Shared Memory for Distributed Systems

A thesis
submitted in partial fulfillment
of the requirements for the degree
of

Bachelor of Technology
in
Computer Science and Engineering

by

Mohit Aron **Arijit Sarcar**

Under the guidance of
Prof. B. B. Madan

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

May 1995

Certificate

This is to certify that the thesis titled *Shared Memory for Distributed Systems* being submitted by *Mohit Aron* and *Arijit Sarcar* to the Indian Institute of Technology, Delhi, for award of the **Bachelor of Technology** in **Computer Science and Engineering** is a record of bonafide work carried out by them under my supervision.

The results presented in this thesis have not been submitted elsewhere, either in part or in full, for the award of any other degree or diploma.

Prof. B. B. Madan
Professor
Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi 110016

To Our Parents

The first men to be created and formed were called the Sorcerer of Fatal Laughter, the Sorcerer of Night, Unkempt, and the Black Sorcerer ... They were endowed with intelligence, they succeeded in knowing all that there is in the world. When they looked, instantly they saw all that is around them, and they contemplated in turn the arc of heaven and the round face of the earth ... [Then the Creator said]: "They know all ... what shall we do with them now? Let their sight reach only to that which is near; let them see only a little face of the earth! ... Are they not by nature simple creatures of our making? Must they also be Gods?"
- The Popol Vuh of the Quiché Maya

The known is finite, the unknown infinite; intellectually we stand on an islet in the midst of an unlimitable ocean of inexplicability. Our business in every generation is to reclaim a little more land.
-T.H. Huxley, 1887

Acknowledgments

We would like to convey our deeply felt gratitude towards Prof. B. B. Madan , for providing us the opportunity to work on a project which is of fundamental importance to the research world of Computer Science. He was a constant source of inspiration and encouragement, and his superlative guidance and impeccable technical advice was of immense help in the culmination of this work. He showed us the light on every dark corner, and we duly appreciate his patience and motivating drive.

We also extend our thanks to Mr. Atul Varshneya . His pertinent insights were of tremendous help in fine-tuning our efforts.

This work owes much to the cooperation extended to us by our friend Neeraj Mittal in helping us with lex and yacc in the project.

Last but not the least, we appreciate the cooperation we received from Mr. Jaswant and Mr. Prasad regarding lab activities.

Mohit Aron

Arijit Sarcar

Table of Contents

List of Figures

Chapter 1

Introduction

A single address space enhances the programmability of a parallel machine by reducing the the problems of data partitioning and consequent dynamic load distribution, two of the toughest problems in programming parallel machines. A shared address space also improves support for automatically parallelizing compilers, standard operating systems, multiprogramming and incremental tuning of parallel applications: features that make a single address space much easier to use than a message passing machine. Thus a need to develops a **scalable** shared memory architecture has arose which led to the birth of directory-based cache coherence, first proposed in the late 1970s. However, the original directory structures were not scalable because they used a centralized directory that quickly became a bottleneck. The *DASH RESEARCH GROUP* of Stanford University have overcome this limitation by partitioning and distributing the directory and main memory, and by using a new coherence protocol, *distributed directory based coherence protocol*, that can suitably exploit distributed directories.

1.1 Motivation

The major problem with existing cache-coherent shared-address machines is that they have not demonstrated the ability to scale effectively beyond a few high-performance processors. To date, only message passing machines have shown this ability. The research group of the Stanford's DASH project are of the opinion that **Distributed** Directory based coherence mechanism will permit single-address-space machines to scale as well as message-passing machines, while providing a more flexible and general programming model.

Before developing a hardware platform to implement the new protocol, it was thought it would be far more prudent to develop a detailed software simulator for the system.

1.2 Objectives

- To provide a shared address space on a cluster of workstations interconnected together on a LAN.
- Use the distributed directory for maintaining pointers onto cache locations which cache memory locations.
- Use a new set of coherence protocols which closely imitates those designed by the DASH research group.

Thus the objective is to design a software simulator which in turn will provide a set of specifications required for implementing the DASH architecture in hardware.

1.3 Implementation Overview

Any instruction which accesses the memory is directly catered to by the MMU (Memory Management Unit) through the kernel. Since the kernel was not to be tinkered around with, the routines providing the shared memory were provided at a higher layer and the user program can access the memory through functions which can provide the following services:

- Read or Write a word i. e. four bytes.
- Read or Write a halfword i. e. two bytes.
- Read or Write a byte.

Apart from partitioning the total shared memory the user wants equally among the number of processors it also maintains consistency, by using the local LAN to implement its coherence protocols through *RPC* calls, among the various cached locations of the shared memory.

Thus by using this software any user can run one's process on any processor and get the following services:

1. One needn't worry about which processor actually holds the memory location he/she is accessing.

2. One also gets the simulated access-time it took to satisfy his/her request, thus giving an idea of the actual time the access would have taken, had the shared memory been provided by the hardware.

3. The user is also provided with a **TSL** synchronization primitive for dealing with critical regions when writing parallel applications.

Finally a Graphical User Interface to the software is presented for the convenience of users. Two application are built which runs on this environment. An analysis of the results obtained has also been given to prove the scalability of our shared memory model.

1.4 Organization of the thesis

The thesis is organized in ten chapters:

1. Introduction.
2. The Stanford's *DASH* Project.
3. Problem specification, solution model taken and why the model was taken.
4. Memory Model and the Coherence protocols designed to support the Model.
5. Communication primitives and its implementation through *RPCs*.
6. Functioning of the Memory servers (Local and Global).
7. The initialization (init) and termination (finish) routines for the shared memory environment.
8. The **TSL** Synchronization Primitive.
9. The GUI presented to the user for the software.
10. Usage of the software.
11. Applications on this environment and Conclusion.
12. Areas where there is scope for further advancement or improvement.

Chapter 2

The Stanford DASH Multiprocessor - A brief overview

2.1 Introduction

The computer Systems Laboratory at Stanford University is developing a shared-memory multiprocessor called Dash (an abbreviation for Directory Architecture for Shared Memory). The fundamental premise behind that architecture is that it is possible to build a scalable high-performance machine with a single address space and coherent caches.

The *DASH* prototype system is the first operational machine to include a scalable cache-coherence mechanism. The prototype incorporates up to 64 high performance RISC processors to yield performance up to 1.6 billion instructions per second and 600 million scalar floating point operations per second. The design of the prototype will provide deeper insight into the architectural and implementation challenges that arise in a large scale machine with a single address space. The prototype will also serve as a platform for studying real applications and software on a large parallel system.

2.2 Why Dash architecture

Most existing multiprocessors with cache coherence rely on snooping to maintain coherence. Unfortunately snooping schemes distribute the information about which processors are caching which data items among the caches. This inherently limits the scalability of these machines because the common bus and the individual processor caches eventually saturate. Directory structures avoid the scalability problems of snoopy schemes by removing the need to broadcast every memory request to all processor caches holding a copy of each memory block. Only the caches with copies can be affected by an access to the memory block, and only those caches need be notified of the access. Thus the processor caches and interconnect will not saturate due to coherence requests. Furthermore, directory-based coherence is not dependent on any specific interconnection network like the bus used by most snoopy schemes. The same scalable, low-latency networks such as *Omega* networks or *k-nary n- cubes* used by non-coherent and message passing machines can be employed.

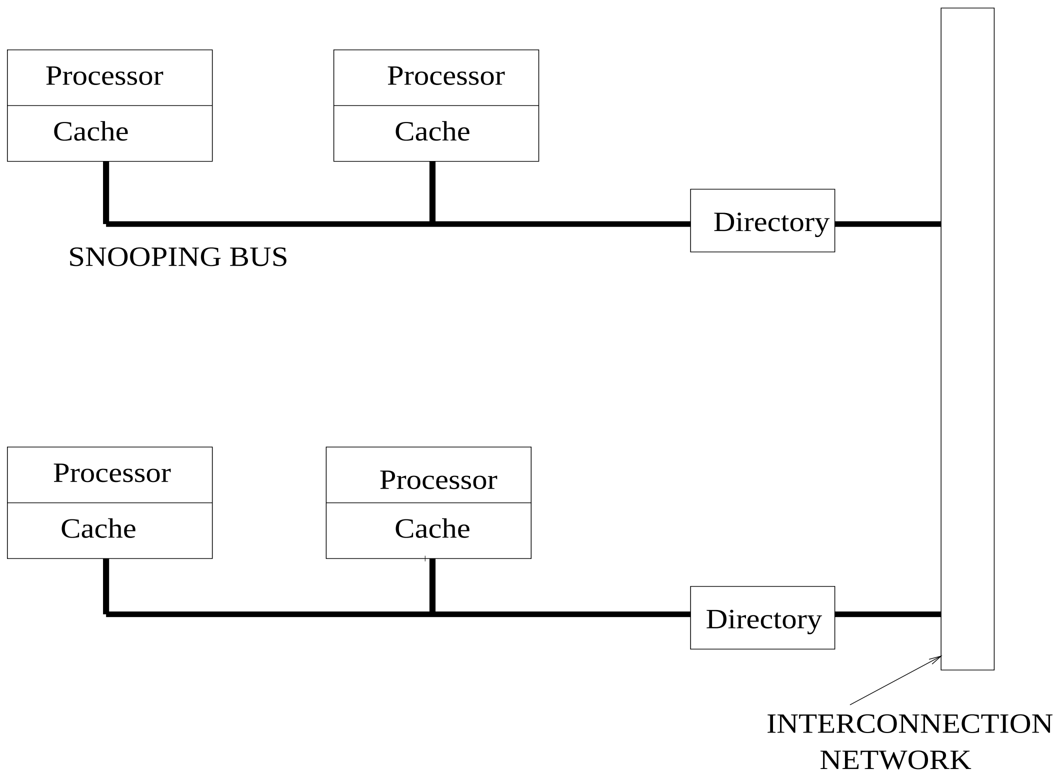


Figure 2.1: The Dash architecture consists of a set of clusters connected by a general interconnection network. Directory memory contains pointers to the clusters currently caching each memory line

The figure shows Dash's high-level organization. The architecture consists of a number of processing nodes connected through directory controllers to a low-latency interconnection network. Each processing node or *cluster* consists of a small number of high-performance processors and a portion of the shared memory interconnected by a bus. Multiprocessing within the cluster can be viewed either as increasing the power of each processing node or as reducing the cost of the directory and network interface by amortizing it over a larger number of processors. Distributing memory with the processors is essential because it allows the system to exploit locality. All private data and code references along with some of the shared references, can be made local to the cluster. These references avoid the longer latency of remote references and reduce the bandwidth demands on the global interconnect.

2.3 Memory hierarchy

Dash implements an invalidation-based cache-coherence protocol. A memory location may be in one of three states:

- *uncached* - not cached by any cluster;
- *shared* - in an unmodified state in the caches of one or more clusters;
- *dirty* - modified in a single cache of some cluster.

The directory keeps the summary information for each memory block, specifying its state and the clusters that are caching it. The dash memory system can be logically broken into four levels of hierarchy, as illustrated by figure 2. 2.

The first level is the processor's cache. This cache is designed to match the processor speed and support snooping from the bus. A request that cannot be serviced by the processor's cache is sent to the second level in hierarchy, the *local cluster*. This level includes the other processors' caches within the requesting processor's cluster. If the data is locally cached, the request can be serviced within the cluster. Otherwise, the request is sent to the *home cluster* level. The home level consists of the cluster that contains the directory and physical memory for a given memory address. For many accesses (for example, most private data references), the local and home cluster are the same, and the hierarchy collapses to three levels. In general, however, a request will travel through the interconnection network to the home cluster. The home cluster can usually satisfy the request immediately, but if the directory entry is in dirty state or in shared state when the requesting processor requests exclusive access, the fourth level must be accessed. The *remote cluster* level for a memory block consists of the clusters marked by the directory holding a copy of the block.

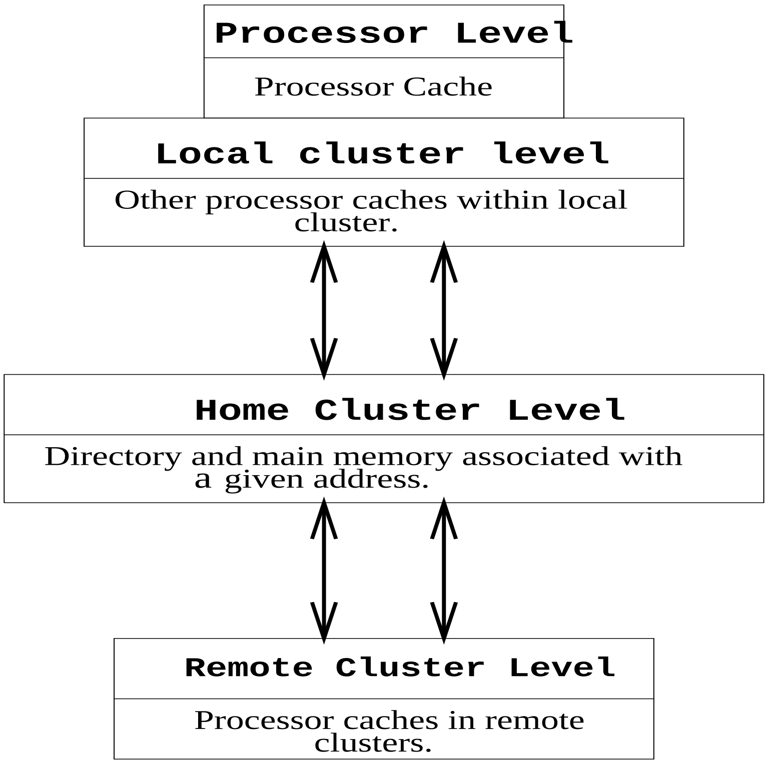


Figure 2.2: Memory hierarchy of Dash

2.4 The Dash Implementation

The prototype system uses a Silicon Graphics Power Station 4D/340 as the base cluster. The 4D/340 system consists of four Mips R3000 processors and R3010 floating-point coprocessors running at 33 megahertz. Each R3000/R3010 combination can reach execution rates up to 25 VAX MIPS and 10Mflops. Each CPU contains a 64-kilobyte instruction cache and a 64-Kbyte write-through data cache. The 64-Kbyte data cache interfaces to a 256-Kbyte second-level write-back cache. The interface consists of a read buffer and a four-word-deep write buffer. Both the first- and second-level caches are direct-mapped and support 16-byte lines. The first level caches run synchronously to their associated 33-MHz processors while the second level caches run synchronously to the 16-MHz memory bus.

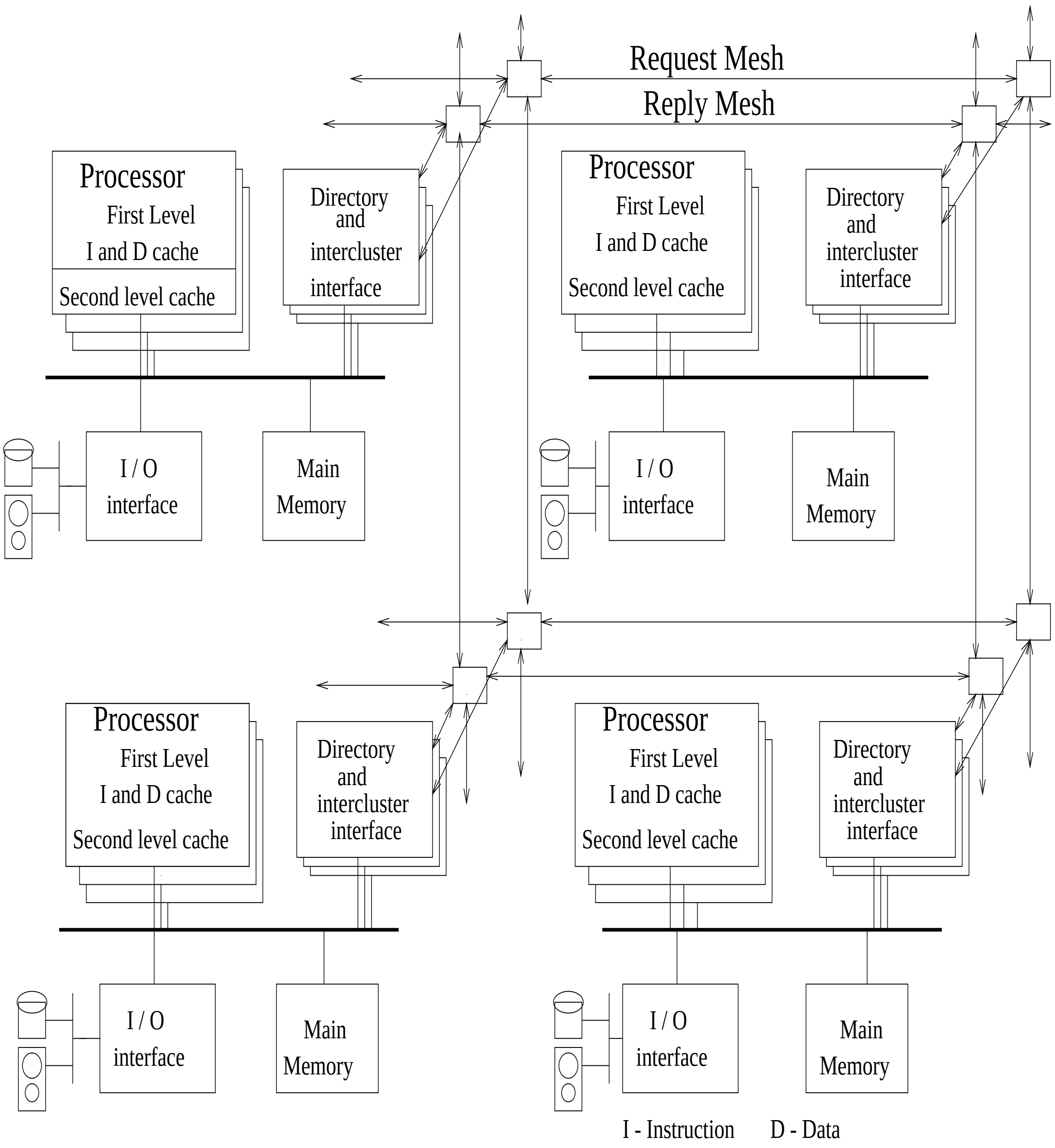


Figure 2.3: Block Diagram of a 2X2 Dash system

Chapter 3

Problem Definition and Solution

3.1 Problem Specification

A shared memory environment similar to that of DASH is to be provided on a set of workstations using the *LOCAL AREA NETWORK* for communication required for the exchange of coherence information (see figure 3.1). Since the kernel is not to be tinkered with, the environment will in effect be a software simulator which will interact with the application program to provide it the standard memory services of read and write. Apart from that it will also return the user the simulated time required to service each of its memory request. The simulated time is the time it would have taken to service the request had there been a dedicated shared memory management unit (in real hardware). Synchronization primitives (e.g. TSL) are also to be provided for synchronizing the parallel applications running on the shared memory environment.

Thus the total memory is to be divided equally among the number of processors (each processor being simulated by a workstation) such that any processor can access the allocated memory of any other processor.

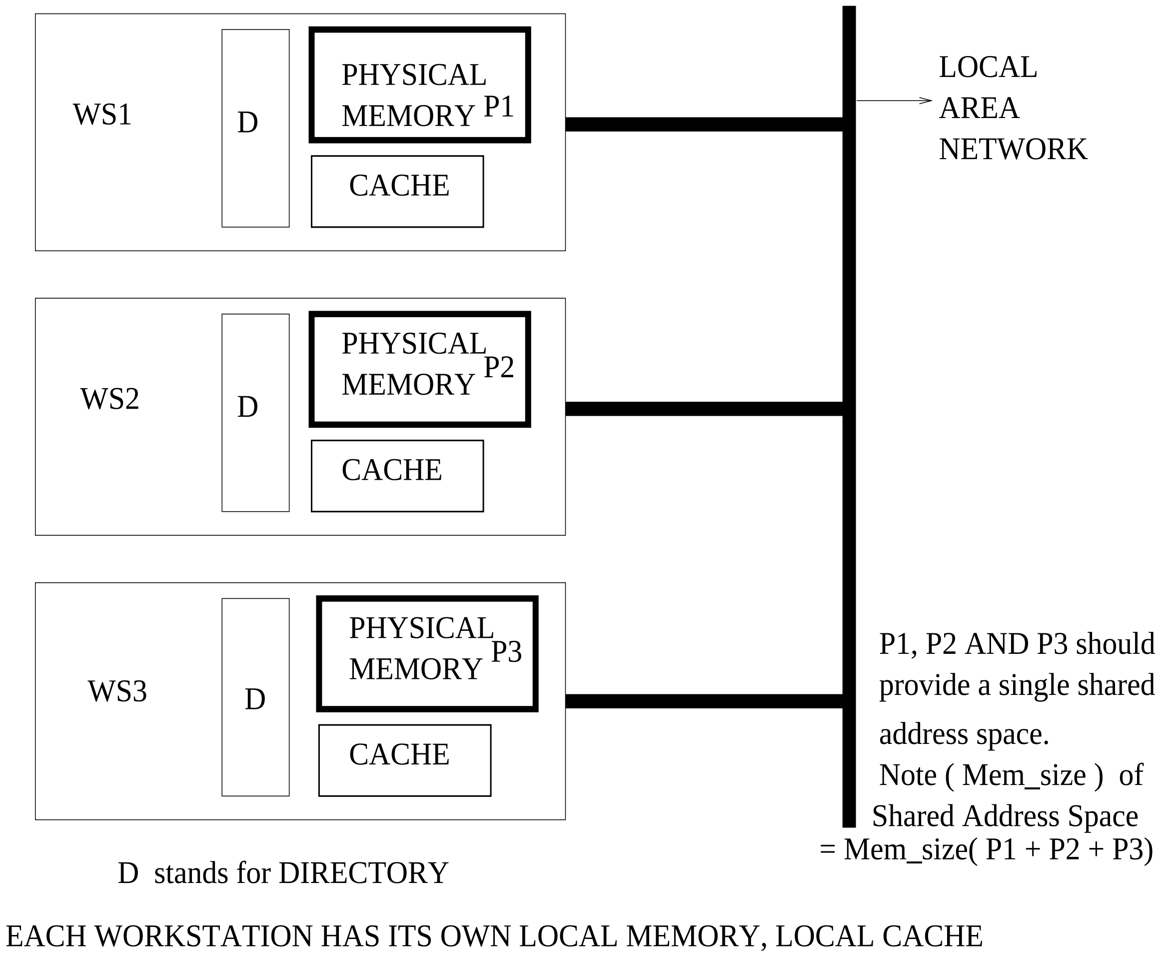


Figure 3.1: Shared Memory provided across Workstations

3.2 Solution Model

The solution model consists of three active processes running on each Work Station, one initialization and one termination routine:

proc:
: It is the application process which uses functions like membyte memword and membyte for making memory calls. These functions then issue *RPC* calls to the process lmemsrv on the same machine to service its request.

lmemsrv :

This process is Global Memory Server which receives memory calls from proc and then forks itself creating a child which now exclusively caters to servicing this request. The forked process now contacts the pmemsrv process of the same machine to service the request and if the request can't be locally serviced the process contacts the lmemsrv of the of the remote machine which has the physical memory of the address sought. See chapter 6 for a detailed discussion on the memory servers.

pmemsrv :

This process is the local memory server. The Physical Memory, the Directory and the local cache can only be accessed through this process. Thus changing the status of any block, local to the WS or reading or writing onto the cache or the local memory is to be serviced through the pmemsrv only.

init :

This is the process which initializes the shared memory environment in the format as desired by the user. Thus through a file rpc.cfg and mem.addr the user specifies the shared memory parameters and the initial values of the Physical Memory respectively. See chapter 7 for a detailed discussion on initialization.

finish :

This is the process which terminates the Shared Memory Environment in an appropriate way. That is, finish ensures that there are no background processes running after its execution.

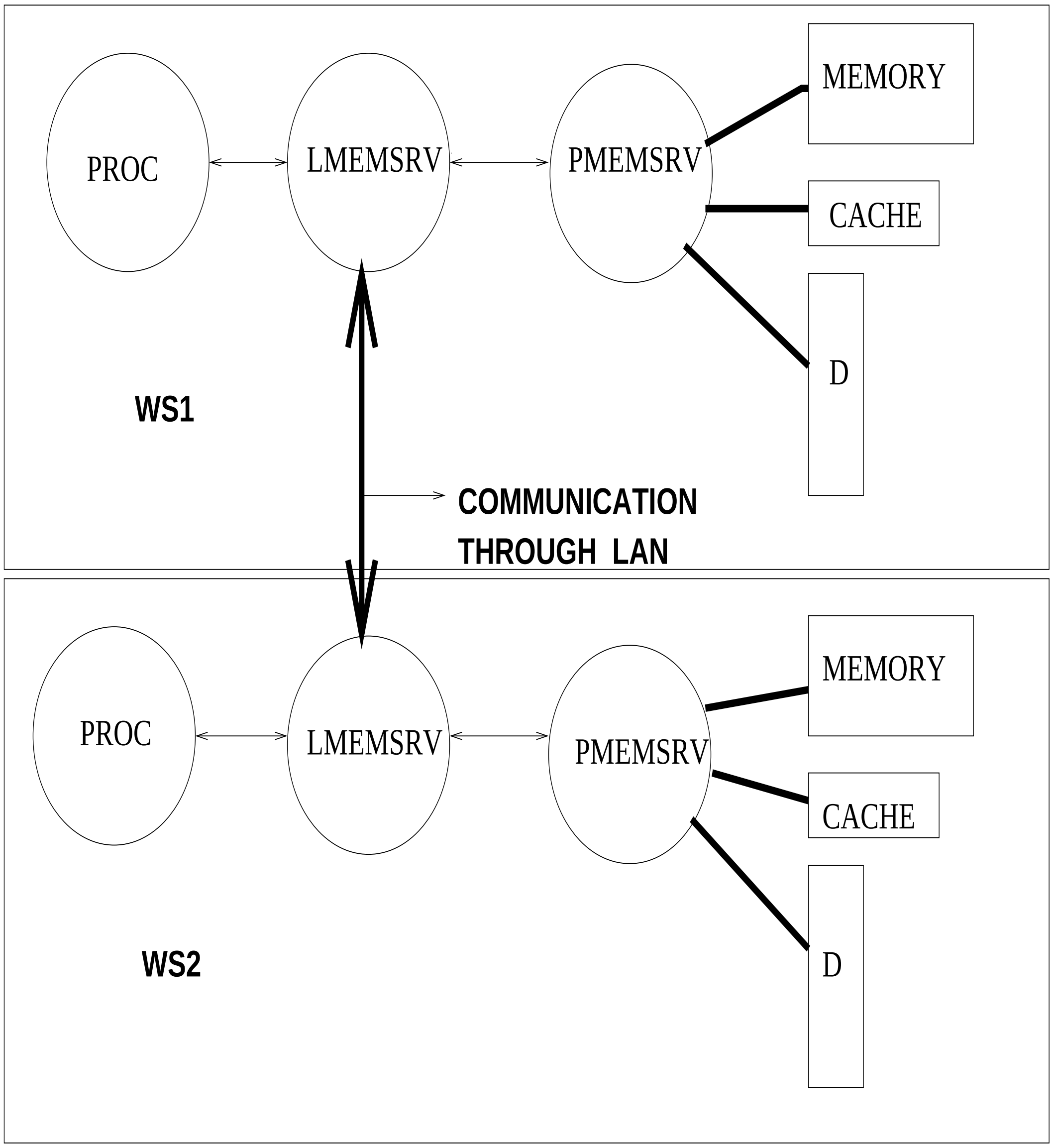


Figure 3.2: Three processes running on each workstation

3.3 Reason of Choice

The reason that two memory servers were provided instead of a single memory server are manifold and thus it warrants some explanation:

- Consider the situation as given in figure 3.3.

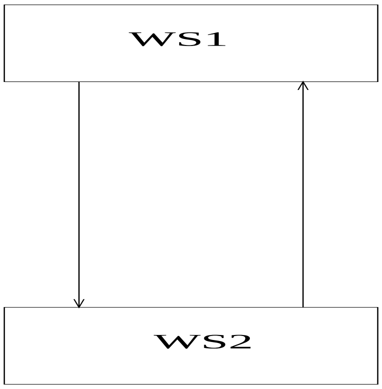


Figure 3.3: Deadlock due to both WS1 and WS2 waiting for a memory block presently residing in the other's cache. Thus each of them is waiting for the other to respond to its request resulting in a DEADLOCK CYCLE

  Thus it would be far better if the memory server forks itself and while the child takes care of the request the parent again goes back to its ready state to receive requests. This way the Memory server will always be ready to "listen" without having to compromise on the service delay.

- The Memory server will be occupying a lot of memory space because it has the physical memory, the local cache and the directory information. Now if the Server forks at each memory reference then as is known with each fork the core image also gets duplicated. Consequently a lot of memory will get unnecessarily duplicated. Thus a far more logical solution is to break the Memory Server into two viz. Global Memory Server, which deals with the broader issues of communication among various WSs, forks to service each memory request and is always ready to service any request, and the Local Memory Server which takes care of the Physical Memory, Local Cache and the Directory management.

- The need of the initialization routine is obvious because of the flexibility desired in the configuration set-up of the Shared Memory Environment.

Chapter 4

The Memory Model and Coherence Protocols

The Memory Model and the Coherence protocols are such that it closely reproduces that of *DASH* barring the minute differences that were necessary because of the fundamental difference in the hardware platform of the *DASH* and that of the problem.

4.1 Memory Model

The Memory consistency model is identical to that of the *DASH* model. The salient features of the model is that any Memory block can be in 3 states:

1. Uncached : Not cached by any processor.
2. Shared : When the memory block is in one of the Caches and it is *CONSISTENT* with its copy in the physical memory i. e. in an unmodified state in the caches of one or more processor.
3. Dirty : Modified in a single cache of some processor.

4.2 Coherence Protocols

Thus to realize this memory model a set of coherence protocols were designed. Consequently associated with each state transition there are certain actions. The state transitions are:

1. Uncached to Shared
   Reason : A processor requests a read on an uncached memory block.
   Actions : The uncached memory block is transferred to the cache of the requesting processor and then the read proceeds.

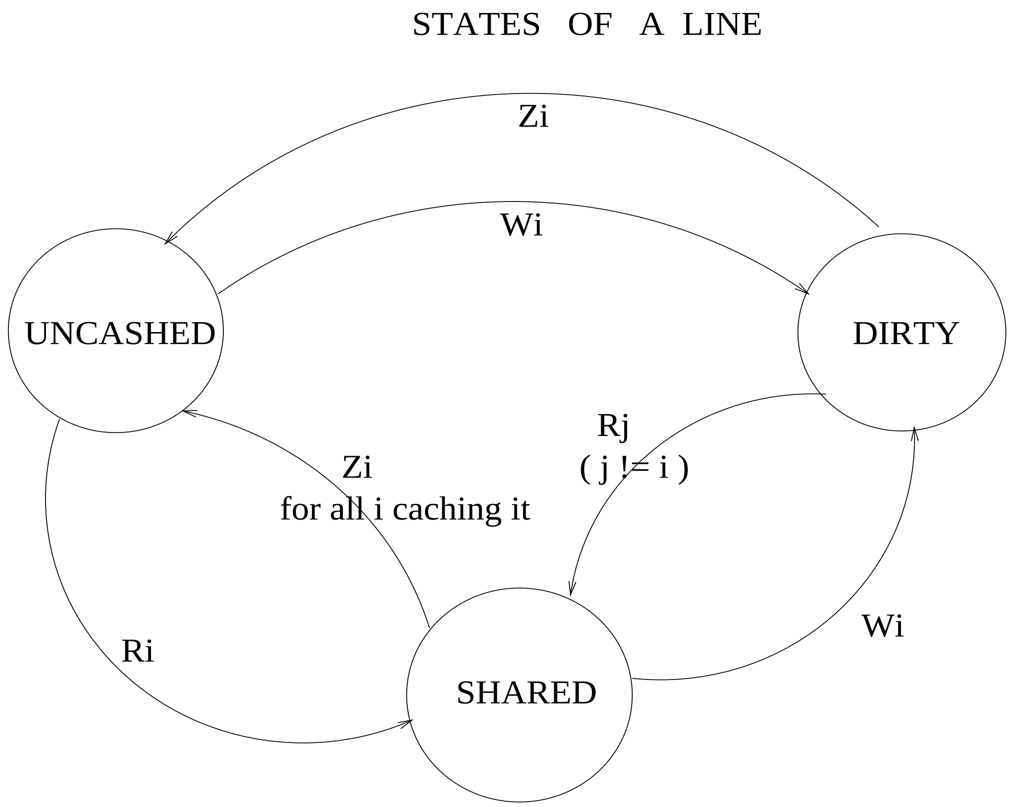
2. Uncached to Dirty
   Reason : A processor requests a write on an uncached memory block.
   Actions : The uncached memory block is transferred to the cache of the requesting processor and then the write proceeds.

3. Shared to Dirty
   Reason : A processor requests a write on a shared memory block.
   Actions : The processor first becomes the exclusive owner of the block by transferring the block onto its local cache if it is not already present. Subsequently all other processors get invalidation requests on that block.

4. Shared to Uncached
   Reason : Only one processor has a block and it is replaced by another block due to page fault.
   Actions : The block is just overwritten by the other block as the physical memory already has its consistent copy.

5. Dirty to Shared
   Reason : A processor requests a read on a dirty block cached in some other processor.
   Actions : The block is copied onto the cache of the requesting processor from the cache of the processor which had the dirty block.

6. Dirty to Uncached
   Reason : A dirty block is replaced by another block due to page fault.
   Actions : The dirty block is copied onto the physical memory such that the physical memory now has the consistent values of the block.



Ri : Read by processor i on the line

Wi : Write by processor i on the line

Zi: Line being replaced by processor i

Figure 4.1: State Transition Diagram for the Memory Blocks

To illustrate it in terms of the events that would occur in course of the execution of various memory instructions, five set of figures are given: Note that initially, only the Physical Memory has the block and the block is in uncached state. Also note that the Pi's are the individual workstations each having it's own local memory and local cache. The next four figures clearly illustrates the Memory Consistency Model.

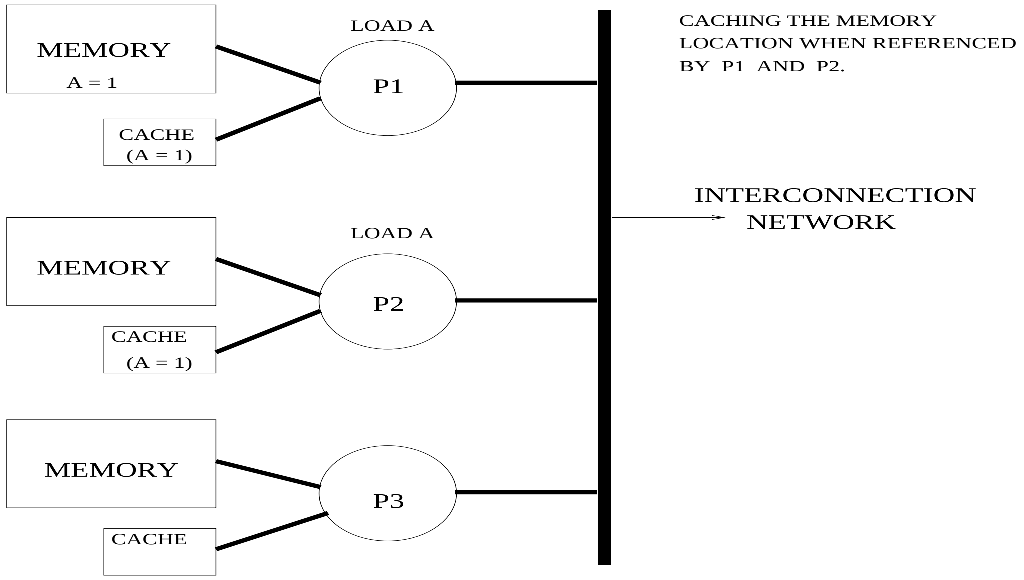


Figure 4.2: State(Block A) - Shared

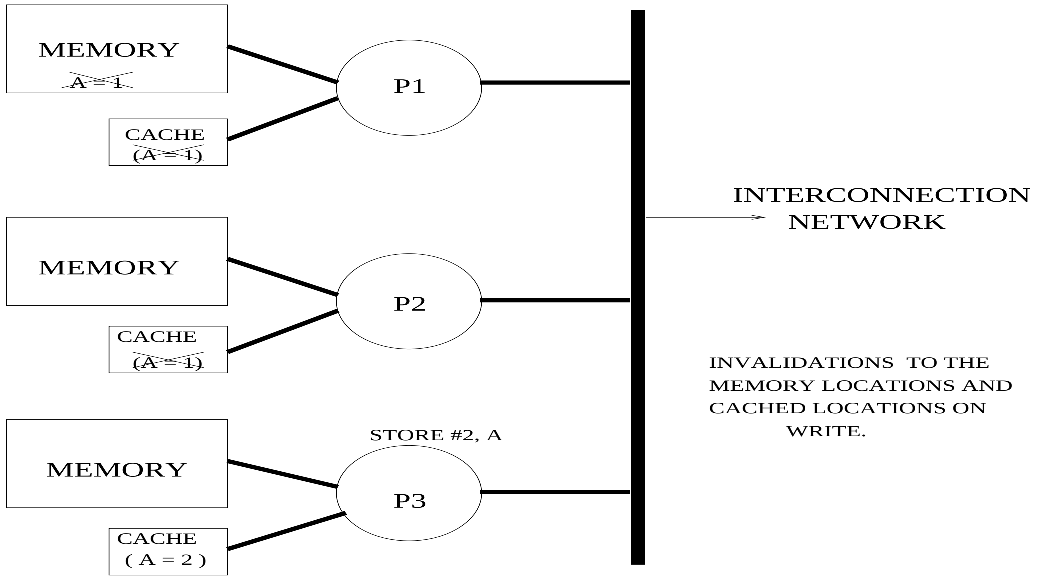


Figure 4.3: State(Block A) - Dirty

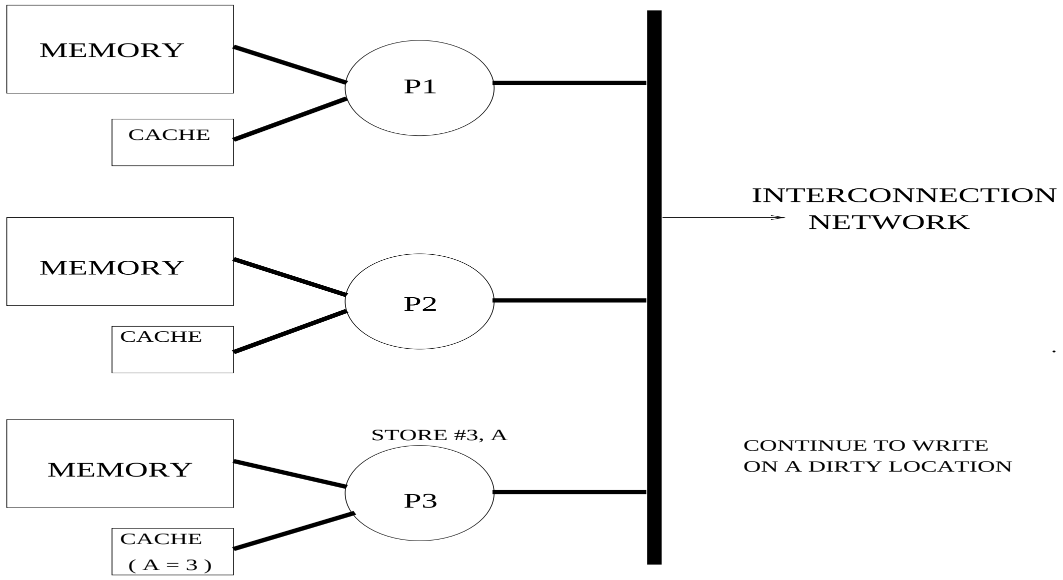


Figure 4.4: State(Block A) - Dirty

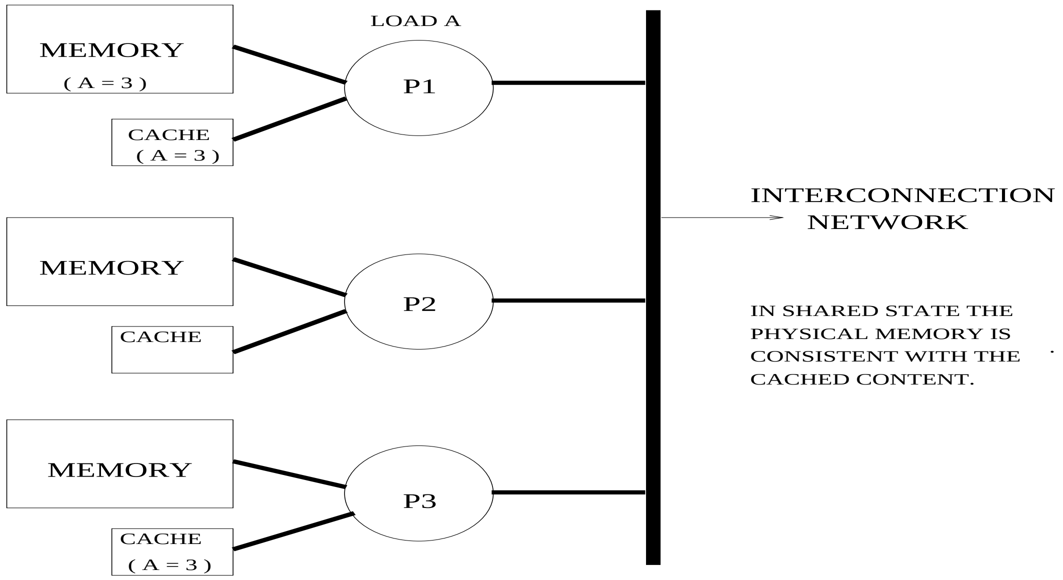


Figure 4.5: State(Block A) - Shared

Chapter 5

Remote Procedure Calls

5.1 Introduction

The Remote Procedure Call(RPC) is a high-level network applications model. RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

RPC lets network applications use procedure calls that hide the details of the underlying networking mechanism. RPC is a logical client-to-server communications system that specifically supports network applications. RPC is transport independent. It runs on whatever networking mechanisms (such as TCP/IP) are available. In figure 5.1 the client makes a procedure call that sends a request to the server and blocks. When the request arrives, the server calls a dispatch routine that performs the requested service, sends a reply to the client. The client continues.

5.2 RPC Overview

By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

In a remote procedure call, a process on the local system invokes a procedure on a remote system. The reason that it is called a procedure call is because

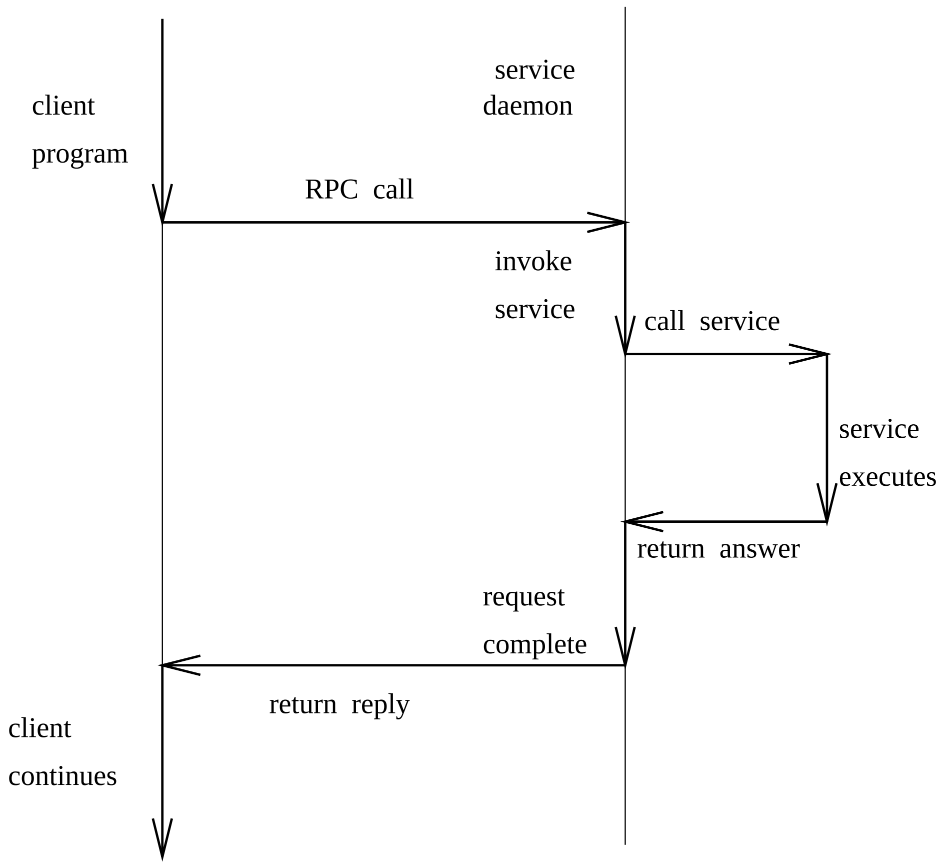


Figure 5.1: **Network Communication with Remote Procedure Call.**

the intent is to make it appear to the programmer that a normal procedure call is taking place. The term 'request' is used to refer to the client calling the remote procedure, and the term response is used to describe the remote procedure returning its results to the client.

The passing of parameters between the client and the server can be transparent. Parameters that are passed by value are simple – the client stub copies the value from the client and packages it into a network message. The problem arises if parameters can be passed by reference. Here it makes no sense for the client stub to pass the address to the server, since the server has no way of referencing memory locations on the client's system.

The SUN RPC is designed to work with either a connection-oriented or a connectionless protocol. When connectionless protocol is used, the client stub typically has to worry about the lost packets while the connection-oriented protocol handles these issues for us, but the overhead is higher when a connection oriented protocol is used.

Each RPC procedure is uniquely identified by a program number, version number, and procedure number. The program number identifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a change is made to a remote service (such as adding a new procedure), a new program number does not have to be assigned. Changes in a program, such as adding a new procedure, changing the arguments or return value of a procedure, or changing the side-effects of the procedure require that the version number be changed.

5.3 Data Representation

For RPC to function on a variety of system architectures requires a standard data representation. RPC uses eXternal Data Representation (XDR). XDR is a machine-independent data description and encoding protocol. Using XDR, RPC can handle arbitrary data structures, regardless of different hosts' byte orders or structure layout conventions.

The data representation standard used by SUN RPC is called XDR. It imposes a big endian byte ordering and the minimum size of any field is 32 bits. This means e.g. that when a VAX client passes a 16-bit integer to a server that is also running on a VAX, the 16-bit value is first converted to a 32 bit big endian integer by the client, then converted back to a little endian 16 bit integer by the server. The SUN RPC implementation uses what is called implicit lylang. That is, only the value of a variable is transmitted across the network, not the type of the variable.

Chapter 6

The Memory Servers

6.1 The Local Memory Server - pmemsrv

The *local memory server* on each processor is the sole manager of the:

(a) Local Cache;

(b) Local Directory;

(c) Physical Memory.

It interacts with the lmemsrv process (Global Memory Server), running on its machine through the function pmem_server_1() and performs the various local tasks necessary to manage the shared memory.

The functions of the pmemsrv are:

- Managing the Cache - The read and write from the cache blocks, reporting a miss or a hit, invalidating cache blocks, replacing cache blocks on some cache replacement policy in case of a cache-miss.
- Managing the Physical Memory - Reading or writing a block from the Memory.
- Managing the Directory - The directory maintains pointers to various processors which cache its memory blocks. It also maintains a queue of the process pids waiting upon a memory block e. g. write waits on a dirty block residing in some remote cache. Thus the directory requires continuous updation which is handled by the pmemsrv.

The messages it uses to perform its functions are:

LOAD_CACHE - Whenever a processor accesses a memory location which is not in its cache or its physical memory but in some other machine then the block has to be fetched from the other machine and loaded onto the local cache. Thus it is like a cache-miss and it is only after loading the cache that the memory request can proceed. The lmemsrv requests the pmemsrv to *load the cache* with the block it has fetched and the pmemsrv does it.

SEND_UPDATE - The pmemsrv senses the need to send an update to a remote processor, when a dirty block is thrown out in the process of cache replacement by another block, and the replaced dirty block belongs to the physical memory of some remote processor. Thus it sends a SEND_UPDATE to the lmemsrv which subsequently sends an *update signal* to the lmemsrv of the *home processor* i. e. the processor which actually holds the physical memory of the block.

UPDATE - When the lmemsrv receives an update block from a remote machine it forwards the block to the pmemsrv which then updates the physical memory and the directory appropriately.

INVALIDATE - Whenever a processor is interested in writing onto a block which is cached in some other processor or processors, invalidate signal is to be sent to invalidate the copies of that block in other caches. The lmemsrv of the other processors receive the invalidate signal and requests the pmemsrv to invalidate the cache line having the referred block. The pmemsrv then does it.

DONE_INVALIDATE - After the lmemsrv sends the invalidate signal all memory requests on that block wait until the invalidate message is serviced. Thus after its completion the lmemsrv sends a DONE_INVALIDATE signal to the pmemsrv which then frees the queue of all the processes waiting on the block.

READ_DIRTY & READ_DIRTY_AND_INVALIDATE - When the pmemsrv of the *home processor* sees in its directory that the referred block is lying dirty in the cache of some remote processor it issues a READ_DIRTY or a READ_DIRTY_AND_INVALIDATE request depending on whether the memory request was read or write respectively. The lmemsrv then forwards the request to the remote machine which is caching the referred memory block. Now, the lmemsrv of the remote machine issues the READ_DIRTY or READ_DIRTY_AND_INVALIDATE command to the pmemsrv of its own machine. On receiving it the pmemsrv reads the dirty block from cache, invalidates the cache line if the request is READ_DIRTY_AND_INVALIDATE and then forwards the block to the lmemsrv. The lmemsrv in turn forwards the block to the *home processor* for further action. Note that the home processor is the processor which actually holds the physical memory of the block.

DONE_READ_DIRTY & DONE_READ_DIRTY_AND_INVALIDATE - After the lmemsrv of the *home processor* gets the dirty block from the remote processor in return to its READ_DIRTY or READ_DIRTY_AND_INVALIDATE request it sends a DONE_READ_DIRTY or DONE_READ_DIRTY_AND_INVALIDATE to the pmemsrv. The pmemsrv then updates its physical memory in case the request is READ_DIRTY, (for after the READ_DIRTY is satisfied the memory block becomes shared and in the shared state the physical memory should be consistent to the cached contents) updates the directory status of the block accordingly and then sends the block to the lmemsrv which then forward it to the processor which accessed the address. In case the *home processor* itself accessed the address it loads the block onto its cache and then satisfies the read or write request accordingly.

CHECK_LOCAL_CACHE - When the pmemsrv receives the CHECK_LOCAL_CACHE request it checks if the memory block is in the local cache and in the proper status, (i.e. if request is write and status is shared, then the status is **not** appropriate) then the request is immediately satisfied and a hit occurs else a miss is reported back to the lmemsrv.

CHECK_LOCAL_MEMORY - This request involves the maximum complexity in terms of the actions required to be taken on receipt of this request. On receiving this memory request the pmemsrv does the following in chronological order:

(a) Checks if there is any other process pid waiting for the same memory block. In case there is then it enqueues the process pid of the lmemsrv which made the request to the pmemsrv and then goes to a wait state until the queue is freed. It also reads the status of the block.

(b) If request is read and the status of the block is shared or uncached:
   - if the *home processor* made the request then it loads the block onto the cache, changes the entry of the directory accordingly and then reads the address from the cache.

- if any other processor has tried to access the block then it reads the block from the memory, changes the entry of the directory accordingly and then forwards it to lmemsrv which in turn forwards it to the lmemsrv of the processor which tried to access the block.

(c) If request is read and the status of the block is dirty it reads the block from the memory, changes the entry of the directory accordingly and then forwards it to lmemsrv which in turn forwards it to the processor which tried to access the block.

(d) If request is write and status of block is uncached:
- if the home processor made the request then read the block from the memory, load the uncached block on the cache, change the directory entry accordingly and then write on the cache line at the desired address.
- if any other processor has tried to access the block then it reads the block from the memory, changes the entry of the directory accordingly and then forwards it to lmemsrv which in turn forwards it to the processor which tried to access the block.

(e) If request is write and the status of the block is shared:
- Sends invalidate to all the processors which cache the block except the one which requested the memory-write.
- Queues its pid in the directory queue of the block such that if any other processor requests the block it can only proceed after this request has been satisfied.
- Reads the block from the memory.
- If the home processor made the request then load the block on its cache and then proceed with the write
- If some other processor (say P) has requested then the block is read from the memory and passed to the lmemsrv which then forwards the block to the lmemsrv of P.

WAIT - When the lmemsrv of a processor sends a request to the pmemsrv and the pmemsrv senses that the request cannot be immediately satisfied then it sends a *WAIT* request to lmemsrv. The lmemsrv on receipt of the *WAIT* request goes into a waiting state and makes the request to pmemsrv only after it gets a signal to continue (SIGUSR1) from the pmemsrv.

6.2 The Global Memory Server - lmemsrv

The lmemsrv communicates with three processes:

(a) proc running on its own machine - through the function lmem_server_1().

(b) lmemsrv running on another machine - through the function lmem_server_2().

(c) pmemsrv running on its own machine - through the function pmem_server_1().

The steps it takes in chronological order when it receives a memory request from the proc are:

(a) Determines the processor which has the physical memory of the address accessed.

(b) Calls the check_local_cache() function. Detailed explanation of the check_local_cache() function is given in section 6.3.2 .

(c) If last request fails it does the following:

  - If it is the *home processor* then it calls the check_local_mem() function. Detailed explanation of the check_local_mem() function is given in section 6.3.1 .
  - If some other processor is the *home processor* then the lmemsrv:
    * Sends a CHECK_LOCAL_MEMORY request to the lmemsrv of that processor. The lmemsrv of that processor subsequently takes care of the request and passes the whole block corresponding to that address to the lmemsrv of this processor.
    * After getting the block it requests the pmemsrv to LOAD_CACHE with this block. The pmemsrv subsequently loads the cache and satisfies the memory request.
    * If due to loading of the cache the lmemsrv gets a SEND_UPDATE request from the pmemsrv then it sends the block to the lmemsrv of the *home processor* of the block returned with an UPDATE message.

The lmemsrv takes the following steps when it receives a request from the lmemsrv of another processor:

- If the request is update it sends an UPDATE request to the pmemsrv.
- If the request is INVALIDATE it sends an INVALIDATE request to the pmemsrv.

- If the request is READ_DIRTY or READ_DIRTY_AND_INVALIDATE it sends the request to the pmemsrv and gets the dirty block from the physical memory. Subsequently it forwards the dirty block to the lmemsrv of the *home processor*.
- If the request is CHECK_LOCAL_MEMORY then it calls the check_local_mem() function. Subsequently it forwards the memory block received from the pmemsrv to the lmemsrv of the processor which had called the lmemsrv of this processor.

6.3 Important Functions

6.3.1 check_local_mem() function

The **check_local_memory**() takes the following steps in chronological order:

(a) Keeps requesting CHECK_LOCAL_MEMORY to the pmemsrv until WAIT answer is not received.

(b) If returned message is READY then the request has been satisfied. Further if the pmemsrv has sent:
  - SEND_UPDATE then send UPDATE to the lmemsrv of the *home processor* of the block.
  - SEND_INVALIDATE then send INVALIDATE for the block to the lmemsrv of all those processors, which are in the invalidate list, returned by the pmemsrv. After completing the invalidation send a DONE_INVALIDATE to the pmemsrv and then return.

(c) If returned message is READ_DIRTY or READ_DIRTY_AND_INVALIDATE then send a READ_DIRTY or READ_DIRTY_AND_INVALIDATE (as the case may be) request to the lmemsrv of the remote processor caching the dirty block and get the block from there.

(d) After getting the dirty block send a DONE_READ_DIRTY or DONE_READ_DIRTY_AND_INVALIDATE to the pmemsrv along with the dirty block.

6.3.2 check_local_cache() function

The **check_local_cache**() sends a CHECK_LOCAL_CACHE request to the pmemsrv. If the pmemsrv returns a HIT then it returns 0 (implying success) else it returns -1 (implying failure).

Chapter 7

Initialization and Termination

7.1 Initialization of the Shared Memory Environment

The initialization process (init) basically initializes the shared memory parameters and sets the initial value of the physical memory of each workstation.

7.1.1 Setting the configuration parameters

The shared memory parameters are entered through the rpc.cfg file which contains the configuration parameters. The parameters are:

(a) block_size - It is the block size of the memory.

(b) highest_mem_addr - The shared memory address would range from 0 to this highest_mem_addr - 1. Thus it is the total size of the shared memory provided.

(c) num_procs - It is the total number of processors which will share the memory. The total memory will be divided equally among the WSs and all of them will share it.

(d) sched_strategy - There are many scheduling strategies a memory manger can use for replacing cache lines in the case of a miss. In this software only the FCFS strategy is implemented.

(e) map - This is used to define the mapping strategy that one is going to use. Three types of mapping have been implemented viz DIRECT, FULLY

ASSOCIATIVE and SET ASSOCIATIVE and the user can choose any one of them.

(f) cache_size - This sets the size of the cache that each processor would be having.

(g) num_sets - If the type of mapping is SET ASSOCIATIVE then the number of sets can be specified through num_sets.

(h) hostnames - Through this list the user specifies the name of the WSs on which the processors are to be simulated. This software implicitly assumes that the first mentioned machine will have the lowest share of addresses, the next machine the second lowest and so on. The hostnames must be separated by commas and given within flower braces.

See the appendix for getting a sample file format of rpc.cfg . Using lex and yacc the rpc.cfg parameters are read by the init.c program which then makes the init_pmemsrv_1() RPC of the pmemsrv of all the processors and the initialization of the pmemsrv is done. Similarly the initialization of the lmemsrv is done by making the init_lmemsrv_1() RPC of the lmemsrv of all the processors.

7.1.2 Initializing the physical memory

The initialization of the memory is done by making the load_local_memory_1() RPC for all the processors which in turn calls the load_memory() function of the lmemsrv of each processor. The load_memory function assumes the existence of a file memȧddr. The mem.addr must be in the following format: addr1 size1 byte1 byte2 byte3 .... addr2 size2 byte1 .....
The first entry addr is the starting address of the Shared Memory. The size is the share of the number of bytes that are reserved for the physical memory of the processor. Thus the load_memory() would load size1 number of bytes (as specified by byte1 byte2 ..) from the starting address addr1 to processor number 1. Similarly it would load size2 number of bytes from starting address addr2 to processor number 2 and so on.

7.1.3 The init_pmemsrv() function

This function does the following in chronological order:

(a) Sets the memory parameters such as block size, cache size etc.

(b) Calls init_cache() function which in turn sets the cache parameters such as num of sets, mapping strategy etc and allocates memory for the cache.

(c) Calls init_local_mem() function which in turn allocates memory for the directory and the physical memory and initializes the directory i.e. sets the default directory entries for all memory blocks.

7.1.4 The init_lmemsrv() function

The init_lmemsrv() creates handle for communicating with lmemsrv of the other processors and another handle for communicating with the pmemsrv of its own machine.

7.2 Termination of the Shared Memory Environment

The termination of the Shared Memory environment had to be handled carefully because there were a lot of background and forked processes running on each machine to support this environment. The termination is basically done by finish.c which first creates handle to the pmemsrv and lmemsrv of all the processors and makes an RPC call to the finish_pmemsrv_1() and the finish_lmemsrv_1() respectively. The finish_pmemsrv_1() in turn frees the memory allocated to the physical memory, the local directory and the local cache. Similarly the finish_lmemsrv_1() frees all the memory that it had allocated during init_lmemsrv and also destroys all the communication handles created to the lmemsrvs of other processors. After these steps the individual pmemsrvs and the lmemsrvs send an RPC reply to the finish process and then call exit().

Chapter 8

The TSL Synchronization Primitive

8.1 Introduction

We have provided the calls test_and_set_lock() and release_lock() to the user to enable him to deal methodically with the programs which concurrently access shared variables.

The TSL instruction sets a hardware register and returns its previous value. The access is guaranteed to be atomic. If the value returned by TSL is 0 then the lock has been acquired, else the process goes into a busy wait executing the TSL instruction again and again till it gets the lock. It is the responsibility of the user to reset the lock register by executing release_lock(), otherwise the register will be locked for ever. A special timeout can be provided after which a signal can be sent to the physical memory server to clear the lock.

8.2 Implementation

Basically the system has NUM_LOCK_REGS number of lock registers. The macro NUM_LOCK_REGS is defined in cosmic.h. These lock registers are divided equally among the physical memory servers. If equal division is not possible, then the remaining registers are given to the processor having the highest proc_id. The argument taken by both the above function calls is the number of the lock register, which can vary from 0 to NUM_LOCK_REGS-1.

The process which simulates the processor makes a call to the logical memory server on that machine. The logical memory server decides on which machine does the lock register reside and accordingly forwards the request either to the foreign logical memory server or if the lock register lies in the current machine, then the physical memory server is called. Hence ultimately the call reaches the physical memory server which sets the lock register and returns its previous value. The access is guaranteed to be atomic because the physical memory server can only handle one request at a time.

Chapter 9

Applications and Conclusion

All the applications written on this shared memory environment are provided in the btp/examples directory.

9.1 Producer-Comsumer Problem

The producer-consumer model defines the following problem. The producer produces items and keeps on putting it into a buffer of limited space as long as the buffer is not full. The consumer keeps on consuming items from the buffer as long as the buffer is not empty.
Thus there are two processes:

prod_prog - The producer process which keeps on producing random numbers i.e. items and writes it into the shared memory i.e. the limited buffer space.

cons_prog - The consumer process which keeps on consuming i.e. reading random numbers from the buffer.

We have kept a shared variable COUNT which keeps a track of the number of items in the buffer. Since the updation of COUNT is done by both the processes its *exclusive* access at any instant is guaranteed by the use of a TSL instruction.

9.2 Square of the Norm of a Vector

The norm of a vector V is defined as:

$$||V|| = \sqrt{\sum_{i=0}^{n-1} a_i^2} \tag{9.1}$$

An application has been written to compute the square of the norm of a vector.

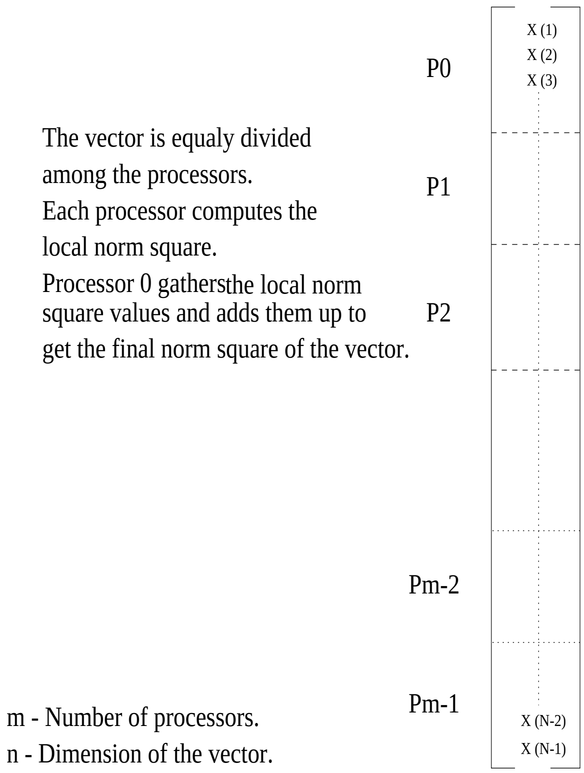


Figure 9.1: Computation of norm square

As shown in the figure the vector has been partitioned among the processors on which the shared memory is built. The elments of an n-dimensional vector range from 1 . . . n-1. Although the diagram shows a linear distribution of the vector elements the distribution is modular m where m is the number of processors. This is for load balancing. Thus the first processor will have 0, m, 2m .... The second processor will have 1, $m+1$, $m+2$ and so on. Each of the processor computes its local norm square i.e.

$$||V_{local}|| = \sum_{local\_i} a_i^2 \tag{9.2}$$

After computing this the processors place their respective $||V_{local}||$ in specified locations. Each of them then writes the time required for computation in another specified location and subsequently changes the value of a flag, protected by the TSL instruction to ensure exclusive access, from 0 to 1 to specify that the square of local norms have been computed.

The processor 0 after computing its local norm square collects and adds the other local square norm values from other processors by reading them from the specified locations. It also computes the total *simulated* time of computation required for doing these. Obviously before doing that it checks that the flag value, whose access is protected by the TSL instruction to ensure mutual exclusion, has been changed to 1. Note that the methodology used to ensure *Write before Read* is a generic one and can be used to enforce any sort of order e.g. *Read before Write.*

At the end it prints the value of the square of the norm of the vector, the total time required for computation and the time of computation per element of the vector. The results obtained are presented in the following section.

9.2.1 Results

The notations are:

m - The total number of processors which are used to compute the value.

n - The dimension of the vector.

Value -

T - The total time required for computation.

t - Time of computation required per processing element.

m	n	Value	T	t
2	8	140	40	5.0000
4	8	140	20	2.5000
2	16	1240	80	5.0000
4	16	1240	40	2.5000
2	32	10416	160	5.0000
4	32	10416	80	2.5000
2	64	85334	6560	102.5000
4	64	85334	160	2.5000
2	128	690880	11520	90.0000
4	128	690880	6560	51.2500
2	256	5559680	21440	83.7500
4	256	5559680	11520	45.0000
2	512	44608250	41280	80.6250
4	512	44608250	21440	2.500
2	1024	357389824	80960	79.0625
4	1024	357389824	41280	40.3125

9.2.2 Analysis

The analysis of the results show the improvement of performance of the application in terms of computation time when the number of processors are increased. In fact the computation time becomes halved when the number of processors are doubled.

9.3 Conclusion

The previous examples thus corroborates the claim of the *DASH research group* of the linear scalability of the **Shared Memory environment**. Thus when the number of processors are doubled the improvement in performance was also improved by two-folds.

Chapter 10

Usage of the software

10.1 Installation

The software can be installed in a network of UNIX hosts (maybe heterogeneous) running 4.3 BSD operating system or a superset. Copy the tarred and zipped file btp.tar.gz into any directory and issue the command:

```
gunzip -c btp.tar.gz | tar xvhf - .
```

10.2 Setting the shared memory environment

Do the following before running any application:

(a) change directory to btp/code/init and edit the **rpc.cfg** file and the **mem.addr** file. The file mem.addr is a binary file. The format of the file mem.addr is: The first four bytes contain the address, the next four bytes contain the number of bytes which are followed by those many bytes of data. The next four bytes are again the address and so on. The file mem.addr is used for the initializing the memory when the system is first started. Absence of the file mem.addr will issue a warning that the memory locations have not been initialized.

(b) change directory to btp/code/bin and just type: make spotless.

(c) type: make.

(d) type: lmemsrv & pmemsrv & - on each of the machine to get the Memory servers running in background.

(e) type: init - on any one machine to initialize the configuration of the environment.

10.3 Running an application

(a) compile your application program, say app.c and then link it with btp/code/proc/libproc.a to make proc.

(b) type: proc.

(c) a few examples are provided in btp/code/examples.

10.4 Testing

For testing the code add the option -DDEBUG to CFLAGS in the makefiles. While running, the various processes will generate several helpful messages which will enable the program to be debugged. To avoid deadlocks, lmemsrv provides a concurrent server. So with each new request it forks itself into a child which carries out the request. If -DITERATIVE is defined in CFLAGS in the makefile in btp/code/lmem, then lmemsrv will provide an iterative server. lmemsrv can then be debugged with standard Unix debuggers like dbx and gdb.

Chapter 11

Graphical User Interface

11.1 Introduction

Working at the command line can be irksome. Consider the amount of work a user has to do for just setting up the system. He has to edit the rpc.cfg file, he has to log in to each individual host to start up the servers lmemsrv and pmemsrv, then he has to initialize the servers using the init program. All this work just goes in settting up the system. For running application programs, the user has to again log in to the individual hosts and start his application. For stopping the servers, the program finish has to be called. All this pain can be alleviated if an interface program can be provided which asks the user for configuration parameters, starts up the servers, initializes them and also has the capability for starting up application programs and showing their output results without causing the trouble of making the user log in to various hosts.

11.2 Overview

We have provided a user interface in X using the Athena Widget set. The interface is capable of asking the user for the configuration parameters and then preparing the rpc.cfg file. It will then start up the servers on various hosts and then initialize them. All this can be done simply by clicking a few buttons – no need of logging in to different hosts for the painful job. The user can now stop this program and is only left with the job of logging in to individual hosts to run his applications.

We have also provided the facility of starting application programs from within the user interface program itself. In this case the program opens new windows which show the output results of the program. However the program cannot take user input from application programs dynamically. So only those applications can be run which do not require any user input. The output results will be displayed in the individual windows corresponding to each host in the system.

11.3 Usage

The user interface program uses "rsh" commmand for running a command on the remote hosts. So it is necessary that access across the hosts is provided through .rhosts files. On some systems, specifically HP-UX, the "rsh" program has been replaced by "remsh" program. In such systems, before compilation, the macro REMOTE_COMMAND defined at the beginning of the file xsetup.c should be suitably changed.

The code for the user interface resides in the directory btp/code/gui. A makefile is provided, which will compile the program into the binary executable called xsetup. Resource setttings are defined in the file XSetup in the same directory. Before starting the program xsetup, the resource file should either be loaded with the command "xrdb -merge XSetup" or one can set the environment variable XAPPLRESDIR to the directory in which the file XSetup resides. Full information about running the program has been provided in the README files in each directory.

Chapter 12

Scope for additional features and Improvements

12.1 Additional features

12.1.1 Synchronization Primitives

Any parallel program requires synchronization mechanism to coordinate it's effort with other processes. To explain this point consider a Producer - Consumer application program. The producer goes on writing onto the shared memory while the consumer reads from the memory. But a basic understanding between the two processes must be there such that the consumer doesn't consume when the buffer (in this case the shared memory is empty), and similarly, the producer shouldn't produce when the buffer is full. Although the *TSL* synchronization primitive has been provided additional primitives such as semaphores, queue based locks, fetch-and-increment and fetch-and-decrement operations can be provided for effectively using the Shared Memory Environment.

12.1.2 Memory Consistency Model

The memory consistency model supported by an architecture directly affects the amount of buffering and pipelining that can take place among memory requests.

At one end of the consistency model spectrum is the sequential consistency model, which requires the execution of the parallel program to appear as an interleaving of the execution of the parallel processes on a sequential machine. *Sequential Consistency* can be guaranteed by requiring a processor to complete one memory request before it issues the next request. Sequential consistency while conceptually appealing, imposes a large performance penalty on memory accesses.

As an example, consider the case of a processor updating a data structure within a critical section. If updating the structure requires several writes, each write in a sequentially consistent system will stall the processor until all other cached copies of that location have been invalidated. But these stalls are unnecessary as the programmer has already made sure that no other process can rely on the consistency of that data structure until the critical section is exited. If the synchronization points can be identified, then the memory need only be consistent at those points. In particular the *release consistency* model can be supported, which only requires the operations to have completed before a critical section is released.

12.2 Improvements

12.2.1 Memory Access Optimizations

The following features can be incorporated to optimize memory access time:

Prefetch Operations - A prefetch operation is an explicit nonblocking request to fetch data before the actual memory operation is issued. Nonblocking prefetch allows the pipelining of read misses when multiple cache blocks are prefetched.

Update and deliver operations - The *update-write* operation sends the new data directly to all the processors that have cached the data, while the *deliver* operation sends the data to specified processors. These semantics are particularly useful for event-synchronization, such as release event for a barrier.

Appendix A

The memory calls

membyte (int request, unsigned int addr, char *data, unsigned *tm_ticks_ptr) - This request is used to read or write one byte of data.

memhword (int request, unsigned int addr, char *data, unsigned *tm_ticks_ptr) - This request is used to read or write two bytes or a half-word of data.

memword (int request, unsigned int addr, char *data, unsigned *tm_ticks_ptr) - This request is used to read or write four bytes or one word of data.

A.1 Parameters' description

The parameters can be:

request - READ or WRITE.

addr - 0 to *highest_mem_addr* (as defined in your rpc. cfg file).

data - string of 1, 2 or 4 bytes of data for request of memword(), memhword() or memword() respectively.

tm_ticks_ptr - The total simulated time to service this memory request is returned through this parameter.

Appendix B

The TSL function calls

test_and_set_lock (unsigned int reg_num) - Atomic action which reads the value of the Register having the register number reg_num and then sets it to 1 thus grabbing the lock.

release_lock (unsigned int reg_num) - Atomic action which sets the value of the Register having register number reg_num to value 0 thus releasing the lock.

B.1 Parameter's description

The parameter is register_number. Note that there are a finite number (as defined by the macro NUM_LOCK_REGS) of TSLs provided in the shared memory environment. The TSLs are divided equally among the number of processors. Thus any processor can use any TSL, which are identified by their register number varying from 0 to to NUM_LOCK_REGS - 1, to synchronize its activity with any other processor.

Appendix C

Sample file of rpc.cfg

This file specifies the configuration of the system

the number of bytes in a block
$block\_size = 1024$
the highest memory address (the lowest is assumed to be 0)
$highest\_mem\_addr = 0xffffffff$
number of processors in the system
$num\_procs = 2$
scheduling strategy to be used for page faults
$sched\_strategy = FCFS$
specify the type of mapping -> DIRECT, FULL or SET
$mapping = SET$
specify the cache size in bytes
$cache\_size = 65536$
specify the number of sets
$num\_sets = 1$
# specify the names of hosts on which the processors are to be simulated
hostnames = { dhoop, baadal }

Bibliography

[1] Daniel Lenoski, James Laudon, Kourosh Gharachorlo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz and Monica S. Lam, "The Stanford Dash Multiprocessor", *Proc. 1992 Int'l Conf. Parallel Processing*, IEEE Computer Society Press, Los Alamitos, Calif., Order No. 9162, 1992, pp. 63-78.

[2] A. Gupta, W-D Weber and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes", *Proc. 1990 Int'l Conf. Parallel Processing*, IEEE Computer Society Press, Los Alamitos, Calif., Order No. 2101, pp. 312-321.

[3] B.W. O'Krafka and A.R. Newton, "An Empirical Evaluation of Memory-Efficient Directory Methods", *Proc. 17th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., Order No. 2047, 199, pp. 138-147.

[4] C. Scheurich and M. Dubois, " Dependency and Hazard Resolution in Multiprocessors ", *Proc. 14th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., Order No. 776, 1987, pp. 234-243.

[5] Unix Network Programming, Richard Stevens.

[6] Solaris Manuals on Networking.