DISTRIBUTED SCHEDULING IN SOMBRERO, A SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

Milind Patil

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 1999

DISTRIBUTED SCHEDULING IN SOMBRERO, A SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

Milind Patil

has been approved

August 1999

APPROVED:

_____, Chair

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Graduate College

ABSTRACT

Sombrero is a 64-bit single address space distributed operating system. It manages the set of resources in the distributed system: central processing units (CPUs), hard drives, printers, et cetera, as a single virtual system. This thesis adds a distributed scheduling mechanism to Sombrero. This distributed scheduling algorithm provides for sharing, as well as better usage of resources across the system. It allows threads in the distributed system to be scheduled among the different processors in such a manner that CPU usage is balanced. Key contributions of this research are:

Using the properties of a single address space and the functionality of Sombrero to allow the threads of a multithreaded activity to be distributed individually. This is not possible in a process-oriented operating system without setting up a separate process for each activity.

Determination of a reasonable distributed scheduling algorithm for the Sombrero single address space environment.

Construction of a simulation of the necessary Sombrero elements e.g., nodes, routers, load tables, and use of it to show that the distributed scheduling algorithm is correct and works in a shared memory environment. Demonstration that the algorithm scales in a linear manner with respect to the number of messages and tables required for load balancing.

Use of a hierarchical clustering mechanism to extend the distributed scheduling algorithm so that it scales well.

Production of functional software for distributed scheduling, inter-node communication and virtual address space partitioning that can be integrated into the Sombrero prototype when it is operational.

To my parents, sister and friends.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Donald Miller, for all his support, advice and feedback. I also thank my Committee members, Dr. Rida Bazzi and Dr. Bruce Millard, for their support. In addition, I would like to thank the research group members, Alan Skousen and John Olson, for their help.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1. Introduction 1

2. Background: Distributed Scheduling 3

2.1 Distributed scheduling overview 3

2.2 Related Work 6

2.2.1 Sprite 7

2.2.2 PVM and Condor 7

2.2.3 UNIX 7

2.3 Distributed scheduling in Sombrero 7

3. Methodology 9

3.1 Testing of Distributed Scheduling in Sombrero 10

3.2 Assumptions 10

3.3 Load Measurement 11

4. Implementation 14

4.1 Inter-node Communication 14

4.1.1 Sombrero Node 15

4.1.2 Sombrero Router 16

4.2 Address Space Allocation 18

4.3 Router Tables 20

4.4 Load Tables 23

4.5 Scheduler 24

4.6 Distributed Scheduler 26

CHAPTER ..... Page

4.6.1 Level Zero (Node level) 27

4.6.2 Level One and Above 28

5. Testing 30

5.1 Assumptions 30

5.2 Single Cluster Tests 31

5.2.1 Special Case 32

5.3 First Level Cluster Tests 33

5.4 Second Level Cluster Tests 35

6. Conclusion and Future Work 39

6.1 Summary of Value of Research 39

6.2 Future Work 40

6.2.1 Sombrero Node 40

6.2.2 Sombrero Router 41

REFERENCES 42

APPENDIX

A. Message Formats 45

A.1 Inter-Node Messages 45

A.2 Windows Messages 50

B. Sombrero Script Files 53

B.1 Sombrero Router Script File 53

B.2 Sombrero Node Script File 54

C. Source Code 56

D. Simulation Test Inputs 58

LIST OF TABLES

Table Page

1. Example: Router 0x1 22

2. Example: Router 0x3 22

3. Single Cluster Testing 31

4. Special Case Testing 33

5. First Level Cluster Testing 34

6. Second Level Cluster Testing 36

LIST OF FIGURES

Figure Page

Figure 1. Load Balancing Algorithms 6

Figure 2. Sombrero nodes communicate with each other through the routers..... 15

Figure 3. Architecture of Sombrero Nodes..... 16

Figure 4. Architecture of a Sombrero router..... 18

Figure 5. Example for Address Allocation 19

Figure 6. Format of the router script file 53

Figure 7. Format of the node script file 54

1. Introduction

Sombrero is a very large single address space operating system that is intended to be extensible to a multiple node system consisting of a set of homogeneous workstations, servers and LANs. In a process-oriented operating system, the process boundary inhibits processes from addressing data or executing code in another processes' virtual address space. However, in a single address space operating system, neither threads nor programs nor the access rights associated with these programs are bound by a reused virtual address space. Only a single address space exists and the virtual addresses have the same meaning anywhere on a local area network. Sombrero is designed to have the appearance of a single large multithreaded process distributed over many nodes. This thesis addresses the issue of thread scheduling over multiple nodes in the system. It will further work on the Sombrero project [1, 8], Alan Skousen's Ph.D. dissertation topic.

Distributed scheduling in Sombrero takes advantage of the distributed single address space features provided by the operating system. This may provide considerable performance gains as opposed to process-oriented systems. In process-oriented systems threads are encapsulated within a process and true protection domain crossing and machine migration without an address space and thread context switch is essentially impossible. Threads are bound in a common process environment including virtual address (VA) space, memory objects, protection domain and resources reachable by the process such as files. In an SASOS, objects are bound permanently to virtual addresses. The thread control block, stack, other private static data, thread ID, and other thread state, e.g., run status and priority, can now retain their virtual addresses irrespective of their physical locations. This means a thread can be suspended on a particular computer node and can be restarted in the same context on some other node. General register contents can be loaded into the CPU general registers on the destination node. Through the caching effect, any missing data can migrate to the thread as well. A SASOS can not only cache data across the network, but it can also migrate execution across the system in a

straightforward manner. The ability of threads to migrate in a simple manner across machines has a potentially far-reaching affect on the performance of the distributed scheduling mechanism.

2. Background: Distributed Scheduling

2.1 Distributed scheduling overview

One of the techniques for scheduling processes of a distributed system is the transparent load-distribution approach. The aim of this technique is to transparently equalize workload amongst the nodes to maximize total system throughput. Transparency can only be achieved by using dynamic load distribution techniques, which operate with absolutely no a priori knowledge of the characteristics and resource requirements of the executing threads. This user transparent activity provides a greater amount of processing power to all users of the system. Any good scheduling algorithm should have the following features [2]:

- **Dynamic in Nature**--The algorithm should be capable of making thread assignment decisions based on the dynamically changing load on various nodes on the system.
- **Quick Decision-Making Capability**--The thread assignment decisions should be made with minimal computational effort possible.
- **Balanced System Performance and Scheduling Overhead**--The algorithm should strive for a balance between the desired system performance and the overhead required while gathering the global state information.
- **Stability**--The situation where the nodes spend most of their time making migration decisions and then migrating the threads should be avoided. Otherwise the algorithm would most likely be counter-productive and radically decrease the performance of the system.
- **Scalability**--The algorithm should adapt to large as well as small networks.
- **Fault-Tolerance**--The failure of nodes in the system should have minimal adverse effects on the performance of the algorithm.

Distributed scheduling techniques for load-distribution are either *global* or *local* based on the information they gather to make the scheduling decisions. Alternatively, they are either

centralized or *distributed* depending on whether the scheduler is located at one master node or if the scheduler is distributed among the processors, respectively [3]. A combination of these techniques produces the following four different strategies:

In global schemes, the state information is collected from all nodes in the system. The state information could be the load for each node or the performance profile of each node.

- **Global Centralized Scheduling Schemes**--A scheduler located on a master node (centralized) performs the computation of load distribution. The state information should be made available to the centralized scheduler so that it can perform the necessary computations. The scheduler then instructs all nodes to distribute the load amongst each other.
- **Global Distributed Scheduling Schemes**--There is a scheduler located on each node to perform the load distribution computations. Hence, the state information should be made available to all the nodes in the system. The nodes then cooperate to distribute the load amongst each other.

In local schemes, the nodes are organized to form groups of particular sizes. In this case the state information exchange and the load distribution is limited within each group. The groups are either static or they can be restructured to form different groups dynamically. The flexibility to change groups dynamically provides the option of inter-group load balancing.

- **Local Centralized Scheduling Schemes**--A scheduler is assigned for each group of size *k* to perform thread-scheduling operations for the group it belongs to. Other *k-1* nodes in the group make the state information available to the group's scheduler.
- **Local Distributed Scheduling Schemes**--Every node in the group has the responsibility of maintaining balanced load in the group.

The centralized approach to scheduling is relies heavily on the performance of the centralized scheduler. If the node on which the scheduler fails then all scheduling in the system

will come to a halt. In addition, this centralized server usually causes a bottleneck for collecting state information and distributing scheduling decisions in the system. In contrast to the centralized scheme, a distributed scheme involves all nodes in the system in making scheduling decisions. Since a centralized server is not solely responsible for taking all scheduling decisions, the scheme becomes fault-tolerant. However, the global scheduling algorithms are inherently more complex. The state information now has to be made available to all nodes in the system. Sombrero is an operating system designed to be extensible to large LANs. On a large LAN, the overhead of distributing state information to all nodes in the system is extremely large. Hence, it seems intuitive that a local distributed scheduling scheme must be adopted to perform thread scheduling for Sombrero.

Depending on the type of node that initiates a global search for a suitable node for the migration of a process, the distributed scheduling schemes are of the following types [2]:

- **Sender-initiated scheme**--The sender node of the process decides where to send the process. In a sender-initiated scheme, heavily loaded nodes search for lightly loaded nodes to which work may be transferred (figure 1). This scheme works well in a system of lightly loaded nodes, but as the number of heavily loaded nodes increases its performance decreases due to the large number of messages being passed around.
- **Receiver-initiated scheme**--The receiver node of the process decides from where to get the process. In a receiver-initiated scheme, lightly loaded nodes search for heavily loaded nodes from which work may be transferred (figure 1). This scheme performs well when the system load is high, but its performance decreases if the number of heavily loaded nodes is low due to the large number of messages being passed around.
- **Dynamic scheme**--At high system loads the receiver-initiated scheme is used while at low system loads the sender initiated scheme is employed.

Because of good performance at both low and high system loads, a dynamic distributed scheduling scheme should be used for Sombrero.

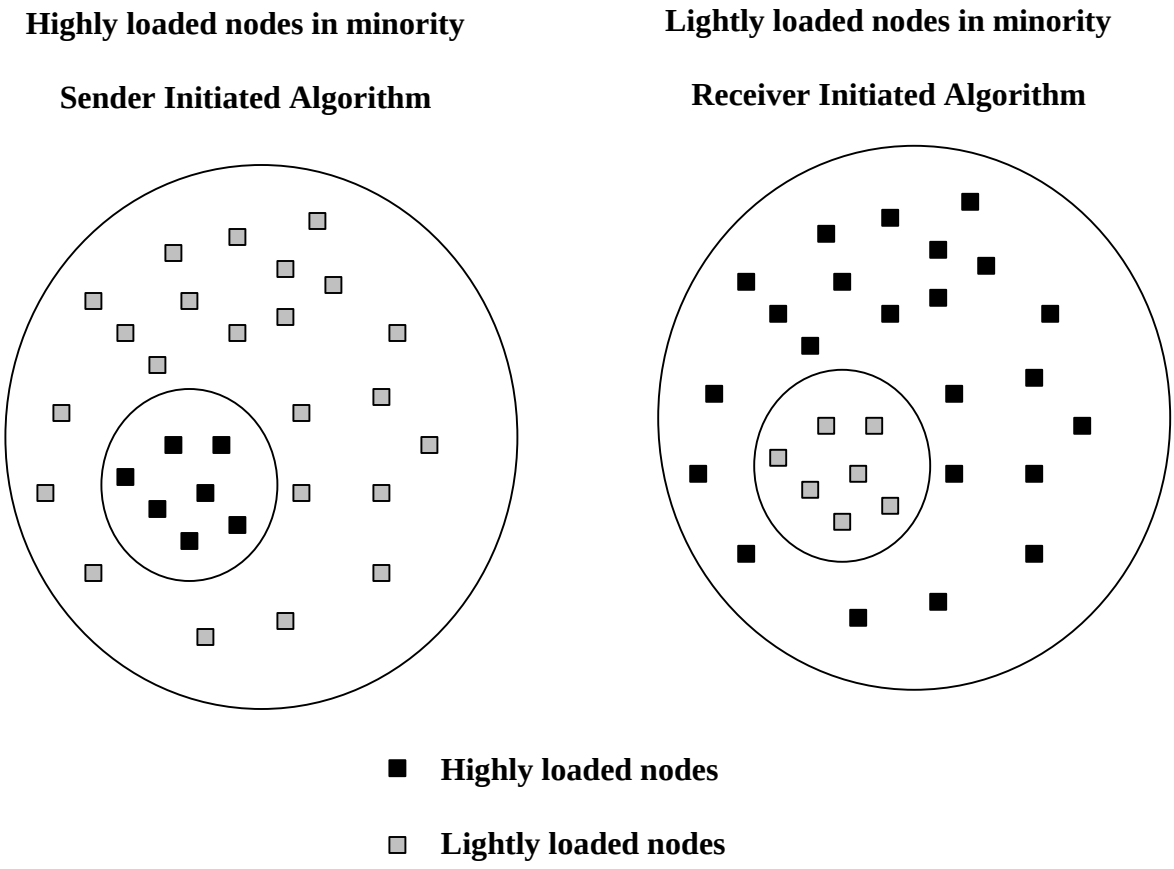


Figure 1. Load Balancing Algorithms

2.2 Related Work

There are a few existing load-balancing algorithms on other parallel execution environments. These include PVM, Condor, Sprite and UNIX. A brief description of these has been given below.

2.2.1 Sprite

The Sprite network operating system was extended to include a load-balancing algorithm at the University of California at Santa Cruz [10]. A centralized sender-initiated load-balancing algorithm was implemented using a threshold policy. The algorithm uses an initial-placement strategy i.e. it starts the processes on nodes with the lowest load instead of migrating a process in the state of execution.

2.2.2 PVM and Condor

This work [11], carried out at the University of Texas at Austin, extended the parallel execution environment of PVM to utilize the currently idle workstations. The approach taken was to integrate into PVM that functionality from Condor [16] which implements process migration and to add a load balancing strategy. In [16], the authors have conducted load-balancing experiments on Condor to compare initial placement and process migration as the load balancing strategy.

2.2.3 UNIX

A load-sharing system called ‘Utopia’ was implemented on a large heterogeneous UNIX network at the University of Toronto [17]. In this system, each user maintains a list of processes that can execute remotely, along with a detailed description of what resources each process requires. Different clusters could have different operating systems provided there is a channel of communication between them.

2.3 Distributed scheduling in Sombrero

In process-oriented systems processes being in different virtual address spaces are, by definition, in different protection domains. Hence inter-process data exchange and synchronization involves cross-domain communication, which leads to considerable overhead (example: - consider remote

procedure call). This is because addresses in a process do not have the same meaning across the process boundaries. Inter-boundary data access and procedure calls require stubs to perform a software communication protocol to check access rights to entry points, perform marshalling/unmarshalling of parameters/results, etc. In contrast, in hardware supported single address space operating systems (SASOS) an ordinary procedure call can be used for inter-domain operations [5]. After the first call, in which authorization is obtained, a hardware-based protection cache called the Range Protection Lookaside Buffer (RPLB) holds the <VA, access rights> pair representing access privileges to the protection domain. The allowed domain crossings can now happen transparently (hardware controlled rather than under software control) and the run-time overhead is the same as that for any intra-program procedure call. These unique features of Sombrero provide the means for a scheduler with components distributed over the entire system. The scheduler components communicate transparently through shared memory for data distribution and entirely avoid using cumbersome IPC and file access protocols required in process-oriented operating systems.

Currently, Sombrero is being prototyped on Alpha 21164 machines running NT 4.0. Sombrero is being developed as a specialized process on NT potentially capable of utilizing the entire available address space (2^{43} bytes). This is done by modifying the NT PALCode to emulate the RPLB. The NT operating system running in Alpha Kernel and Alpha Executive modes in the NT kernel and NT user space respectively provide the basic I/O facilities to Sombrero, which runs in Supervisor mode. The Sombrero prototype uses a pool of NT threads to support its own threads. When any NT thread blocks for I/O on the behalf of a Sombrero thread then some other thread can be scheduled in its place. Using a big enough pool of threads increases the probability that some Sombrero threads will be able to run at any given time. An appropriate algorithm is used to schedule the Sombrero threads onto the pool of NT threads.

3. Methodology

A dynamic local distributed scheduling scheme has been implemented for Sombrero. Making the scheme distributed allows all nodes in the system to take part in the scheduling. This makes the distributed scheduling fault-tolerant. The scheme is local since otherwise distributing the state information of the entire system to all the nodes will occur at the cost of a heavy overhead and not scale well. The entire system is divided into scheduling clusters to limit the distribution of the state information. Clusters help to achieve a balance between the desired system performance and the overhead required to gather the state information.

LANs can be used to organize the Sombrero nodes into clusters. A large sized LAN is usually made up of a number of small LANs depending upon the division of an organization into different subgroups. These LANs can be viewed, as the logical clusters required for efficient distributed scheduling. The clusters have a hierarchical organization in which a cluster can be made up of several other clusters. Scheduling of threads takes place amongst nodes within a particular cluster or between clusters which form a bigger cluster.

A mechanism is needed to distribute load information of the nodes in the system. The shared memory inherent to a distributed SASOS provides an excellent mechanism for this purpose. Shared memory is used to set up a data structure per cluster for storing load information of the cluster members. The data communicated amongst the members of a cluster is of two types:

- Load information
- Thread related data (e.g.:- thread control blocks)

The token tracking mechanism [6] in Sombrero allows the Sombrero programs to be unaware of the physical location of the virtual addresses (VA). Token tracking uses a communication mechanism based on virtual addresses instead of IP addresses. Hence, the Sombrero nodes are designed to communicate with each other through router like services. These

routers use tables to look up the IP address corresponding to a VA. Each first level Sombrero cluster has a router designated to it. This router uses a table to store VA to IP mappings for the nodes in its cluster. A router also maintains consolidated tables of VA to IP mappings for all other first level clusters.

3.1 Testing of Distributed Scheduling in Sombrero

The distributed scheduling mechanism for Sombrero has been tested to verify its correctness, stability and scalability. The testing has been carried out using a simulator. The Sombrero router software that has been developed for use in the actual Sombrero prototype is employed for the testing. All message communication between Sombrero nodes occurs through these routers. The routers are also used to measure the number of messages required for load balancing during testing. The testing is done for different combinations of initial loads on the nodes at three levels of clusters. Since the simulation uses semaphores provided by NT, the testing has to be carried out on a single NT machine.

Script files are used to initialize the nodes at startup. In the simulation, each node is initialized with a particular number of Sombrero threads through the script file. The script files also have appropriate load table addresses for the nodes to subscribe to. The simulation is then allowed to run and the number of messages required for load balancing is measured. It is verified that the number of messages required to achieve load balancing, as the number of heavily loaded nodes increases, does not rise exponentially.

3.2 Assumptions

Initially, two single-CPU Alpha 21164 NT workstations were to be used for the project. A working prototype of Sombrero is needed that gives NT the ability to manage 8-terabyte data sets

across a network in a single distributed virtual address space. This distributed address space should support a mechanism that allows multiple network users to have access to and have a consistent view of the same addresses while at the same time being protected from one another's activities.

The prototype of Sombrero, which demonstrates the single address space properties, was not completely implemented in time to meet the scheduled deadlines for this thesis. The distributed scheduling mechanism being proposed needed a functional prototype which supports essential features such as protections domains, Sombrero thread support, token tracking support, etc. Hence, the scheduling mechanism is simulated on NT supported Intel machines. The file mapping objects provided by NT have been used to support shared memory needed for the distributed scheduling.

3.3 Load Measurement

A load-balancing algorithm requires a suitable method to measure the workload on the nodes in the system. A node's workload can be estimated based on some measurable parameters. The Sombrero prototype works on a network of homogeneous nodes. This means the architecture of all node processors is the same. We also make the simplifying assumption that the processing speed of all nodes is also identical. With these assumptions, the parameters for load estimation could include factors such as the following [2]:

- Total number of threads on the node at the time of load measurement
- Instruction mixes of these threads (I/O bound or CPU bound)

Taking these factors into consideration the workload on a particular node can be calculated as:

$$\text{Work Load} = \Sigma_i (p_i \times f_i)$$

$p \Rightarrow$ processor utilization of a thread and,

$f \Rightarrow$ heuristic factor (adjusts the importance of thread depending on how it is being used)

The above formula takes into account the number of threads as well as the instruction mixes of these threads. It ensures that the workload on a node would be considered high if there are some CPU intensive threads present. The workload is also considered high in the case of a large number of I/O intensive threads.

The distributed scheduler has been constructed as a simulator on a single machine supporting NT. In addition, the instruction mixes of the Sombrero threads are not determinable in the simulation. Due to these restrictions, the workload cannot be measured using the given formula. For the purposes of this simulation, the workload has been defined as the number of Sombrero threads running on a particular node. The algorithm has been designed such that it will work correctly with the above-mentioned more general definition of workload for the actual implementation of the Sombrero prototype.

NT provides the `RegQueryValueEx()` API call to find the processor utilization of a particular thread. The ‘Processor Time’ counter for each ‘Thread’ object gives the processor utilization of threads in NT. Processor Time is the percentage of elapsed time for which a thread used the processor to execute instructions. An instruction is the basic unit of execution in a processor, and a thread is the object that executes instructions. The heuristic factor ‘f’ should have a large value for I/O intensive threads and a small value for CPU intensive threads. The

values of this factor can be empirically determined by using a fully functional Sombrero prototype.

4. Implementation

The Microsoft Development Studio and the Microsoft Visual C++ v5.0 development system were used to develop the distributed scheduling mechanism. The code is written using the Win32 API. The important parts of the project are described next.

4.1 Inter-node Communication

Distributed scheduling requires a reliable inter-node communication amongst the nodes of the system. The Sombrero system is organized into hierarchies of clusters for managing distributed scheduling. The first level cluster consists of nodes and a router is assigned to each cluster. A group of clusters forms a higher level cluster. The token tracking mechanism allows the nodes to be unaware of each other's IP addresses. The routers maintain router tables to store the VA to IP mappings for the nodes. They also manage consolidated router tables for VA to IP mappings of other routers. All the messages exchanged between any of the nodes are redirected through the routers. It’s the router's responsibility to find the IP address of the node depending upon the VA inserted as data in the message package. A router forwards the message to an appropriate router if the target VA is not listed in its own router table of VA to IP mappings.

Figure 1 shows two clusters. A message from the node of Root Memory Object Control Block (RMOCB) 0x1000 directed to the MOCB 0x2000 (message A) goes through the router 0x1 as shown. The router 0x1 has the VA to IP mapping of both these nodes. A message from the node of RMOCB 0x1000 directed to the MOCB 0x3000 (message B) goes through the router 0x11. The router 0x11 stores the VA to IP mapping of the node 0x3000. In addition, both the routers have each other's consolidated router tables.

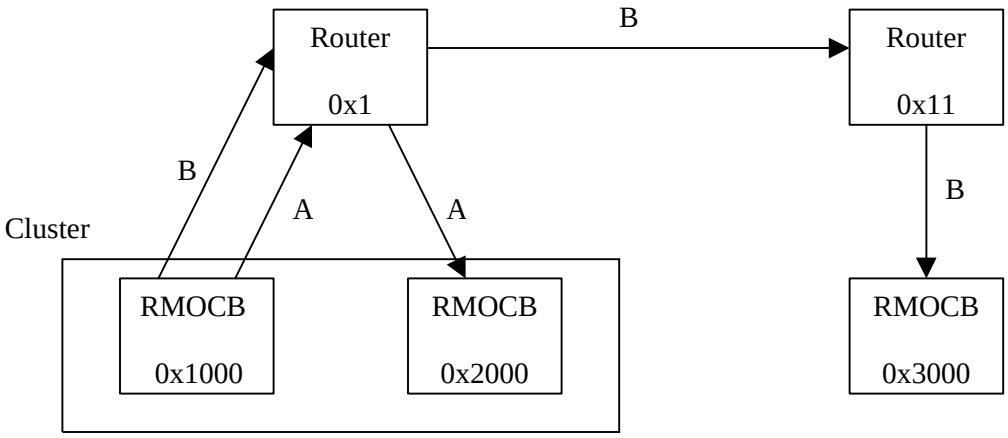


Figure 2. Sombrero nodes communicate with each other through the routers.

4.1.1 Sombrero Node

In the simulation, the only function of the Sombrero nodes (see figure 2) is to demonstrate load balancing. They use Windows message-based notification of network events for a socket. The Windows Sockets `WSAAsyncSelect()` API function is used to achieve this. A node sets up a socket to communicate with the router assigned to its cluster. The Sombrero node is a specialized Windows application. The main window of the node will receive a UWM_READ message when there is data that can be read from this socket. This will happen when the router sends a message to the node. The UWM_READ messages are handled in the `BOOL CMainFrame::PreTranslateMessage(MSG* pMsg)` function (MainFrm.cpp[1]). This function handles the preprocessing of all the windows messages. The purpose and structure of each message sent between the Sombrero nodes is explained in Appendix A.

The main window is also responsible for sending messages on behalf of the Sombrero node. A Windows message will be sent to the main window with the appropriate data and

[1] All .cpp and .h files can be found in the source code given in Appendix C.

destination VA. The main window then forwards the message to the router. The purpose and structure of each Windows message sent to the main window of the application is explained in Appendix A.

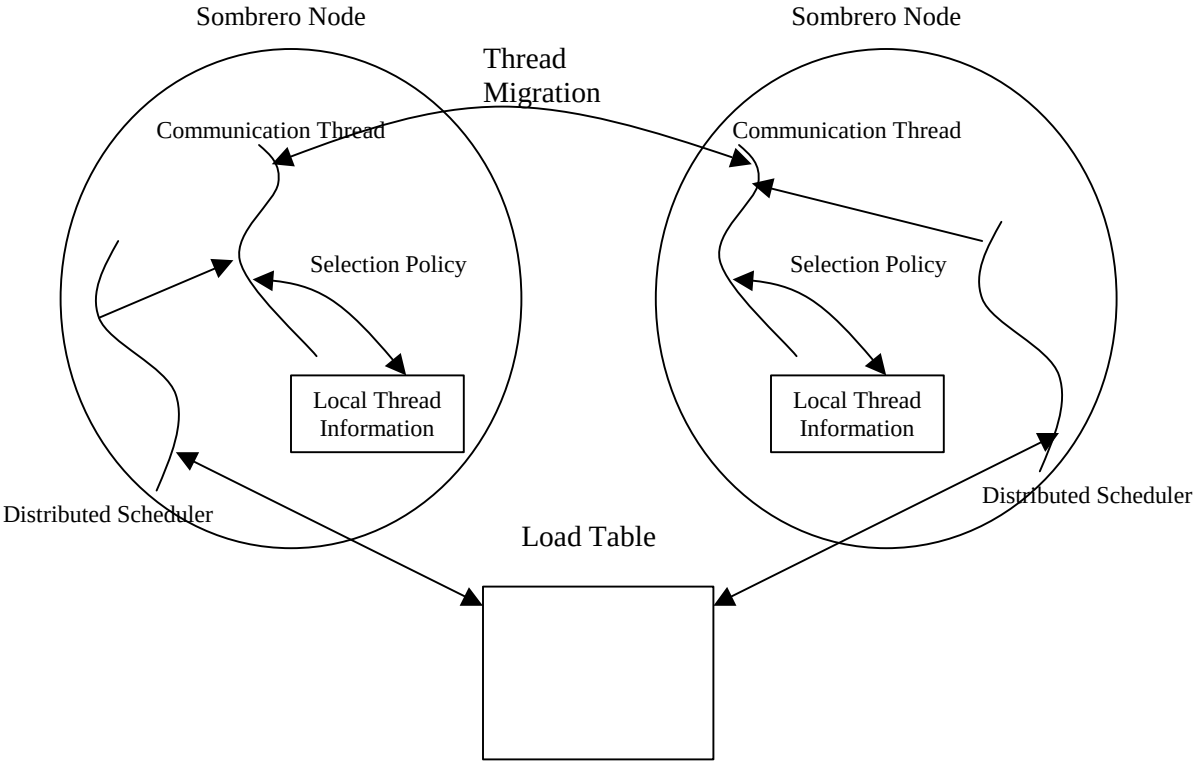


Figure 3. Architecture of Sombrero Nodes

4.1.2 Sombrero Router

A Sombrero router's main job is to direct messages sent from a node in the system to the destination node using VA to IP mapping tables. Each router has a well-known port on which the router listens for NEWNODE, NEWROUTER messages. As part of its setup, a node sends a NEWNODE message to its router. On receiving this message, the router makes the VA to IP

mapping for that node. Similar initializations take place as part of a new router's setup. The NEWROUTER message is broadcast to all existing routers. A script file provides the necessary information about all existing routers. The structure of the script file is discussed in Appendix B. All even virtual addresses can be used to name a MOCB. Hence, every Sombrero router is identified with a unique odd VA.

In an effort to make the Sombrero router scalable, I/O completion ports provided by NT [14] have been used for the communication involving sockets. All the sockets created by the router are associated with an I/O completion port. Whenever a message is sent to any of these sockets, an I/O completion request is queued to the port. A pool of threads is created to service the packets queued at an I/O completion port. This architecture prevents the server from having to create one thread per client (where the client could be a Sombrero node or another router). This property of the router makes the system extremely scalable. A large number of clients can be easily handled by the router without wasting valuable system resources. At the same time having multiple threads for servicing the requests allows the router to handle several requests at a time. This increases the parallelism of the router.

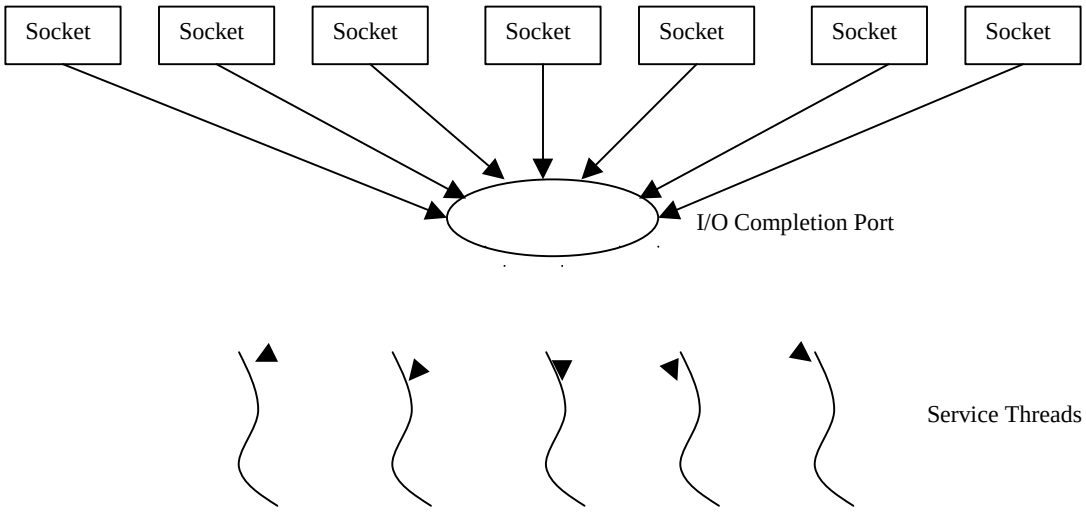


Figure 4. Architecture of a Sombrero router

4.2 Address Space Allocation

This project implements an address space allocation mechanism to distribute the 2^{64} bytes address space amongst the nodes in the system. This address space will be used to allocate its memory objects. As part of its startup initialization, a Sombrero node contacts its router with a request for an address space. The router keeps record of a default parent node that can grant an address space upon request. This node will continue receiving requests as long as its address space does not fall below a certain lower limit. When the address space of the node falls below the lower threshold, the router looks for an alternative node in its cluster. Any node is eligible provided its address space is greater than the threshold. If such a node is not available then the address space request is randomly forwarded to some other router. Consider the case when a default parent node is not selected by the router. Now when an address space is granted to a particular node, the router makes it the default parent node.

Example

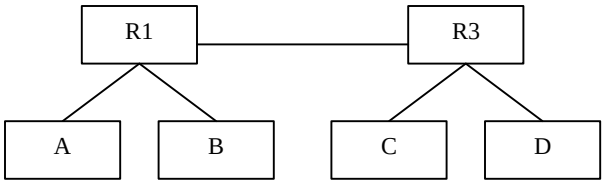


Figure 5. Example for Address Allocation

Consider a system of four Sombrero nodes: A, B, C and D (figure 5). The four nodes are divided into two clusters equally and a router is assigned to each of those clusters (router identification addresses: 1 and 3). These two clusters could form a bigger cluster. The order of initialization of the Sombrero nodes is A, B, C and D in that order. The address space allocated for the nodes when A is initialized will be:

A: 0x0000000000000000 – 0xffffffffffffffff

The address space allocated for the nodes when B is initialized will be:

A: 0x0000000000000000 – 0x7fffffffffffffff

B: 0x8000000000000000 – 0xffffffffffffffff

The address space allocated for the nodes when C is initialized will be:

A: 0x0000000000000000 – 0x3fffffffffffffff

B: 0x8000000000000000 – 0xffffffffffffffff

C: 0x4000000000000000 – 0x7fffffffffffffff

The node C belongs to a new cluster. The address space for C will be obtained by fissioning the address space allocated to any one the nodes A or B. In the above example the node A has been requested to supply the new address space.

The address space allocated for the nodes when D is initialized will be:

A: 0x0000000000000000 – 0x3fffffffffffffff

B: 0x8000000000000000 – 0xffffffffffffffff

C: 0x4000000000000000 – 0x5fffffffffffffff

D: 0x6000000000000000 – 0x7fffffffffffffff

The node D belongs to the same cluster as node C. Hence, the request for the new address space is forwarded to node C.

4.3 Router Tables

The routers use the following structures to store the VA to IP address mappings of the Sombrero nodes in its cluster and the nodes in other clusters.

```
//Structure to store information about remote nodes
struct Remote {
      UINT64 rmocb;
      UINT64 mask;
};
//Structure to store information about local nodes and other
//routers
struct RouteTable {
      UINT64 rmocb;          //even: node, odd: router or tempID
      UINT64 mask;           //only a node has a mask
      struct Remote *remote; //router
      char hostName[25];
      UINT port;
      SOCKET sock;
};
```

An entry is made in the router table on receiving a NEWNODE or a NEWROUTER message. The table entry is then initialized with the appropriate values sent along with the message. The socket corresponding to the node or router is associated with the I/O completion

port. The mask for a router table entry is calculated using the RMOCB and the size of virtual address space allocated to a particular node. The formula for calculating the mask is given below:

```
mask = ~(rmocb ^ (rmocb + range));
```

The router makes a table entry for each of the nodes in its cluster. For nodes belonging to other clusters, a consolidated remote table `(struct Remote *remote)` is maintained. The router maintains a single entry in a remote table for all those nodes that have been allocated address space from a contiguous block. It makes a new entry if the address space allocated to a particular node is discontinuous. The router uses the function `RMOCBTOSocket()` to find the table entry corresponding to a VA (mserver.cpp). This function uses the mask value stored along with the RMOCB value to achieve the VA to IP address mapping.

Example

Consider the system of four nodes A, B, C and D described in section 4.3. The router tables of the two routers are given below. Given a VA we can find out the IP address by using the values given in the columns RMOCB and MASK. A ‘NULL’ value in the column REMOTE indicates that the table entry corresponds to a Sombrero node. If the table entry is for a Sombrero cluster, then the REMOTE column gives a pointer to another table of <RMOCB, MASK> pairs corresponding to that cluster.

Table 1. Example: Router 0x1

RMOCB	MASK	REMOTE	IP ADDRESS
0	0xC000000000000000	NULL	…
0x8000000000000000	0x8000000000000000	NULL	…
0x3	xx		…

RMOCB	MASK
0x4000000000000000	0xC000000000000000
1	xx

Table 2. Example: Router 0x3

RMOCB	MASK	REMOTE	IP ADDRESS
0x4000000000000000	0xE000000000000000	NULL	…
0x6000000000000000	0xE000000000000000	NULL	…
0x1	xx		…

RMOCB	MASK
0	0xC000000000000000
0x8000000000000000	0x8000000000000000
1	xx

4.4 Load Tables

For each Sombrero cluster, a table is maintained to store the load information of all the cluster members. Since distributed-shared memory is used to allocate load tables, the information is available to all the cluster members by default. Token tracking mechanism is used for consistency management of the distributed-shared memory in Sombrero. For the purposes of the simulation on NT, mutexes are used for consistency management of the shared memory. Shared memory is not available across multiple machines in NT. Hence, the members of a particular cluster should be on the same physical machine for the simulation. Thresholds of load have been established to minimize the exchange of load information in the network. Only threshold crossings will be recorded in the load tables. By increasing the granularity of load information, the frequency of changes to the load table is reduced.

```
//Structure for storing load information of a cluster
struct LoadData {
      UINT nNodeCount;           //count of cluster members
      UINT nAlgorithm;           //sender or receiver initiated
      UINT nHighCount;           //highly loaded members
      UINT nLowCount;            //lightly loaded members
      UINT64 nMOCB[MAXNODES];    //members
      UINT nLoadType[MAXNODES];  //corresponding load
      UINT nLock[MAXNODES];      //locks for the members
};
```

```
//Structure for storing information related to load tables
struct LoadTablePtr {
      UINT64 nMOCB;               //cluster name
      struct LoadData *pLoadData; //pointer to load table
      UINT nIndex;                //index in the load table
      struct CopySet cs;          //copyset data
      HANDLE hCSMutex;            //copyset mutex
      HANDLE hLDMutex[MAXNODES];  //mutexes for load table entries
};
```

In the simulation the load tables are mapped and initialized after the address space is allocated to the node. Information about the load tables is obtained from a script file. The structure of the script file is discussed in Appendix B. All members of a particular cluster have to be included in the copy set graph of the load table's MOCB. If the load table VA is in a node's allocated address space then that node is the copy set owner of the load table. Cluster members have to send a request of copy set insertion to the copy set owner.

```
//Structure used for a copy set graph
struct CopySet {
      UINT64 nParent;             //parent node in the copy set
      UINT64 nChild[ARITY];       //children nodes
      UINT nCount;                //count used to populate graph
};
```

4.5 Scheduler

A simple round-robin algorithm has been implemented for the simulation to take care of the local scheduling needs of a Sombrero node. The algorithm uses multilevel priority queues for scheduling the Sombrero threads. The algorithm is designed such that it can be incorporated in the actual Sombrero prototype with minimum efforts. It is assumed that the prototype will allocate a pool of NT threads to run the Sombrero threads. The NT threads are then divided into

three subgroups, one for each priority queue. Sombrero threads from a priority queue are scheduled on the corresponding pool of NT threads. This simulation doesn't execute actual Sombrero threads. The scheduling is demonstrated by displaying the scheduling decisions while the algorithm executes. The scheduler just maps a Sombrero thread structure to an NT thread and then sleeps for a small amount of time to simulate the quantum of execution.

A Sombrero thread control block (TCB) has the structure given below. This class can be used in the Sombrero prototype after adding members representing the thread context data to it.

```
//Sombrero TCB
class CTPDCB//: public CPDCB
{
public:
	//Temporary variable only being used on x86 machines
	UINT64 tno;
	//All thread context data here
	//The quantum value for this Sombrero thread
	UINT64 Quantum;
	UINT64 Priority; //Sombrero thread priority [0-2]
};
```

The scheduling queue for Sombrero threads of a particular priority has the following structure:

```
//Sombrero priority queue head
class CSchedQueue
{
      //memory block for allocating queue links
      CQueueBLK allocateQ;
      p64LINK Head;              //pointer to head of queue
      p64LINK Tail;              //pointer to tail of queue
public:
      static   void   Initialize(p64CSchedQueue   Xthis,   p64VOID
      pvBase, UINT64 size);
      static void Insert(p64CSchedQueue Xthis, p64CTPDCB tcb);
      static p64CTPDCB Remove(p64CSchedQueue Xthis);
      static bool IsEmpty(p64CSchedQueue Xthis);
};
//Link in a Sombrero priority queue
struct SchedLink {
      p64CTPDCB TCB;        //Sombrero thread
      p64LINK Next;         //pointer to next thread
};
```

4.6 Distributed Scheduler

The distributed scheduler in Sombrero uses the load balancing technique to schedule threads across multiple nodes in the distributed system. A separate thread daemon is initiated in the Sombrero prototype for distributed scheduling. The system has been organized into a hierarchical structure of clusters. The nodes in the system are at the lowest level of the organization. This makes the scheduling algorithm scalable. Load information exchange is thus limited between members of a cluster, which themselves may be clusters or nodes. Load tables are allocated to

maintain load information about members of the cluster. Distributed scheduling primarily occurs at the node level. As the load on clusters at lower levels increases, the distributed scheduling starts occurring at higher levels. The mechanism for scheduling at levels one and above (between clusters) is identical. However, at the lowest level (between nodes) slightly different rules are applied. The two different mechanisms are explained below.

4.6.1 Level Zero (Node level)

The measure of load at the node level is the number of Sombrero threads. There are three thresholds of load at this level viz., high, medium and low. Two variables HIGHLOAD and MEDIUMLOAD (dsched.h) are used to define the thresholds. The three thresholds are defined as follows:

- high--number of Sombrero threads ≥ HIGHLOAD
- low--number of Sombrero threads < MEDIUMLOAD
- medium--number of Sombrero threads < HIGHLOAD and number of Sombrero threads ≥ MEDIUMLOAD

For the simulation, the values of the two variables HIGHLOAD and MEDIUMLOAD are chosen as some arbitrary constants. When the Sombrero prototype is operational, these values can be determined empirically by carrying out tests on the actual prototype. Another mechanism that might be implemented at that time is an adaptive algorithm that selects the values as the load on the system varies.

The algorithm used to implement load balancing is dynamic i.e. at any given time only the minority number (smaller group of the group of highly loaded nodes or the group of lightly loaded nodes) of nodes is active. This means that at lower loads the algorithm is sender initiated and at higher loads, the algorithm is receiver initiated. At any given system load, care is taken to

ensure that a minimal required number of nodes are involved in the thread exchanges. As far as possible the distributed scheduler is operating at this level to achieve load balancing. The distributed scheduler gets involved in thread migration at a higher level only if it is not possible at a lower level. The steps followed by the algorithm are:

1. If this node is loaded in the medium range or if there is only one node in the cluster then sleep for some time.
2. If node belongs to the majority group (larger group of the group of highly loaded nodes or the group of lightly loaded nodes) then distributed scheduling is not to be done.
3. If node belongs to the minority group then distributed scheduling is to be done. This condition is true in the following two cases:
   - The node is heavily loaded and the algorithm is sender initiated. In this case, a lightly loaded node is chosen at random and the RGETTHREADS (appendix A) message protocol is followed for thread migration.
   - The node is lightly loaded and the algorithm is receiver initiated. In this case, a highly loaded node is chosen at random and the GETTHREADS (appendix A) message protocol is followed for thread migration.

4.6.2 Level One and Above

Consider the case when there are no lightly loaded nodes in a cluster. No load balancing can now occur at the node level. The heavily loaded nodes in the cluster cannot share their load with any other node in that cluster. In this case, the nodes participate in thread exchanges with nodes of other clusters. At any level above the nodes the three thresholds are defined as given:

- high: - no cluster members are lightly loaded and at least one member is highly loaded
- low: - no cluster members are highly loaded and at least one member is lightly loaded

- medium: - all other cases of loads where load balancing can occur within the cluster members or when all members of the cluster are medium loaded

Load balancing is only intermittently attempted at higher levels of clusters. A constant MULTIPLE (dsched.h) is defined to compute the frequency of load balancing. Load balancing at an n^{th} level cluster will be attempted every (n×MULTIPLE) times the number of unsuccessful attempts at the node level. At any level of cluster only the nodes belonging to the minority group at that level will be active. When such a node becomes active, the corresponding first level cluster is locked so that no other node in that cluster will initiate a load balancing operation on its own. A suitable n^{th} level target cluster is found through the corresponding load table and a request for thread migration (TRANSFERREQUEST message protocol) is sent to the target. The target node will check whether the first level cluster it belongs to is a suitable one. If the cluster is suitable it is locked and an acknowledgement (TRANSFERACK message protocol) is sent to the initiating node. In addition, a DOTRANSFER message is broadcast to the copy set of the target load table. The initiating and target clusters now exchange appropriate number of threads.

5. Testing

The distributed scheduling algorithm has been tested to verify its correctness, stability and scalability. The testing has been conducted for three levels of Sombrero clusters. Script files (Appendix B) were used to initialize the Sombrero nodes and routers. The script files start the Sombrero routers and nodes with an appropriate setup of load tables and initial load values (number of Sombrero threads). The Sombrero routers then measure the number of messages required to achieve load balancing within the Sombrero nodes. This procedure is repeated for all the important cases of load values at each of the three cluster levels.

5.1 Assumptions

The testing of the distributed scheduling algorithm is done with the following assumptions:

1. The actual Sombrero prototype will use the token tracking mechanism [4, 6] to provide shared memory required by the distributed scheduling algorithm. This mechanism will itself use messages to maintain the consistency of the shared memory. As the token tracking mechanism has not yet been implemented, the simulation uses memory-mapped files provided by NT to support shared memory. Hence, the messages that will be required for the consistency of shared memory cannot be measured in this testing. It is clear though that as the level of clustering increases the shared memory accesses and thus the number of messages required will increase proportionally.
2. For the purposes of simulation, the following values are used to define the load thresholds:
   - high--number of Sombrero threads in the ready queue ≥ 50
   - low--number of Sombrero threads in the ready queue < 30
   - medium--number of Sombrero threads in the ready queue < 50 and number of Sombrero threads in the ready queue ≥ 30

3. For all cases of testing, the load values are chosen so that the lightly loaded nodes can accept the entire extra load from the heavily loaded nodes. For example, consider the case that there is only one lightly loaded node present in a cluster and all the other nodes are heavily loaded. In this case the loads will be chosen so that all the heavily loaded nodes can be brought down to medium loads through load balancing with the single lightly loaded node.
4. A particular cluster is represented as the triplet of the following three values:- [no. of highly loaded nodes, no. of medium loaded nodes, no. of lightly loaded nodes]

5.2 Single Cluster Tests

The single cluster tests were carried out on a cluster of eight Sombrero nodes. The number of heavily loaded nodes is gradually increased from a single node to seven nodes.

Table 3. Single Cluster Testing

Cluster (Before Load Balancing)	**Cluster (After Load Balancing)**	**Messages**
[1,0,7]	[0,1,7]	3
[2,0,6]	[0,2,6]	6
[3,0,5]	[0,3,5]	9
[4,0,4]	[0,4,4]	12
[5,0,3]	[0,5,3]	10
[6,0,2]	[0,6,2]	12
[7,0,1]	[0,7,1]	14

Notice that at the most three messages are required to transfer the load from a highly loaded node to a lightly loaded node in the single cluster tests (first row in table 3). These three messages are:

1. RGETTHREADS
2. GETTHREADS
3. SENDTHREADS

5.2.1 Special Case

Testing was also carried out for the special case of a single extremely highly loaded node that distributes its load to lightly loaded nodes. This case is an exception to the assumption that a heavily loaded node can be brought down to medium load by load distribution. It is demonstrated in the simulation by setting up an extremely heavily loaded Sombrero node and a number of lightly loaded nodes. After the cluster reaches a stage when no lightly loaded nodes are available for load balancing, some random number of Sombrero thread control blocks are periodically deleted from the medium loaded nodes. This leads to some of the medium loaded nodes becoming lightly loaded so that the load balancing restarts with the new load configuration. The heavily loaded node keeps sending threads to lightly loaded nodes, as they become available. In the table below, the column 'Event' gives a situation that will allow further load balancing to occur within the nodes.

Table 4. Special Case Testing

Event	Cluster (Before Load Balancing)	Cluster (After Load Balancing)	Messages
Simulation begins.	[1,0,7]	[1,7,0]	21
Some threads are removed from a few nodes.	[1,3,4]	[1,7,0]	12
Some threads are removed from a few nodes.	[1,5,2]	[1,7,0]	6
Some threads are removed from all nodes.	[1,0,7]	[1,7,0]	21

5.3 First Level Cluster Tests

The first-level cluster tests were carried out for three clusters each having eight Sombrero nodes. The load on a cluster is defined as high, medium and low as given in section 4.6.2. There are ten possible combinations of loads on three clusters. The test values have been chosen so that the algorithm was verified for all these ten combinations of cluster loads. The column ‘Load’ gives the load configuration of the clusters involved in the testing.

Table 5. First Level Cluster Testing

Load	Before Load Balancing			After Load Balancing			Messages
	Cluster I	**Cluster II**	**Cluster III**	**Cluster I**	**Cluster II**	**Cluster III**	
{L,L,L}	[0,0,8]	[0,0,8]	[0,0,8]	[0,0,8]	[0,0,8]	[0,0,8]	-
	[2,0,6]	[2,0,6]	[2,0,6]	[0,2,6]	[0,2,6]	[0,2,6]	18
{L,L,M}	[2,0,6]	[2,0,6]	[0,8,0]	[0,2,6]	[0,2,6]	[0,8,0]	12
{L,L,H}	[0,0,8]	[0,0,8]	[1,7,0]	[0,0,8]	[0,0,8]	[0,8,0]	5
	[0,0,8]	[0,0,8]	[2,6,0]	[0,0,8]	[0,0,8]	[0,8,0]	8
	[0,0,8]	[0,0,8]	[3,5,0]	[0,0,8]	[0,0,8]	[0,8,0]	11
	[0,0,8]	[0,0,8]	[4,4,0]	[0,0,8]	[0,0,8]	[0,8,0]	14
	[0,0,8]	[0,0,8]	[5,3,0]	[0,0,8]	[0,0,8]	[0,8,0]	17
	[0,0,8]	[0,0,8]	[6,2,0]	[0,0,8]	[0,0,8]	[0,8,0]	20
	[0,0,8]	[0,0,8]	[7,1,0]	[0,0,8]	[0,0,8]	[0,8,0]	23
	[0,0,8]	[0,0,8]	[8,0,0]	[0,0,8]	[0,0,8]	[0,8,0]	26
{L,M,M}	[0,0,8]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	-
	[2,0,6]	[0,8,0]	[0,8,0]	[0,2,6]	[0,8,0]	[0,8,0]	6
{L,M,H}	[0,0,8]	[0,8,0]	[1,7,0]	[0,0,8]	[0,8,0]	[0,8,0]	5
	[0,0,8]	[0,8,0]	[2,6,0]	[0,0,8]	[0,8,0]	[0,8,0]	8
	[0,0,8]	[0,8,0]	[3,5,0]	[0,0,8]	[0,8,0]	[0,8,0]	11
	[0,0,8]	[0,8,0]	[4,4,0]	[0,0,8]	[0,8,0]	[0,8,0]	14
	[0,0,8]	[0,8,0]	[5,3,0]	[0,0,8]	[0,8,0]	[0,8,0]	17
	[0,0,8]	[0,8,0]	[6,2,0]	[0,0,8]	[0,8,0]	[0,8,0]	20

Table 5. First Level Cluster Testing (contd.)

Load	Before Load Balancing			After Load Balancing			Messages
	Cluster I	Cluster II	Cluster III	Cluster I	Cluster II	Cluster III	
{L,M,H}	[0,0,8]	[0,8,0]	[7,1,0]	[0,0,8]	[0,8,0]	[0,8,0]	23
	[0,0,8]	[0,8,0]	[8,0,0]	[0,0,8]	[0,8,0]	[0,8,0]	26
{L,H,H}	[0,0,8]	[8,0,0]	[1,7,0]	[0,0,8]	[0,8,0]	[0,8,0]	31
	[0,0,8]	[8,0,0]	[2,6,0]	[0,0,8]	[0,8,0]	[0,8,0]	34
	[0,0,8]	[8,0,0]	[3,5,0]	[0,0,8]	[0,8,0]	[0,8,0]	37
	[0,0,8]	[8,0,0]	[4,4,0]	[0,0,8]	[0,8,0]	[0,8,0]	40
	[0,0,8]	[8,0,0]	[5,3,0]	[0,0,8]	[0,8,0]	[0,8,0]	43
	[0,0,8]	[8,0,0]	[6,2,0]	[0,0,8]	[0,8,0]	[0,8,0]	46
	[0,0,8]	[8,0,0]	[7,1,0]	[0,0,8]	[0,8,0]	[0,8,0]	49
	[0,0,8]	[8,0,0]	[8,0,0]	[0,0,8]	[0,8,0]	[0,8,0]	52
{M,M,M}	[0,8,0]	[0,8,0]	[0,8,0]	[0,8,0]	[0,8,0]	[0,8,0]	-
{M,M,H}	[0,8,0]	[0,8,0]	[8,0,0]	[0,8,0]	[0,8,0]	[8,0,0]	-
{M,H,H}	[0,8,0]	[8,0,0]	[8,0,0]	[0,8,0]	[8,0,0]	[8,0,0]	-
{H,H,H}	[8,0,0]	[8,0,0]	[8,0,0]	[8,0,0]	[8,0,0]	[8,0,0]	-

5.4 Second Level Cluster Tests

In Sombrero, the clusters are named from bottom-up. This means a cluster containing just nodes is known as a first level cluster, a cluster containing first level clusters is known as a second level cluster and so on. The second level cluster tests were carried out with six first level clusters each having eight Sombrero nodes. Two second level clusters (viz., cluster A and cluster B) were

organized each having three first level clusters (clusters I, II and II form cluster A) as the cluster members. Cluster A is heavily loaded where as cluster B is lightly loaded. The testing starts with one node in the cluster A being heavily loaded. The number of heavily loaded nodes is then increased gradually until all nodes in the cluster A are heavily loaded.

Table 6. Second Level Cluster Testing

Cluster A			**Cluster B**			**Messages**	
I	**II**	**III**	**IV**	**V**	**VI**		
[1,7,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	5	**Before**
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		**After**
[2,6,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	8	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[3,5,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	11	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[4,4,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	14	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[5,3,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	17	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[6,2,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	20	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[7,1,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	23	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	26	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[1,7,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	31	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[2,6,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	34	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[3,5,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	37	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[4,4,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	40	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[5,3,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	43	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[6,2,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	46	

Table 6. Second Level Cluster Testing (contd.)

Cluster A			Cluster B			Messages	
I	**II**	**III**	**IV**	**V**	**VI**		
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[7,1,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	49	**Before**
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		**After**
[8,0,0]	[8,0,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]	52	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[1,7,0]	[0,0,8]	[0,8,0]	[0,8,0]	57	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[2,6,0]	[0,0,8]	[0,8,0]	[0,8,0]	60	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[3,5,0]	[0,0,8]	[0,8,0]	[0,8,0]	63	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[4,4,0]	[0,0,8]	[0,8,0]	[0,8,0]	69	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[5,3,0]	[0,0,8]	[0,8,0]	[0,8,0]	72	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[6,2,0]	[0,0,8]	[0,8,0]	[0,8,0]	75	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[7,1,0]	[0,0,8]	[0,8,0]	[0,8,0]	78	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		
[8,0,0]	[8,0,0]	[8,0,0]	[0,0,8]	[0,8,0]	[0,8,0]	81	
[0,8,0]	[0,8,0]	[0,8,0]	[0,0,8]	[0,8,0]	[0,8,0]		

Notice that at the most five messages are required to transfer the load from a highly loaded node to a lightly loaded node in the second level cluster tests (first row in table 6). These three messages are:

1. TRANSFERREQUEST
2. TRANFERACK
3. RGETTHREADS
4. GETTHREADS

5. SENDTHREADS

It is observed that the increase in number of messages is proportional to the increase in number of heavily loaded nodes. The number of messages required for load balancing at the second and third levels is the same if the ratio of heavily and lightly loaded nodes is kept constant at both levels. This is true provided the messages required for the access and consistency management of shared memory are not taken into consideration. Only one additional load table is required per additional level of clustering. Hence, the required number of messages is expected to increase by a small constant factor as the level of clustering increases. It can therefore be safely concluded that the algorithm is scalable and will be stable as the number of nodes in the system increases.

6. Conclusion and Future Work

The testing of distributed scheduling using the simulator verifies that the algorithm functions correctly. The algorithm is tested for three levels of Sombrero clusters. The level of clustering is increased as the number of nodes in the system increases. As shown in chapter 5, the required number of messages increases by a small constant factor as the level of clustering increases. It can therefore be safely concluded that the algorithm is scalable and will be stable as the number of nodes in the system increases.

It should be noted that the simulation has certain limitations. One is that the algorithm would result in an overhead for the migration of threads such as those caused by consistency management of the shared memory. After any thread control block has been transferred to be scheduled on some other processor, the number of page faults is expected to rise for access to the thread data and instructions. A good selection policy should choose an appropriate thread so that there are as few page faults as possible. In addition, as the size of the system increases in terms of the number of nodes present, the network latency is expected to rise with it. Network latency will cause consistency management problems for the shared memory. This in turn could affect the distributed scheduling algorithm adversely as shared memory is used for its load tables.

The research resulted in functioning software for distributed scheduling in Sombrero, which can be used for further experimentation and development. An important benefit was that several additional issues (e.g., inter-node communication, address space allocation, scalability, etc.) necessary for the full implementation of a functioning prototype of Sombrero were addressed and working parameters for them were developed in this project.

6.1 Summary of Value of Research

The main aim of the thesis was to design, implement and test a distributed scheduling mechanism for Sombrero. Specifically, the thesis addressed the following issues in Sombrero:

1. Implemented data structures for supporting copy set graphs of distributed memory objects and threads.
2. Constructed a facility (Sombrero router) for communications in Sombrero that uses a VA to IP-address translation service. This facility is implemented using TCP/IP on NT 4.0.
3. Implemented an algorithm for distributing the 2^{43} bytes address space amongst Sombrero nodes.
4. Implemented algorithms and data structures for scheduling on a single node.
5. Implemented algorithms and data structures for distributed scheduling on multiple nodes using load balancing.
6. Tested the correctness, stability and scalability of the distributed scheduling algorithm.

6.2 Future Work

Currently, the distributed scheduling mechanism for Sombrero has been implemented as a simulation. The task of porting the distributed scheduling algorithm from NT to Sombrero has been initiated. This task can only be completed when the Sombrero prototype is operational on the alpha 21164 machines. The distributed scheduling mechanism needs a functional prototype which supports essential features such as protections domains, Sombrero thread support, token tracking, etc.

6.2.1 Sombrero Node

The code for the simulation of the Sombrero node only demonstrates the correctness of the distributed scheduling algorithm. This code uses NT's Win32 API to simulate Sombrero like behavior. It carries out functions like contacting the assigned Sombrero router for an address space, sets up shared memory for the load tables, creates copy sets for load tables, starting the

local and distributed schedulers, etc. Most of this code will not be needed for the Sombrero prototype. Equivalent Sombrero code to implement the above mentioned functionality has to be implemented for the Sombrero prototype. The functions that implement the distributed scheduling algorithm are:

- `void DistributedScheduler()`
- `void DoClusterLoadBalancing(UINT nLevel)`
- `void DoLoadBalancing(struct LoadData *pLoadData, UINT nIndex)`

The above three functions have been ported to Sombrero and are successfully compiled. The function `BOOL CMainFrame::PreTranslateMessage(MSG* pMsg)` contained in MainFrm.cpp carries out the function of communication with other Sombrero nodes through the Sombrero router. The task of selection policy [13] i.e. the task of selecting threads for migration is also carried out in this function. Code in this function can be reused to implement the actual communication protocols for the Sombrero prototype. The porting of this function cannot be carried out until the Sombrero prototype is completely functional.

6.2.2 Sombrero Router

The Sombrero router developed for use with the simulation can be reused with the Sombrero prototype with a few changes. Currently the router uses 32-bit addresses. The router will function correctly with the Sombrero prototype by changing the necessary variable and message array types to 64-bit `unsigned int`. In addition, the code specific for the functioning of the distributed scheduling simulation on NT should be removed.

REFERENCES

[1] Skousen, A. C. Sombrero: a very large single address space distributed operating system. Thesis (MS), Arizona State University, 1994.

[2] Sinha, P. K. *Distributed operating systems: concepts and design,* Chapter 7. IEEE Press, 1997.

[3] Zaki, M. J., and Li, W., and Parthasarthy, S. Customized Dynamic Load Balancing for a Network of Workstations. Computer Science Department, The University of Rochester, Technical Report TR-602, December 1995.

[4] Skousen, A. C., and Miller, D. S. Using a Single Address Space Operating System for Distributed Computing and High Performance. In *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99),* February 1999.

[5] Skousen, A. C., and Miller, D. S. Resource Access and Protection in the Sombrero Protection Model, Software Protection Data Structures and Hardware Range Protection Lookaside Buffer. *ASU 64 Bit OS Group White Paper 2,* Computer Science and Engineering Department, Technical Report TR-95-013, January 1996.

[6] Skousen, A. C., and Miller, D. S. Implementing a Very Large Single Address Space Across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency and Fault Tolerance. *ASU 64 Bit OS Group White Paper 10,* Computer Science and Engineering Department, Technical Report TR-95-021, January 1996.

[7] Skousen, A. C., and Miller, D. S. The Sombrero Distributed Single Address Space Project", In *Proceedings of the 2nd USENIX Windows NT Symposium,* August 1998.

[8] Skousen, A. C., and Miller, D. S. Operating System Structure and Processor Architecture for a Large Distributed Single Address Space. In *Proceedings of PDCS'98: 10th International Conference on Parallel and Distributed Computing Systems,* October 1998.

[9] Skousen, A. C., and Miller, D. S. Using a Distributed Single Address Space Operating System to Support Modern Cluster Computing. In *Proceedings of 32nd Hawaii International Conference on System Sciences*, HICSS-32, January 1999.

[10] Osser, W. Automatic Process Selection for Load Balancing. Thesis (MS), University of California, Santa Cruz, June 1992.

[11] Wagle, Y. S. Process Migration in Network Parallel Software Systems. Thesis (MS), The University of Texas, Austin, December 1994.

[12] Tanenbaum, A. S. *Distributed Operating Systems.* Prentice-Hall, Inc., 1995.

[13] Singhal, M. K., Shivaratri, N. G. *Advanced concepts in operating systems: distributed, database, and multiprocessor operating systems*, Chapter-11. McGraw-Hill, 1994.

[14] Beveridge, Wiener. *Multithreading Applications in Win32.* Addison-Wesley Developer's Press, 1997.

[15] Richter, J. *Advanced Windows.* Microsoft Press, 1996.

[16] Kriton, K., Rafael, A., An Experimental Comparison of Initial Placement Vs. Process Migration for Load Balancing Strategies. Department of Computer Science, Princeton University, Technical Report CS-TR-199-88, December 1988.

[17] Zhou, S., Zheng, X., Wang, J., Delisle, P., Utopia: A Load Sharing System for Large, Heterogeneous Distributed Computer Systems, University of Toronto, Technical Report CSRI-257, November 1991.

APPENDIX A

MESSAGE FORMATS

A. Message Formats

A.1 Inter-Node Messages

The format of the messages sent between the Sombrero nodes and routers is explained in this chapter. An array of `unsigned integers` is used for the communication of messages using sockets. The first two integers in any message should be used for the message size and type respectively. Two constants viz., MSGSIZE and MSGTYPE have been defined for this purpose. If a message has to be sent to a particular node through a router then the third integer should be the destination address. The file protocols.h defines the constants for the message formats.

- NEWNODE, NEWROUTER: - These messages are sent as a part of initialization of a Sombrero node or a router. The NEWNODE message is sent by a node to the router to which it has been assigned. The NEWROUTER message is broadcast to all existing routers in the system.

```
/*
buffer[2]: dummy or new router ID
buffer[3]: port, buffer[4]: host name
*/
enum msgnew {
NROUTERID=2,
PORT,
HOSTNAME
};
```

- GETTHREADS, RGETTHREADS, SENDTHREADS: - The nodes use these messages to exchange threads between each other.

```
/*
buffer[2]: destination, buffer[3]: source
buffer[4]: level of clusters involved, buffer[5]: max number
of threads, buffer[6,7,..]: thread control blocks
*/
enum msgsend {
   DESTINATION=2,   //Destination has to be at position 2
         SOURCE,
         LEVEL,
         THREADCOUNT,
         TCB
};
```

- ASREQUEST: - A Sombrero node uses this message to make an address space request to its router. The router keeps track of a parent node to which the request can be made.

```
/*
buffer[2]:router flag, buffer[3]:parent,
buffer[4]:temporary ID, buffer[5]: router ID
*/
enum msgasrequest {
   ROUTERFLAG=2,
         PARENT,
         TEMPID,
         ROUTERID
};
```

- ASREPLY: - A node sends this message as a reply to an ASREQUEST message. This message carries information about the granted address space.

```
/*
buffer[2]: router flag, buffer[3]: parent,
buffer[4]: temporary ID, buffer[5]: router ID,
buffer[6]: new rmocb, buffer[7]: range
*/
enum msgasreply {
   ASROUTERFLAG=2,
         ASPARENT,
         ASTEMPID,
         ASROUTERID,
         ASRMOCB,
         ASRANGE
};
```

- UPDATION: - A router broadcasts an UPDATION message to all other routers when a new entry is added to its local router table. This happens when an address space is granted to a new node.

```
/*
buffer[2]: router flag, buffer[3]: router ID
buffer[4]: new rmocb, buffer[5]: range
*/
enum msgupdation {
   UROUTERFLAG = 2,
         UROUTERID,
         URMOCB,
         URANGE
};
```

- CSNEW: - This message is sent by the root node of a copy set graph to all those nodes that have to be inserted in the graph. The destination node then sends a CSREQUEST to the root.

```
/*
buffer[2]: destination MOCB, buffer[3]: source MOCB,
buffer[4]: load table address
*/
enum msgcsnew {
   CSNDMOCB = 2,    //Destination has to be at position 2
         CSNSMOCB,
         CSNLTA,
         CSNMOCB
};
```

- CSREQUEST: - This message is sent by a node to the root node of a copy set graph to request its insertion in the graph. It is also used by the root to forward the insertion request along the graph.

```
/*
buffer[2]: destination MOCB, buffer[3]: source MOCB,
buffer[4]: load table address
*/
enum msgcsrequest {
   CSDMOCB = 2,     //Destination has to be at position 2
         CSSMOCB,
         CSLTA
};
```

- CSREPLY: - When a node is finally inserted in a copy set graph, the parent of the requesting node sends the CSREPLY message to the node.

```
/*
buffer[2]: destination MOCB, buffer[3]: CS parent
buffer[4]: load table address
*/
enum msgcsreply {
   CSRDMOCB = 2,
         CSRPARENT,
         CSRLTA
};
```

- TRANSFERREQUEST: - An active node which initiates thread exchanges at level one clusters and above sends the TRANSFERREQUEST message to the target cluster.

```
/*
buffer[2]: destination MOCB, buffer[3]: load table address,
buffer[3]: source MOCB, buffer[4]: source load,
buffer[5]: count of nodes interested in transfer,
buffer[6,7...]: nodes interested in transfer
*/
enum msgtransferrequest {
   TRQDEST=2,
         TRQLTA,
         TRQSOURCE,
         TRQSLOAD,
         TRQCOUNT,
         TRQMOCB
};
```

- TRANSFERACK: - A node replies to a TRANSFERREQUEST message with a TRANSFERACK message when it finds that the first level cluster it belongs to can take part in the load balancing.

```
/*
buffer[2]: destination MOCB,
buffer[3]: negotiated number of transfers
*/
enum msgtransferack {
   TRADEST=2,
         TRACOUNT
};
```

- DOTRANSFER: - A target node that can participate in a thread transfer for load balancing receives this message.

```
/*
buffer[2]: destination MOCB, buffer[3]: target node
*/
enum msgdotransfer {
   DTDEST=2,        //Destination has to be at position 2
         DTNODE,
};
```

A.2 Windows Messages

In the simulation, the distributed scheduler sends Windows based messages to the main Window, which is responsible for the internode communication. The purpose and structure of each of these messages are given below.

- UWM_TRANSFERREQUEST: - This message instructs the main Window thread to send a TRANFERREQUEST message to the specified destination node. The parameters for this message are

  WPARAM: pointer to the TRANSFERREQUEST message

  LPARAM: size of the message

- UWM_GETTHREADS, UWM_RGETTHREADS: - The main Window thread initiates the thread exchange protocol with the specified Sombrero node. The parameter for this message is

  WPARAM: destination RMOCB

- UWM_CREATEASMUTEX: - A mutex is created and held by the main Window thread. The mutex is released when an address space is assigned to the node.
- UWM_ASREQUEST: - An address space request is sent to the router.
- UWM_CSNEW: - A CSNEW message about a newly created load table address is sent to the specified MOCB. The specified MOCB could be either a load table address or a RMOCB. The parameters for this message are

  WPARAM: load table address

  LPARAM: destination MOCB

- UWM_CSREQUEST: - A CSREQUEST message for insertion in the copy set graph of a load table address is sent through the router. The parameter for this message is

  WPARAM: load table address

APPENDIX B

SOMBRERO SCRIPT FILES

B. Sombrero Script Files

To run the simulation of distributed scheduling for Sombrero two types of script files are needed. The Sombrero routers need one type of script file, one file for each router. The other type is needed by the Sombrero nodes, one file for each node.

B.1 Sombrero Router Script File

A Sombrero router contacts all the existing routers as part of its initialization process (NEWROUTER message). This is done so that the other routers can make an entry for the new router in their router table. In addition, the new router needs to make entries for all existing routers in its own router table. A script file is needed to give the host names and ports of the existing routers. This script file also gives the router its unique identification number. Each Sombrero router has a unique 64-bit (32-bit in the simulation) odd identification number. The format for the router's script file is given:

```
Identification number
Host-name  Port-number
Host-name  Port-number
Host-name  Port-number
…
```

Figure 6. Format of the router script file

B.2 Sombrero Node Script File

A Sombrero node maps shared memory for load tables in two different situations: First, as a part of initializations, when the node reads addresses of already existing load tables from the script file. If this node is the owner of the copy set root of the load table then it may inform some already existing nodes about the new load table (CSNEW message). RMOCB addresses of these interested nodes are available in the node's script file. On receiving a CSNEW message about a particular load table, a Sombrero node maps the corresponding shared memory.

Second, during operations a node might map new load tables. The script file consists of a list of load table addresses a node should map. Along with each load table address, a list of the nodes to which CSNEW message is to be sent may be given. The list is to be given only if the node is the copy set owner of the concerned load table. The members of this list could be either a RMOCB of a node or address of another load table. In the latter case, the copy set owner broadcasts the CSNEW message to the members of its copy set. In this table, the value 0xffffffffffffffff acts as a delimiter. The format of a node script file is given below:

```
Load Table Address
      0xffffffffffffffff
Load Table Address
      RMOCB or Load Table Address
      …
      0xffffffffffffffff
Load Table Address
…
0xffffffffffffffff
```

Figure 7. Format of the node script file

APPENDIX C

SOURCE CODE

C. Source Code

Appendix C contains a copy of the entire source code for distributed scheduling in Sombrero. Included are the simulator source code and the code for the distributed scheduling algorithm that has been ported from NT to Sombrero. A copy of this document is available in the Operating Systems Laboratory (GWC 321), located in the Goldwater Center at Arizona State University. A soft copy is also available in the ‘milind’ directory in the ‘rdmiller’ account on enuxsa.eas.asu.edu at Arizona State University.

APPENDIX D

SIMULATION TEST INPUTS

D. Simulation Test Inputs

Appendix D contains a copy of the simulation test inputs for distributed scheduling in Sombrero. The results corresponding to these inputs were observed on the output windows of the simulation. A copy of this document is available in the Operating Systems Laboratory (GWC 321), located in the Goldwater Center at Arizona State University. . A soft copy is also available in the ‘milind’ directory in the ‘rdmiller’ account on enuxsa.eas.asu.edu at Arizona State University.