cliffs, New Jersey, 1973.

[Conway 63] M.E. Conway.
A Multiprocessor System Design.
In *Proceedings of Fall Joint Computer Conference*, pages 139–146, AFIPS Press, 1963.

[Corbato 62] F.J. Corbato and et al.
An Experimental Time Sharing System.
In *Proceedings of Spring Joint Computer Conference*, pages 335–344, AFIPS Press, 1962.

[Daley 68] R.C. Daley and J.B. Dennis.
Virtual Memory, Processes, and Sharing in MULTICS.
*Communications of the ACM*, 11(5):306–312, May 1968.

[Deminet 82] Jarek Deminet.
Experience with Multiprocessor Algorithms.
*IEEE Transactions on Computers*, C-31(4), April 1982.

[Denning 68] Peter J. Denning.
The Working Set Model for Program Behavior.
*Communications of the ACM*, 11(5):323–333, May 1968.

[Denning 70] Peter J. Denning.
Virtual Memory.
*ACM Computing Surveys*, 2(3):153–189, September 1970.

[Denning 72] Peter J. Denning.
On Modeling Program Behavior.
In *Proceedings of Spring Joint Computer Conference*, pages 937–944, AFIPS Press, 1972.

[Denning 80] Peter J. Denning.
Working Sets Past and Present.
*IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[Dennis 66] J.B. Dennis and E.C. Van Horn.
Programming Semantics for Multiprogrammed Computations.
*Communications of the ACM*, 9(3):143–155, March 1966.

[Dijkstra 68] E.W. Dijkstra.
Cooperating Sequential Processes.
In F. Genuys, editor, *Programming Languages*, Academic Press, New York, 1968.

[Easton 76] M.C. Easton and P.A. Franaszek.
*Use Bit Scanning in Replacement Decisions.*
Technical Report RC-6192, IMB Research, Yorktown Heights, September 1976.

[Ellis 85] John R. Ellis.
Private communication, 1985.

[Emrath 85] Perry Emrath.
Xylem: An Operating System for the Cedar Multiprocessor.
*Software*, :30–37, July 1985.

[Finkel 80] R. Finkel.
*The Arachne Kernel.*
Technical Report TR-380, University of Wisconsin, April 1980.

[Finkel 85] R. Finkel and U. Manber.
DIB – A Distributed Implementation of Backtracking.
In *The Fifth International Conference on Distributed Computing Systems*, May 1985.

[Fitzgerald 86] R. Fitzgerald and R.F. Rashid.
The Integration of Virtual Memory Management and Interprocess communication in Accent.
*ACM Transactions on Computer Systems*, (2):147–177, May 1986.

[Flynn 66] M.J. Flynn.
Very High Speed Computing Systems.
In *Proceedings of IEEE*, pages 1901–1909, December 1966.

[Fowler 86] Robert J. Fowler.
*Decentralized Object Finding Using Forwarding Addresses.*
PhD thesis, University of Washington, 1986.

[Frank 84] Steven J. Frank.
Tightly Coupled Multiprocessor System Speeds Memory-access Times.
*Electronics*, :164–169, January 1984.

[Friedman 76] D. Friedman and D. Wise.
CONS Should not Evaluate its Arguments.
In S. Michaelson and R. Milner, editors, *Automata, Languages, and Programming*, pages 257–284, Edinburgh University Press, 1976.

[Fuller 78] S. Fuller, J. Ousterhout, L. Raskin, P. Rubinfeld, P. Sindhu, and R. Swan.
Multi-microprocessors: an overview and working example.
*Proceeding of the IEEE*, (2):216–228, February 1978.

[Gelernter 85] David Gelernter.
Generative Communication in Linda.
*ACM Transactions on Programming Languages and Systems*, (1):80–112, January 1985.

[Gonzalez 78] T. Gonzalez and S. Sahni.
Preemptive Scheduling of Uniform Processor Systems.
*Journal of the ACM*, 25(1):92–101, January 1978.

[Goodman 83] James R. Goodman.
Using Cache Memory to Reduce Processor-memory Traffic.
In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, June 1983.

[Habermann 76] A.N. Habermann.
*Introduction to Operating System Design.*
Science Research Associates, 1976.

[Halstead Jr 84] Robert H. Halstead Jr.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *ACM Symposium on Lisp and Functional Programming*, pages 9–17, 1984.

[Halstead Jr 85] Robert H. Halstead Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
*ACM Transactions on Programming Languages and Systems*, October 1985.

[Heid 70] M. Heid and R.M. Karp.
The Traveling-salesman Problem and Minimum Spanning Trees.
*Operation Research*, 17(12):1139–1167, December 1970.

[Henderson 76] P. Henderson and J.H. Morris.
A Lazy Evaluator.

In *Conference Record of the third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[Herlihy 82] M. Herlihy and B. Liskov.
A Value Transmission Method for Abstract Data Types.
*ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hillis 85] Daniel M. Hillis.
*The Connection Machine.*
MIT Press, 1985.

[Hoare 62] C.A.R. Hoare.
Quicksort.
*Computer Journal*, 5(1):10–15, 1962.

[Hoare 74] C.A.R. Hoare.
Monitors: An Operating System Structuring Concept.
*Communications of the ACM*, 17(10):549–557, October 1974.

[Hudak 85] P. Hudak and B. Goldberg.
Distributed Execution of Functional Programs Using Serial Combinators.
*IEEE Transactions on Computers*, C-34(10):881–891, October 1985.

[Hudak 86a] P. Hudak and L. Smith.
Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems.
In *Conference Record of the Twelvth Annual ACM Symposium on Principles of Programming Languages*, pages 243–254, January 1986.

[Hudak 86b] Paul Hudak.
Private communications, 1986.

[Jones 79] A.K. Jones, R.J. Chansler, I.E. Durham, K. Schwans, and S. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.

In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 117–127, 1979.

[Jones 80] A. K. Jones and P. Schwarz.
Experience Using Multiprocessor Systems — A Status Report.
*ACM Computing Surveys*, 12(2), June 1980.

[Karp 72] R.M. Karp.
Reducibility Among Combinatiorial Problems.
In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, Plenum Press, 1972.

[Katz 85] R. H. Katz, S. J. Eggers, D.A. Wood, C. L. Perkins, and R.G. Sheldon.
Implementing A Cache Consistency Protocol.
In *Proceedings of the 12th Annual Symposium on Computer Architecture*, pages 276–283, June 1985.

[Knuth 73] Donald E. Knuth.
*The Art of Computer Programming, Volume III.*
Addison-Wesley Publishing Company, 1973.

[Kronenberg 86] N.P. Kronenberg, H.M. Levy, and W.D. Strecker.
VAXclusters: A Closely-coupled Distributed System.
*ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[Lai 84] T. Lai and S. Sahni.
Anomalies in Parallel Branch-and-Bound Algorithms.
*Communications of the ACM*, 27(6):594–602, June 1984.

[Lampson 68] Butler W. Lampson.
A Scheduling Philosophy for Multiprocessing Systetms.
*Communications of the ACM*, 11(5):347–360, May 1968.

[Larus 84] James R. Larus.
Using the Baskett Test A Benchmark for the Butterfly Multiprocessor.
February 1984.
Draft.

[Lazoweska 84] E.D. Lazoweska, J. Zahorjan, D.R. Cheriton, and W. Zwaenepoel.
*File Access Performance of Diskless Workstations.*
Tech Report STAN-CS-84-1010, Stanford University, 1984.

[Leach 82] P.J. Leach, B.L. Stumpf, J.A. Hamilton, and P.H. Levine.
UIDs as Internal Names in A Distributed File System.
In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 34–41, 1982.

[Leach 83] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf.
The Architecture of an Integrated Local Network.
*IEEE Journal on Selected Areas in Communications*, 1983.

[Levin 86] Roy Levin.
Private communications, 1986.

[Levy 82] H.M. Levy and P.H. Lipman.
Virtual Memory Management in the VAX/VMS Operating System.
*IEEE Computer*, :35–41, March 1982.

[Li 86] Kai Li.
A New List Compaction Method.
*Software Practice and Experience*, 16(2):145–163, February 1986.

[McCreight 84] E. McCreight.
The Dragon Computer system, An Early Overview, 1984.
Draft document.

[Metcalfc 76] R.M. Metcalfc and D.R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
*Communications of the ACM*, 19(7), July 1976.

[Mitchell 79] J.G. Mitchell, W. Maybury, and R. Sweet.
*Mesa Language manual, version 5.0.*
Tech Report CSL-79-3, Xerox Parc, 1979.

[Motorola 84] Motorola.
*M68000 16/32-bit Microprocessor.*
Prentice-Hall, Inc., fourth edition edition, 1984.
Programmer's Reference Manual.

[Muntz 70] R.R. Muntz and E.G. Coffman Jr.
Preemptive Scheduling of Real-time Tasks on Multiprocessor Systems.
*Journal of the ACM*, 17(2):324–338, April 1970.

[Nelson 81] Bruce J. Nelson.
*Remote Procedure Call.*
PhD thesis, Carnegie-Mellon University, May 1981.

[Ousterhout 80] J.K. Ousterhout, D.A. Scelza, and P.S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
*Communications of the ACM*, 23(2):92–105, February 1980.

[Pease 80] S.M. Pease and L. Lamport.
Reaching Agreement in the Presence of Faults.
*Journal of the ACM*, 27(2):228–234, April 1980.

[Pfister 85] G.F. Pfister, W.C. Brantley, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss.
The IBM Research Parallel Processor Prototype (RP3).
In *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[Powell 83] Michael L. Powell.
Process Migration in DEMOS/MP.
In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 110–119, 1983.

[Rashid 81] R.F. Rashid and G.G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[Reed 79] David P. Reed and Rajendra K. Kanodia.
Synchronization with Eventcounts and Sequencers.
*Communications of the ACM*, 22(2):115–123, February 1979.

[Ritchie 78] D.M. Ritchie and K. Thompson.
The UNIX Time-sharing System.
*Bell System Technical Journal*, 57:1905–1929, July 1978.

[Robinson 79] J.T. Robinson.
Some Analysis Techniques for Asynchronous Multiprocessor Algorithms.
*IEEE Transactions on Software Engineering*, SE-5(1), January 1979.

[Seitz 85] Charles L. Seitz.
The Cosmic Cube.
*Communications of the ACM*, 28(1):22–33, January 1985.

[Smith 82] Alan J. Smith.
Cache Memories.
*ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smith 85] Alan J. Smith.
Disk Cache—Miss Ratio Analysis and Design Considerations.
*ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[Snyder 82] Lawrence Snyder.
Introduction to the Configurable, Highly Parallel Computer.
*Computer*, 15(1):47–56, 1982.

[Snyder 86] Lawrence Snyder.
Type Architectures, Shared Memory and the Corollary of Modest Potential.
In *Annual Review of Computer Science*, 1986.
To appear.

[Spector 81] Alfred Z. Spector.
*Multiprocessing Architectures for Local Computer Networks.*

Ph.D. thesis STAN-CS-81-874, Stanford University, August 1981.

[Spector 82] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
*Communications of the ACM*, 25(4):260–273, April 1982.

[Stone 77] H.S. Stone.
Multiprocessor Scheduling with the Aid of Network Flow Algorithms.
*IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[Tang 76] C.K. Tang.
Cache System Design in The Tightly Coupled Multiprocessor System.
In *Proceedings of AFIP National Computing Conference*, pages 749–753, 1976.

[Tarjan 84] R.E. Tarjan and J. Van Leeuwen.
Worst-case Analysis of Set Union Algorithms.
*Journal of the ACM*, 31(2):245–281, April 1984.

[Thacker 84] Chuck Thacker and et al.
Private communications, 1984.

[Thacker 86] Chuck Thacker and et al.
Private communications, 1986.

[Tuomenoksa 85] D.L. Tuomenoksa and H.J. Siegel.
Task Scheduling on the PASM Parallel Processing System.
*IEEE Transactions on Software Engineering*, SE-11(2):145–157, February 1985.

[Wilkes 79] M.V. Wilkes and D.J. Wheeler.
The Cambridge Digital Communication Ring.
In *Proceedings of the Local Area Communications Network Symposium*, May 1979.

[Yen 85] W. C. Yen, D. W. L. Yen, and K. Fu.
Data Coherence Problem in a Multicache System.
*IEEE Transactions on Computers*, C-34(1):56–65, January 1985.

[Zwaenepoel 85] Willy Zwaenepoel.
*Protocols for Large Data Transfers over Local Networks.*
Tech Report TR85–23, Rice University, July 1985.