AN IMPLEMENTATION OF THE SCHIZOPHRENIC WORKSTATION

by

Leigh B. Stoller

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

December 1993

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Leigh B. Stoller

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Mark Swanson

Robert Kessler

Gary Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Leigh B. Stoller in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the supervisory committee and is ready for submission to The Graduate School.

Date

Mark Swanson
Chair, Supervisory Committee

Approved for the Major Department

Thomas Henderson
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

Today's workstations are extremely powerful and commonplace in many commercial and research organizations. These workstations tend to be underutilized much of time, as their owners engage in meetings, think about research, edit, debug, or go home at night. At other times, the workstation owner needs all of the power his or her workstation can provide, and possibly far more than it can provide. The Schizophrenic Workstation System is a cycle harvesting distributed operating system that utilizes spare workstation resources to shift load from overburdened workstations to idle or underutilized ones. A prototype implementation of the system, hosted on commonly available workstations, is described. The prototype was used to perform a series of experiments to judge the effectiveness of the Schizophrenic Workstation System design, and to determine what applications are suitable for it. The results indicate that the system is suitable for long running, compute-intensive applications that are not time-completion critical. Further, many of the system design goals, such as ease of use and efficiency, have successfully been met.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Dr. Mark Swanson for developing the design on which this thesis is based, and for providing guidance during this work. I would also like to thank Dr. Robert Kessler for his support over the years, and Dr. Gary Lindstrom for providing additional comment and suggestions. Lastly, I would like to thank Dr. Al Davis for giving me the opportunity to complete my degree through his ongoing support of the Center for Software Science.

CHAPTER 1

INTRODUCTION

Today's workstations are extremely powerful and commonplace in many commercial and research organizations. These workstations tend to be underutilized much of time, as their owners engage in meetings, think about research, edit, debug, or go home at night. At other times, the workstation owner needs all of the power his or her workstation can provide, and possibly far more than it can provide. The Schizophrenic Workstation System[21] is a cycle harvesting distributed operating system that utilizes spare workstation resources to shift load from overburdened workstations to idle or underutilized ones. The goal of this project is to implement a functional prototype of the Schizophrenic Workstation System (hereafter called simply Schizo), using existing microkernel technology, on a local area network of workstations.

A second goal of this project is to demonstrate the effectiveness of the Schizo design. To accomplish this, a large application was run on the Schizo system. By utilizing *task migration*, the application migrates from node to node, allowing it to survive node shutdown and increases in load on the participating workstations, continuing to run until normal termination. In addition, workstation owners are subjected to little inconvenience, such as degraded performance or decreased reliability, when Schizo is utilizing their workstations.

A third goal is to classify the types of applications that are suitable for the Schizo system. A series of synthetic tests and benchmarks were performed to determine the effects of remote execution and migration on both applications and the participating workstations. The results of these experiments were used to draw conclusions about the class of applications that are best suited for the Schizo system.

The remainder of the thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 gives an overview of the operating system base from the application programming level. Chapter 4 provides an overview of the Schizo design. Chapters 5 and 6 describe the Schizo implementation in detail. Chapter 7 describes the experimental results. Finally, conclusions and potential areas of future research are given in Chapter 8.

CHAPTER 2

RELATED WORK

Distributed operating systems is an area of research that seeks to combine the individual power of autonomous computers into larger, coordinated systems. In most distributed operating systems, some level of transparency is afforded to the users. Location transparency allows a program to function without knowing whether an object is local or remote, while reference transparency allows a program to access local and remote objects in the same way. An object can be reference transparent without being location transparent. The *port* abstraction of Chorus [17] and Mach [16] is an example of how transparency can be provided. An application simply sends a *message* to a port; the operating system is responsible for finding the receiver and delivering the message. Another example is the *RPC* mechanism of Clouds [24], which treats all services as *objects* to which invocations can either be local or remote. In distributed systems such as these, location and reference transparency can be used to create a large time-sharing system for running large applications (possibly with *process-level* parallelism), or one on which medium-grained parallel programs can be be run.

A different class of distributed system, known as *network operating systems*, seeks to combine networks of independent workstations by providing transparent access to remote resources, such as files and I/O devices. Locus [23], Utopia [28] and Condor [10] are examples of network operating systems that use remote access mechanisms to make nonlocal resources appear local. In Locus, remote files are operated upon by converting system calls into kernel to kernel RPC. In Condor, system calls are handled by a *shadow* program that converts system calls into RPC. This technique requires that Condor programs be linked against a special runtime library to handle the conversion. In Utopia, a software layer implemented entirely at the user level is interposed between the application and the operating system. RPC mechanisms are used to provide global scheduling information, and remote execution.

Some distributed systems can also be described as either *load sharing* or *load balancing*. Load sharing systems employ local and/or global scheduling algorithms [7, 28, 22] to statically determine which workstation is the best candidate for a new task, or whether a newly arriving task should be transferred elsewhere. For example, the Load Information Manager in Utopia periodically gathers statistics about the participating workstations, using CPU average, memory availability and the number of users logged in to determine which workstation is the best candidate for a new task. In a system by Theimer, et al.[22], a slightly less centralized approach is used. Whenever a process needs to be scheduled, a multicast query is sent over the network. Nodes that consider themselves candidates for accepting a new process respond with load information. A best choice selection is then made based on the information returned.

Encompassing load sharing systems are load balancing systems [18, 20, 5], which consider the system as a whole, attempting to remove imbalances in the system either by transferring new tasks as they arrive, or by migrating already running tasks. Load balancing systems tend to be less centralized and more dynamic than load sharing systems since they attempt to keep the system evenly balanced at all times. As a result, these systems are often more complex and expensive to implement. As an example, consider the "Balancing Unfinished Work" algorithm presented in [5]. In this algorithm each node periodically updates all other nodes with load information. Using this information, the algorithm seeks to minimize the amount of unfinished work on each processor by sending jobs from overloaded nodes to underloaded nodes. At the end of each scheduling period, each node estimates the amount of unfinished work it has left to do, and then tries to determine what node has the best chance of completing that work in the shortest amount of time.

Another common aspect of distributed operating systems is whether the individual workstations retain their identity. Instead of being part of a processor pool, as in Amoeba [12], workstations in systems such as Sprite [15], Condor [10] and Butler [13] are both *owned* by someone and are participating in a load sharing system. Owners of those workstations expect them to be reliable, and to provide all the computing power they have come to expect, when they want it. Systems of this type use heuristics to determine when a workstations is idle, and when a foreign process must be evicted because the owner has computing needs of his own. In Butler, eviction is accomplished by terminating the foreign process. In Sprite and Condor, the foreign process is migrated to another workstation. Systems such as these are generally not compatible with load balancing systems since workstations come and go from the resource pool as their owners come and go.

For systems that support migration, the process's memory is moved by a variety of means. In Condor, a process is migrated by checkpointing the program; a "core dump" file serves as the checkpoint, which is transferred to the destination node and used to reconstruct the state of the program. In Sprite, the virtual memory system is used to flush dirty pages to the process' paging file. When the process restarts on another node, it faults in pages from the same paging file, which is available via Sprite's network filesystem. Yet another approach is the one taken by Clouds and the NORMA version of Mach. These systems provide a *distributed shared memory* [9] implementation that allows memory to span multiple nodes. Migration involves transferring process state to the destination node and then restarting; the memory is already available as a result of the DSM system. As the process continues to run, its pages are transferred lazily to the destination node.

CHAPTER 3

OPERATING SYSTEM BASE

Before discussing the Schizo design, it is necessary to introduce the operating system base on which Schizo will be built. The Mach microkernel will be described, as well as extensions to the microkernel that enable a cluster of workstations to act as a single system. Although the Schizo design does not rely on any particular operating system personality, Schizo does use the OSF/1 Single Server, which is described below.

3.1 Microkernel Technology

Mach[16] is a multiprocessor operating system that was originally developed at Carnegie Mellon University. Designed as an operating system that could provide support tailored to multiprocessor machines, Mach is less “featurized” than traditional Unix [8] systems. Like Chorus, Mach has taken a “microkernel” approach, in which the core system provides only essential services, thus reducing its size and complexity. These basic services include virtual memory, process creation and scheduling, interprocess communication, and access to I/O devices. Everything else is implemented in “servers” that run as separate applications on top of the microkernel. The Unix server is an example, providing Unix semantics and functionality, as well as a Unix filesystem. One benefit of this approach is that Mach is a modular, easily extensible operating system. The following sections describe the services provided by the Mach microkernel.

3.1.1 Ports and Messages

In Mach, the *port* is the basic abstraction for establishing communication channels between tasks. A port is a protected queuing object into which messages can be placed and removed, and may have any number of senders, but only one receiver. Access to ports is controlled by send and receive rights; a task must have received the proper right in a message in order to use a port. A right can also be described as a "capability" since holding the right implies permission to either use the right or pass it on. A *message* is a collection of data objects that is to be communicated between tasks using send and receive primitives. Together, ports and messages provide Mach interprocess communication (IPC). An interface generator called MIG is used to create stub routines that handle the *marshalling* of data structures into messages which are then sent to a port. On the receiver side, another stub unpacks the message, thus providing a programming interface much like procedure call invocation. As previously mentioned, Mach ports provide location and reference transparency allowing the application to send and receive messages without knowing where the other side of the channel actually resides. In fact, most Mach services are invoked as procedure calls on ports.

3.1.2 Virtual Memory

Virtual memory (VM) automatically manages the memory hierarchy, allowing processes to share the relatively small amount of physical memory between them. Integral to this sharing is the protection of the memory space, which prevents a process from accessing memory that is not in its address space, whether it be deliberate or a program error[6]. The virtual memory system provides the abstractions that allow processes to run in their own address space, potentially sharing data with other processes using *shared memory*. Virtual memory can potentially be used to improve the speed of local (intranode) interprocess communication by "mapping" data quickly from one process to another. Integrating the virtual memory system with interprocess communication also simplifies the development of message passing applications since large blocks of data can be moved without regard to their contents. This technique can be extended to sharing memory between processes on different nodes by using distributed shared memory to avoid the details of message sending and receiving.

The Mach microkernel provides a virtual memory implementation that addresses all of these issues. The key abstraction of the Mach virtual memory system is the *memory object*[26], which like other Mach abstractions, is represented by a port. A memory object represents a region of memory that can be mapped by a task. The backing store for the region is provided by a memory manager, which is said to *implement* the memory object. Accesses to the region may result in a page fault if the page in question is not resident in memory. Page faults are satisfied by sending a message to the memory manager for the object, which provides the page. When the microkernel needs to free memory, possibly due to a resource shortage, a page is selected and sent to the memory manager of the object that page belongs to.

Basic memory management support is provided by the default, in-kernel memory manager that functions as the backing store for anonymous (nonpermanent) memory. In other words, the default memory manager implements an anonymous memory region by supplying zero-filled pages at first access, and paging out dirty pages to a standard UNIX file. More sophisticated memory management is possible when provided by a user-level memory manager, referred to as an *external pager*. As a user process, the memory manager can provide more exotic forms of memory management since it has access to the same application subsystems as any other process. For example, an external pager can be used to implement distributed shared memory among a cluster of workstations connected with Ethernet by using `sockets` to establish the communication channels. The external pager would supply the pages as necessary, and would be responsible for keeping the memory consistent among the nodes.

The Mach VM abstractions allow memory regions to be tagged with particular access protections. Mach VM also provides a simple mechanism that allows processes to share data, thus aiding in the development of parallel and/or cooperating algorithms. Since most Mach VM abstractions are implemented as procedure calls on ports representing memory objects, such objects can be manipulated by other tasks holding a reference to those ports.

3.1.3 Tasks and Threads

Unix kernels have the notion of a process, with its own address space and a thread of control. Traditionally a process could not share data with another, so there was no way to achieve parallel computation on a data set. Modern Unix systems allow processes to share memory, but only within predefined address regions. Many of the system resources associated with each process, such as page tables, are duplicated. Although it is possible to achieve parallel computation, the processes doing the computation are "heavyweight," incurring considerable CPU overhead to switch contexts.

Mach, on the other hand, splits the process abstraction into two components: the *task* and the *thread*[3]. A Mach task is simply an address space and other system resources such as port rights, but it does not contain a flow of control. A Mach thread provides the flow of control. Within a task there is at least one, and possibly several, threads executing, sharing one address space and associated resources. Since these threads share a common virtual space, they consume fewer system resources than Unix processes, and can potentially be context switched with less CPU overhead. On a uniprocessor, execution is pseudo-concurrent, whereas on a shared memory multiprocessor several threads may execute in parallel.

Mach threads and tasks are further examples of facilities that are referenced by ports. A special type of port, called a *kernel port*, names a task or thread. The difference between this type of port and a normal port is that only the microkernel can hold the receive right; the right can not be extracted and transmitted elsewhere. In all other respects, a kernel port is just like any other port. For example, one thread can suspend another thread by sending a message to the other thread's kernel port, even if that thread is on another node. Tasks can be manipulated in a similar manner.

3.1.4 I/O Devices

The Mach microkernel handles most interactions with I/O device hardware. Devices such as disks, consoles, etc. can be accessed via a port to the device. Operations to manipulate the device take the form of messages sent to the port. For example, to write a character to a console device, a message is sent to the port representing the device; reading a character requires waiting for a message to arrive from the same port. Again, since ports provide location and reference transparency, the device does not need to be on the local node, but can potentially be remote. In addition, the protected nature of ports implies that only tasks that have been granted the proper capability can manipulate a device. Typically, the operating system personality is granted complete access to all devices, which can in turn pass that capability on to "trusted" programs.

3.2 NORMA Version

The NORMA version of the microkernel is a separate version of Mach under development by the Open Software Foundation[14], and is so called because it contains direct support for a single system image on NO Remote Memory Access multicomputers. The direct support entails changes to the microkernel such that a collection of microkernels, each running on a separate processor, presents a unified port name space, a shared virtual memory system providing distributed shared memory[1], and a mechanism to specify the processor on which a new task is created. The NORMA microkernel contains additional support for determining system configuration at system startup.

Location and reference transparency for virtual memory relies on the unified port name space since memory objects are referred to via ports. It also requires an extension of the underlying paging mechanisms, since the contents of memory objects are referenced by virtual addresses. A reference to an address whose physical page is not resident on the local node is resolved via the in-kernel memory manager, resulting in a network transfer of the appropriate page. This approach is similar to earlier attempts which employed user-level external memory managers to resolve page faults [4]. A more in-depth treatment of the NORMA microkernel is given in Section 5.1.

3.3 Operating System Personality

As mentioned above, the Mach microkernel provides only basic operating system support. More traditional support, such as filesystems, signals, and other facilities associated with (for example) UNIX must be provided by the operating system personality. One such personality is OSF/1, which presents its own set of abstractions constructed on a base of Mach abstractions. For example, an OSF/1 process comprises a Mach task with an associated Mach thread and virtual memory objects representing its text and data. System calls are accomplished by converting UNIX system calls into Mach messages that are delivered to a port (called the *BSD request port*), whose receive right is held by the OSF/1 server. System call return values are delivered to the caller in the same manner using a reply port passed along with the request message.[1]

[1]Since an OSF/1 process is contained within a Mach task, it is often the case that referring to a "task" is equivalent to referring to a "process." Except where noted, the terms process and

The OSF/1 personality is actually divided into two parts: the server and the *emulator*. The emulator is a library that is used to transparently convert system call requests from the older, trap-based model, into the message-based model. The advantage of such an approach is that binary compatibility with older UNIX systems can be maintained, easing the operating system development process. In OSF/1, each process has the emulator loaded into its address space when the process is created. System call traps take the process into the microkernel, which returns control of the system call back to the user process at an address within the emulator. The emulator is then responsible for interpreting the system call request, marshalling up the arguments, and forwarding the request to the OSF/1 server as a message. The reply message from the server is then unpacked by the emulator, and the result returned to the program. As will be discussed later, the emulator is used in the implementation of the Schizo system. Figure 3.1 shows the system organization for the microkernel, server, and one user program. The numbered arrows indicate the system call pathway, starting with the trap into the microkernel (arrow "1"), branching into the emulator (arrow "2"), communicating with the OSF/1 server (arrow "3"), and ending when the emulator returns the system call result to the user program (arrow "4").

In addition to the standard OSF/1 server, there exists a new version under development by the Open Software Foundation and Locus Computing Corporation. This new version, called OSF/1 AD TNC[27], is designed specifically for NORMA multicomputers. Its primary goal is to avoid bottlenecks by using a distributed file system and a distributed process management system that allows process management system calls to be handled on the same node as the process. This verion of OSF/1 may have been more appropriate to use in conjunction with Schizo, but liscensing restrictions prevented it.

---

task are used interchangeably.

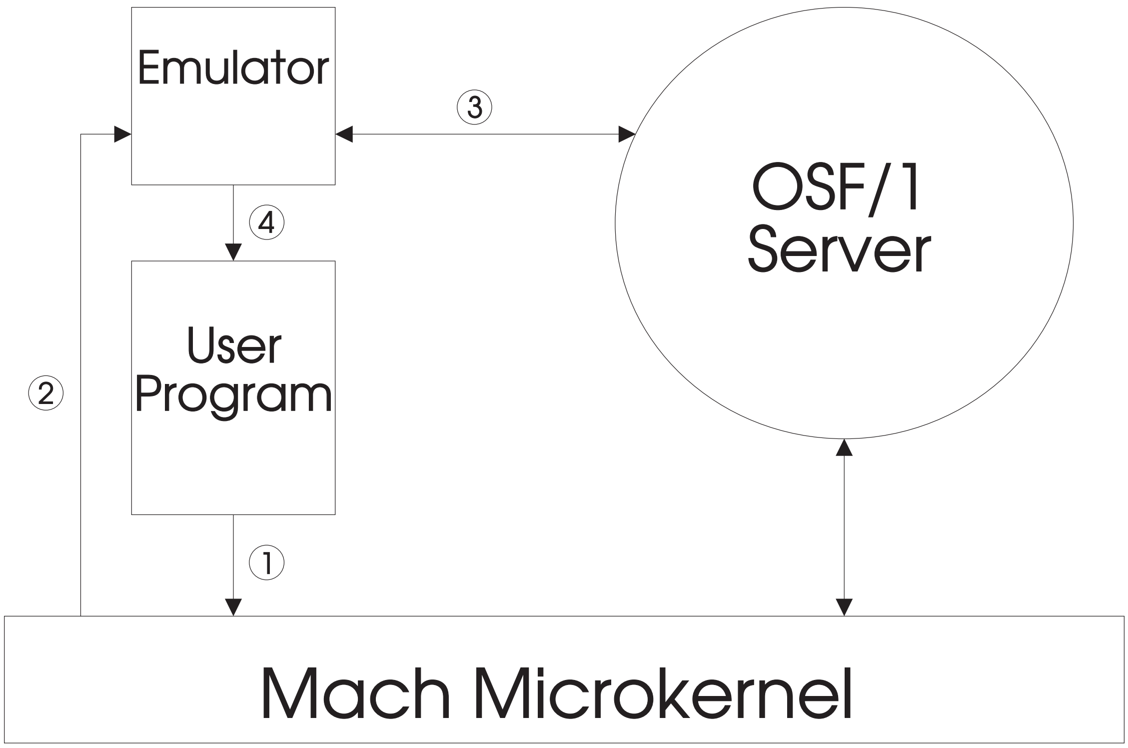

Figure 3.1. System organization for the microkernel, OSF/1 server, and one user program. Numbered arrows indicate the system call pathway for a user program.

CHAPTER 4

THE SCHIZO SYSTEM

The Schizophrenic Workstation System is a distributed operating system that provides a single distributed memory multiprocessor suitable for both time-shared processing and distributed memory parallel computation. It differs from distributed systems composed of dedicated (nonautonomous) workstations in that the constituent workstations retain an autonomous identity *in addition to* participating in the distributed system.

4.1 Motivation

The Schizo design focuses on several areas that must be addressed if a distributed system is to be generally usable by the research community. Central to the design is the need to convince workstation owners to allow their machines to be a part of the distributed system. The Schizo design is presented below, but first the key motivations behind the design are briefly described.

- *Efficiency*: How much overhead does the distributed system impose? If most of the computing capacity of the system is lost to the overhead of running the system, users will see no benefit in using the system.

- *Reliability*: How reliable is the system? There are two sides to this issue. First, the distributed system should not affect the reliability of the workstation being donated to the computing pool. Second, the distributed system should be able to tolerate node loss, allowing applications to continue to run unless catastrophic failure of the system occurs (i.e., power outage).

- *Security*: How well does the distributed system protect the workstation owner from malicious behavior? Guest processes should not be able to access files and devices that they would not ordinarily be able to access. The workstation should also be protected from resource exhaustion (swap space, memory, etc.) caused by an errant guest process.

- *Ease of use*: The system should be easy to use and administrate. The simplest case would be the ability to run an existing application binary on the distributed system unchanged. This implies a certain amount of transparency; applications should not have to be concerned with where they will execute. For example, an application that wishes to open a file should be able to open that file no matter where it is executing.

4.2 Design

The Schizo system is designed around many of the abstractions presented by the Mach microkernel. Central to this design is the ability to cleanly separate the Schizo system from the operating system personality, and the ability to transparently communicate between nodes in the cluster via the unified port name space. Other important aspects of the design are local and global scheduling requirements, and how Schizo interacts with I/O devices. The following sections describe the Schizo design, with attention paid to the Mach facilities that are central to each particular aspect of the design.

4.2.1 Schizo Server

The Schizo system is actually a server running on top of the Mach microkernel, much like the OSF/1 single server. In this sense, Schizo is a second personality for workstations that are participating in the system; one is the local OSF/1 system, the other is the Schizo distributed system. Further, since the Schizo system relies on the underlying microkernel to provide access to each of the workstations, only one workstation runs the Schizo server. The rest communicate with Schizo using Mach IPC mechanisms, and are available to Schizo solely as a resource base. When Schizo needs to schedule a process, it chooses a workstation in the cluster based on resource availability. Although global IPC makes this communication possible, the two personalities are still logically separate. Schizo and its tasks cannot interact with local workstation tasks, and workstation tasks cannot affect Schizo tasks. The only relationship is indirect, through the shared use of the underlying microkernel.

4.2.2 Global Scheduling

The Schizo server is responsible for global scheduling of tasks. The Schizo server periodically queries each of the participating workstations to gather load information. When a new task is created, or when an existing one needs to be migrated, a location is chosen based on the load of each of the participating workstations. If a workstation is found with its load below an acceptable threshold, the task is placed there. If no acceptable node can be found, the task is placed on the machine running the Schizo server. This centralized approach might not scale well to large numbers of processors, but should be adequate for tens or hundreds. Section 7.5 characterizes the effects of such an approach on the underlying network.

4.2.3 Local Scheduling

Local scheduling of Schizo tasks is handled by the normal microkernel priority scheduling system. When Schizo creates a task on a participating workstation, it depresses the priority of that task as low as possible to ensure that local workstation tasks have the CPU whenever they want it. Further, local scheduling information is used by the Schizo server to determine when a Schizo task needs to be migrated because local tasks are consuming most of the CPU, preventing the Schizo task from making adequate progress.

4.2.4 Device Access

In order to prevent both malicious and accidental misuse of local devices, Schizo is designed in such a way that direct device access is not required. Further, Schizo tasks are not expected to require access to particular devices in order to run correctly. This is enforced by the fact that only the local operating system server can provide this capability (in the form of a port right), and the fact that Schizo tasks cannot communicate with the local server to request the capability. Such separation reduces the possibility of interference with the local system, and ensures that dependencies tying a Schizo task to a particular workstation and its devices are not created. By avoiding direct device access, and thus dependency on particular devices, it is assured that a Schizo process can execute correctly on any node in the cluster.

Schizo tasks *do* potentially require indirect access to the local devices. For example, global IPC and virtual memory usage might result in messages being sent to other nodes, either to manipulate objects, or to fault in pages, thereby using the network device. Another example is local disk usage. Schizo tasks can potentially use the default pager for its backing store, thus requiring that pages be written to the local disk when memory runs low.

4.2.5 Static and Dynamic Resource Base

As previously mentioned, only one workstation runs the Schizo server; it is known as the Schizo "core" machine. Since Schizo processes are children of the OSF/1 server on the core machine, all operating system services are provided by this machine. If the core machine fails, the Schizo system terminates, and all of its tasks are lost. Otherwise, there are no dependencies on a particular workstation. Machines are considered to be "owned" by their primary users, and may come and go (shutdown or crash), often without warning. In the case of a controlled shutdown by a participating workstation, any resident Schizo tasks are migrated away to other workstations, or to the core node if no underutilized machine can be found. If a machine crashes, any resident Schizo tasks are lost. Once Schizo has determined that the machine is gone, it will attempt to clean up any residual state.

Figure 4.1 shows the system organization for a two node cluster. The Schizo server, shown as a dashed circle, is running on the core node. A single Schizo process with its emulator, shown as dashed boxes, is running on the remote node. The line labeled “A” indicates the physical communication pathway between the two microkernels. The line labeled “B” represents the logical communication channel between the remote Schizo process and the OSF/1 server on the core node, and is multiplexed over the physical connection.

4.2.6 Design Summary

The overriding theme of the design of Schizo is to maintain separation of the local workstation system from the distributed load sharing system. This separation is accomplished by implementing these two systems as separate servers running on

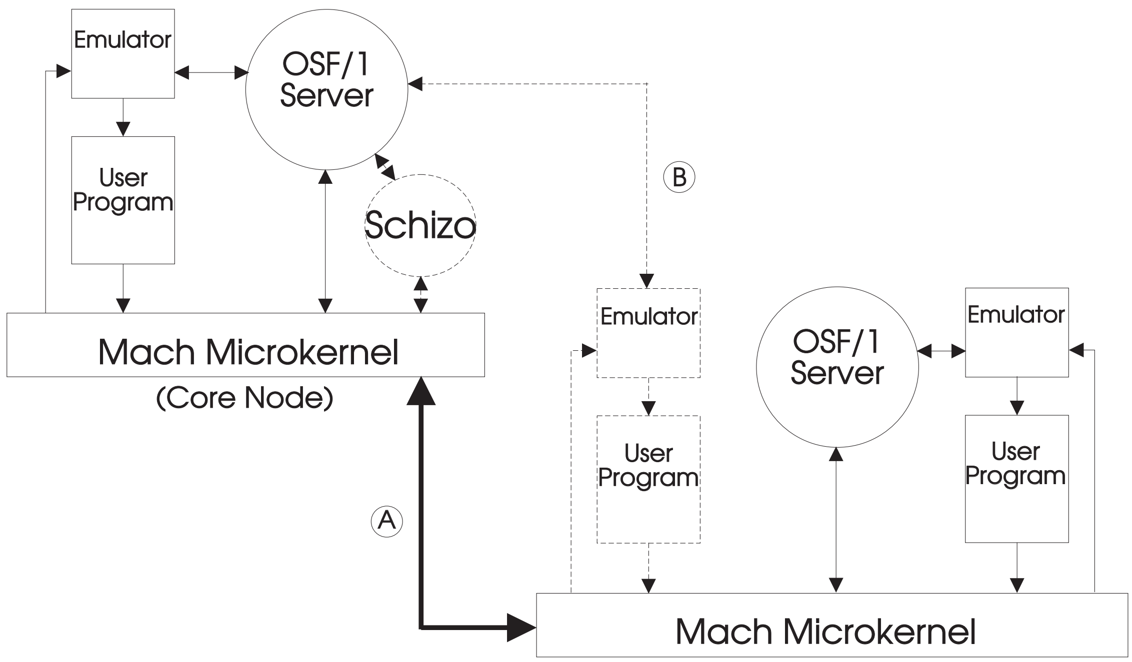

Figure 4.1. Organization of a two node cluster with a single Schizo process.

top of the virtual machine presented by the Mach microkernel. Schizo processes cannot interact with the local OSF/1 server, but instead communicate solely with the OSF/1 server on the core node. In cases in which the microkernel does not enforce adequate separation, Schizo adheres to appropriate *conventions*, such as eschewing direct access to the workstations devices, to maintain the separation.

CHAPTER 5

NORMA MICROKERNEL SUPPORT

The Schizo implementation can be divided into 2 layers. The bottom layer is the NORMA microkernel, whereas the top layer is the Schizo server. The NORMA microkernel (as implemented by the Open Software Foundation) provides the global IPC and distributed shared memory facilities needed by the Schizo server to operate. However, the NORMA microkernel does not fully support the Schizo design; additional support is required to implement Schizo. To provide context for the discussion of microkernel modifications, this chapter first gives an overview of the existing NORMA microkernel implementation. A treatment of the microkernel modifications is then given. The following chapter describes the Schizo server implementation.

5.1 Norma Microkernel Overview

The NORMA microkernel provides both global IPC and distributed shared memory on systems that provide "NO Remote Memory Access." Mach already provides an external memory manager interface that allows (potentially) independent tasks to share memory. These tasks can either be on the same node, or they can be on different nodes by employing the *network memory server*[4] to handle the sharing. The primary drawback to using an external memory manager to share memory between nodes is performance. The server is a user process, which results in more context switches than a macrokernel implementation. Further, the communication channel between the manager and the participating nodes is Mach

IPC. Traditional Mach microkernels do not provide IPC between nodes, so another external mechanism is used to convert Mach IPC to traditional network protocols (for example, TCP), and back again. The NORMA microkernel seeks to ameliorate these performance problems by moving the functionality into the microkernel from user space. The intended result is to reduce the number of context switches and the amount of processing needed to send and receive messages between nodes.

The following sections describe how the NORMA microkernel achieves global IPC and distributed shared memory. Section 5.2 then discusses the extensions that are necessary to completely implement the Schizo design.

5.1.1 Node Configuration

The first step in creating a cluster occurs when machines that are part of the cluster boot. As each node boots, an initialization routine is called to construct a broadcast packet that is sent out over the default network device. Some other machine on the same network will be running the *node daemon*, which is responsible for receiving the packet and constructing a reply. The reply will, among other things, contain the logical node number for the booting node, as well as the network addresses of all the nodes participating in the cluster. The microkernel uses this information to build a table relating logical node number to network address. When a packet is sent to a node, a table lookup is performed to find the proper network address, which is then inserted into the packet. Since this procedure is invoked at boot time, it is not possible to add nodes to the cluster at a later time.

5.1.2 Global IPC

Global IPC is accomplished by extending the port data structure to include a new field that indicates the node on which the receive right for a port resides. The local representation of the port is called the proxy, and the actual port is called the principal. In order to match the proxy port with its corresponding principal, a second new field holds a globally unique port identifier. When a message is to be delivered to a port that is remote (as indicated by a field in the proxy port), the data in the message is converted into the format mandated by the underlying network hardware, and transmitted over the wire to the remote node. The data are then reassembled into kernel representation, and delivered to the local port by matching the global port identifier contained in the message. It is quite possible that the port might have migrated during this process, in which case the local port is again a proxy, and the message is forwarded to the proper node.

Messages that need to be forwarded to remote nodes must be transmitted via some network device. In the current NORMA implementation, Ethernet is used to transmit Unreliable Datagram Protocol (UDP) [2] packets. When a message is to be sent to a remote node, it is converted into network format, and then copied into one or more UDP packets. The amount of data that can be transmitted in a UDP packet is limited by the frame size of the underlying transport mechanism. For Ethernet, 1450 bytes of message data can be transmitted in a single UDP packet, so a message with more data is divided into multiple UDP packets. The packets are then handed to the device layer for transmission. Since UDP, as its names suggests, is an unreliable protocol, a simple acknowledgment protocol is used to impose reliability as well as ordering. As the packets are received, the message is reconstructed. When all the packets have arrived, the message is handed up to the IPC layer, which delivers it to the destination port.

Kernel to kernel remote procedure call (RPC) is a special case of global IPC. An example usage of kernel RPC can be illustrated when a task on one node creates a task on another node. To create the task, a `task_create` message is sent to the *host port* of the remote machine. The host port is a special port that signifies that the receiver of the message is the remote microkernel, not a particular task running on the remote node. Like regular IPC, a reply port is passed in the message that indicates to which node the reply should be sent. In the case of `task_create`, a principal for the new task is created on the remote node, and a proxy port is returned. Kernel requests to that task via the proxy, are forwarded automatically to the remote kernel.

5.1.3 Send Rights

Mach messages may contain ports as well as arbitrary data. Mach semantics dictate that the holder of a receive right can request that a *no-senders* notification be issued when the number of send or send-once rights reaches zero. To do this, the total number of outstanding send and send-once rights per port within the system must be known at all times. In a nondistributed system, keeping track of this information is a straightforward reference counting problem. Whenever a send right is extracted from a receive right, the reference count is incremented. When the send right is later destroyed, the reference count is decremented. Send-once rights are slightly different in that using the send-once right destroys it, but reference count management is still simple to implement.

Keeping track of this information in a distributed system is a complex problem. Further, no-senders notification requires that a garbage detection algorithm[19] be used, rather than automatic garbage collection. The semantics of no-senders is that the holder of the receive right is notified as soon as the count reaches zero. Therefore, the NORMA microkernel uses distributed reference counting to implement garbage detection. A primary source of complexity in this algorithm is that these rights are sent via messages, and in fact, the only outstanding send right for a receiver could be contained within a message in transit.

As a specific example, consider Figure 5.1, in which a send right for a local port is exported to another node. To do this, a task on `Node 1` constructs a message to be sent to port `P2` on `Node 2`. The message contains send right `SR1`, which allows the recipient of the message on `Node 2` to send a message back to port `P1` on `Node`

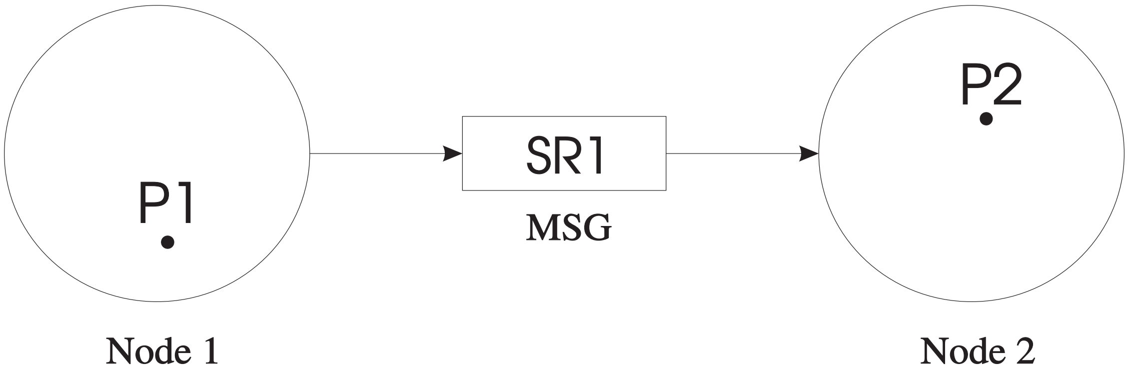

Figure 5.1. Exporting a send right for a local port.

1. In general terms, the message carries a capability that allows the receiver on the remote node to send a message to a port on the local node. Further, local port `P1` can be a principal (the port is indeed local), or a proxy. The latter case is shown in Figure 5.2. `PR1` is a proxy, whose receive right `P1` is on `Node 3`. As a result, send right `SR1` is a capability that allows the recipient of the message on `Node 2` to send a message to port `P1` on `Node 3`.

To handle the wide range of possibilities, the NORMA microkernel uses *transit counts* to track the number of rights in motion within the cluster. Transit counts track the number of rights that have been exported and are either in transit to another node or have arrived at another node. Using Figure 5.1 as an example, consider the case in which port `P1` has two outstanding send rights, and that one of those rights is to be exported in a message to `Node 2`. The microkernel on `Node 1`, upon determining this, will increment the `stransit` (number of *send* transits) field in `P1`'s port structure by one, and send the message on its way. When the message is received by `Node 2`, the send right count for `P1`'s proxy port will be incremented by one, and its `stransit` count decremented by one. In this way, the systemwide number of `stransits` sum to zero for the principal and proxies that represent a port, when no rights are actually in transit. The number of send-once rights are tracked in a similar manner, using `sotransit` counts.

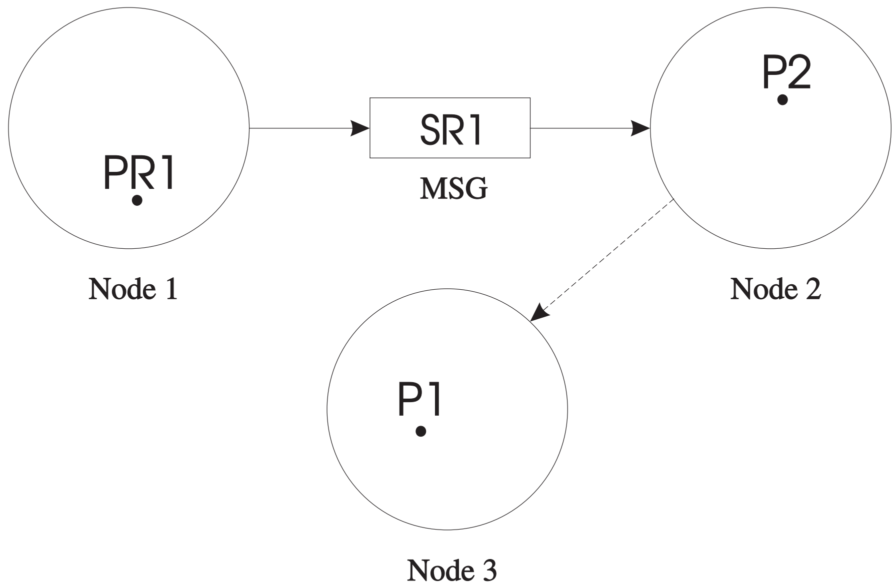

Figure 5.2. Exporting a send right for a proxy port.

There are several reasons for using transit counts in this manner. The first is that a proxy can exchange rights with proxies on other nodes. The sending node would simply increment the `stransit` count, while the receiver would decrement it. There is no need to inform the principal of the transfer since the total number would still sum to zero. Second, a proxy port can generate new send rights without telling the principal, if it has more `stransits` than send rights. This would happen if several send rights were exported to a node, and some were later destroyed. Since the number of rights is greater than zero, and the number of transits is less than one, the proxy can create as many send rights as the principal originally sent it. When the proxy can no longer create its own send rights, a message is sent to the principal requesting more. Lastly, no-senders notification can be implemented efficiently by allowing each node to handle it locally. As each node's number of send rights reaches zero, it notifies the principal, passing the number of remaining `stransits` along. This is the number of rights that were actually destroyed; rights that were transferred result in a smaller `stransit` count. The principal reduces its count of send rights and `stransits` by this amount. Eventually, the number of send rights and the number of `stransits` will reach zero, indicating that there are no more senders for the port.

5.1.4 Receive Rights

In Mach, receive rights are a special type of right, handled much differently than either send or send-once rights. Although there can be many send and send-once rights for a port, there is only a single receive right. A task holding a receive right can send the right in a message to another task, causing the right to be removed from the sending task and transferred to the destination task. Transfering a receive right to a local destination task is straightforward, but when the destination is on a remote node, a number of extra steps are necessary.

- A new principal must be created on the destination node. If there is an existing proxy port (which indicates that the node currently holds send rights for the port), it must be converted to a principal. This entails merging the port state of the proxy (number of rights, transits, etc.) with port state of the original receive right.

- The sending node must intercept incoming messages, and queue them while the receive right is being transferred. When enough port state has been successfully transferred, the queue of messages is forwarded to the new node and placed on the new port's incoming message queue.

- A forwarding pointer must be left behind so that messages arriving at a node where a receive right once was, can be forwarded to the proper node.

Of these, the issue of forwarding pointers deserves the most attention since it has an impact on fault tolerance. In the current NORMA microkernel implementation, forwarding chains are collapsed lazily. When a message for a migrated port is received, an error message is returned that tells the sending node the last known location for the port. The message is then resent to the new node. If the port has since migrated again, another error message indicating another last known location is returned. Eventually the message will find the proper node. In the presence of frequent port migration, this simple method works well, but is lacking in fault tolerance. However, for the system to be fault tolerant, it is necessary to ensure that there are no forwarding pointers to machines that have crashed or have been rebooted.

5.1.5 Paging System

In addition to global IPC, the NORMA microkernel also provides distributed shared memory among nodes in the cluster. The current version uses a *strict consistency* model; memory is always kept consistent between nodes by using a single-writer, multiple-reader protocol[9]. The implementation is an extension of the external pager mechanism discussed earlier; memory objects are implemented by pagers, which are referred to by ports. With global IPC, the pager backing an object can now reside on any node in the cluster. Combined with a memory manager (in kernel space) that provides strict memory consistency among multiple clients, distributed shared memory is achieved.

5.2 Microkernel Changes for Schizo

The NORMA microkernel, as it is implemented by OSF, does not fully support the Schizo design. Task migration and the ability to add and subtract nodes from the cluster dynamically each require additional support from the microkernel. Other Schizo features, such as fault tolerance, can only be supported with extensive modifications to the microkernel. This section describes modifications made so far, and proposed modifications of the NORMA microkernel in support of the Schizo design.

5.2.1 Dynamic Addition of Nodes

Schizo requires the ability to add nodes dynamically, so it was necessary to extend both the node daemon and the initialization code in the microkernel that constructs the node table. The node daemon was modified to send the node table to each node in the cluster for each boot request. Since the node daemon runs from a configuration file, it is simply a matter of changing the configuration file and booting the node that is to be added to the cluster. When the daemon receives the request packet from the node, it responds by sending the new table to each node in the cluster so that each node can reconstruct its node table. The initialization code in the microkernel was then modified to accept updated versions of the node configuration table from the node daemon. It is not possible to dynamically replace or delete a node from the cluster, so the initialization code must check to make sure that the new table contains additional nodes only. This reasons for this limitation are explained in more detail in the next section. It should be noted that this approach does not scale well to large numbers of nodes. A desirable modification of this strategy would be to implement a multicast protocol to reduce the amount of network traffic that occurs when a node boots.

5.2.2 Orderly Shutdown

The Schizo *design* calls for generalized fault tolerance; machines can crash or be rebooted, without adverse affects on the Schizo system. In the case of a machine crash, any tasks resident on that machine are lost, but Schizo tasks on other machines will continue to run normally. For controlled reboots, Schizo tasks are intended to be migrated away from the machine and restarted somewhere else in the cluster.

Machine crashes are extremely difficult to handle since the NORMA microkernel implementation relies on a large amount of distributed state. For example, the number of outstanding send rights for a port is distributed around the cluster. If a machine crashes, an unknown number of rights will disappear from the system. Since no-senders notification requires that the number of outstanding send rights be known, it would be necessary to implement a distributed reference counting system that could recover the number of outstanding rights in the system (the implementation of such a system is beyond the scope of this thesis).

On the other hand, handling controlled reboots is a more tractable problem that can be addressed by migrating resident tasks and cleaning up any residual state. When a machine is to be rebooted, control eventually passes into the microkernel shutdown code. A new routine has been added that iterates through the list of resident tasks, looking for ones that belong to Schizo. This can be determined by examining the exception port for each task to see if it is a proxy or a principal. A proxy indicates that the task belongs to a server on a remote node. For each such task found, an exception message is constructed that carries the new exception type `EXC_MIGRATE` and is synchronously sent to the exception port. A synchronous message is used in order to delay further activity until the exception has been completely processed, and the task in question has been migrated to another node.

Once all tasks have been migrated away to other nodes, a number of steps are necessary to ensure that there is no residual state left on the node, and that other nodes in the cluster will no longer attempt to communicate with the down node. These steps include:

1. Delaying while any pending messages that are waiting to be sent have time to be completely sent and received.

2. Finding all of the forwarding proxies and eagerly telling the remaining nodes in the cluster where the true destination for the port is located. Normally, forwarding chains are collapsed lazily. This has not been implemented, but a possible strategy would be to pass a list of port identifier - forwarding node

pairs to each node in the cluster so that they may redirect matching proxy ports.

3. Notifying other nodes in the cluster that a node is about to leave the cluster. A new kernel to kernel RPC has been added that indicates which node is about to leave the cluster. This prevents applications, such as the Schizo server, from trying to contact the node after it has gone down.

5.2.3 Task Migration

The task migration facility [11] used by Schizo was implemented as part of an independent research project at the University of Kaiserslautern. This section is intended to provide context for the discussion of task migration in the following chapter.

The task migration implementation includes two changes to the microkernel that address shortcomings in the microkernel application programming interface. The first problem stems from the fact that, as previously discussed, the kernel request port for tasks and threads are not true ports. Since the receive right for these ports cannot be extracted from the microkernel, it is impossible to migrate them to another node; messages will continue to be delivered to the node where the receive right is. To solve this problem, two system calls have been added that interpose the kernel port for tasks and threads with a port for which a user process holds the receive right. Messages sent to the kernel port of a task or thread that has been interposed arrive at the process holding the receive right; in this case the Schizo server. Once the migration has been completed, the ports are interposed again on the remote node. The remote microkernel now holds the receive right, and all of the messages Schizo received in the interim can now be forwarded to the remote node.

The second problem results from constraints imposed by the Mach paging system. Shared and externally mapped memory regions cannot be transferred at user level since the default pager and the inode pager are designed to interact with only one kernel at a time. Nor does the NORMA distributed memory system does handle these types of memory regions. [1] The solution takes the form of a new system call that exports a capability for a memory region that can be used to map the region on the destination node. The system call takes an address, and performs a lookup to find the associated kernel memory object. If the object has not been created, it does so. A send right for the object is then returned to the calling process which can be used to map the region using `vm_map`. This allows a user process to map the region into a new task on a remote node, then deallocate the mapping in the original task. The remainder of the task migration implementation is contained in the Schizo server, and is described in Section 6.2.2.

[1] Future versions of the NORMA microkernel will handle this properly.

CHAPTER 6

SCHIZO SERVER

As previously mentioned, the Schizo server makes use of the multiple personality capability provided by the Mach microkernel. Each node in the cluster is running an OSF/1 server, and at the same time, is part of the Schizo distributed system. The Schizo server process runs on the core node, but communicates with all of nodes in the cluster via Mach IPC primitives. The Schizo server only communicates with the other microkernels. It completely ignores the remote OSF/1 servers. To run a program in the distributed system, it is necessary to log into the core node, and hand the Schizo server a program invocation specification (a command line). Schizo then starts the program on an available node in the cluster, or if it fails to find one, starts it on the core node. Schizo is then responsible for tracking the progress of the program, arranging to migrate it when necessary, and cleaning up when the program exits. Schizo is composed of two main activities: the *main loop* and the *scheduling loop*, each running in its own thread. The remainder of this chapter describes these two activities in detail.

6.1 Main Loop

A number of operations must be performed before the Schizo server can begin accepting requests. Foremost among these is to create a port on which to listen for requests. This port is then checked into the Mach *name server*, a simple naming facility that allows Mach programs to exchange port rights by using an agreed-upon string to rendezvous. A second port is then allocated on which to listen for dead-name notifications (see Section 6.1.4). A port set (collection of ports that can be treated as a single unit in Mach message system calls) is created, and both of these ports are added to the port set. A number of locks and mutexes are initialized, and lastly, the scheduling system is initialized. The main loop is then invoked, which listens for messages arriving on the port set, dispatching them to one of four subsystems. These subsystems handle Schizo requests, fork system call requests, exceptions, and notifications. Each module will be discussed in turn.

6.1.1 Schizo Requests

The Schizo subsystem handles requests coming from client programs that connect to Schizo via the port that was checked into the name server. The MIG definitions for two of the requests can be found in Figure 6.1. MIG uses definitions such as these to construct a dispatch function that will call the appropriate routine as indicated by the MIG-assigned ID of the request. The two routines shown in the figure are `schizo_join`, which allows a process to register with Schizo in preparation for starting a program, and `schizo_migrate`, which allows a Schizo process to be migrated by its owner.

6.1.1.1 schizo_join

The current implementation of Schizo depends on a utility program (called `batch`) to handle the work of starting programs. The batch utility is invoked with the command line of the program to be started. Its first operation is to register with Schizo by sending a `schizo_join` request to the server. Once registration is complete, the batch utility issues a fork system call to create a new process. As a result of registering with Schizo, the underlying Mach task for the new process will (potentially) be created on another node. Lastly, the batch utility issues an

```
routine schizo_join(port        : mach_port_t;
        in          hint        : int;
        in          task        : mach_port_move_send_t);

routine schizo_migrate(port     : mach_port_t;
        in             pid      : int;
        in             node     : int;
        out            status   : int);
```

Figure 6.1. MIG definition file for Schizo requests.

exec system call to start the program. Since the underlying task is now on another node, the program actually runs on that node.

The most important aspect of the registration process is to arrange for Schizo to intercept any fork system calls issued by Schizo processes. This allows Schizo to determine on what node in the cluster the underlying Mach task should be created. Schizo issues the necessary microkernel system calls, and then allows the fork to proceed normally. The fork system call could have been intercepted in one of two ways. The first alternative was to replace the BSD request port in each Schizo process with another port for which the Schizo server holds the receive right. This would ensure that all system calls issued by a Schizo process come to the Schizo server first. Schizo can then read the request, and forward all nonfork system calls on to the BSD server. This method is both conceptually simple and trivial to implement. However, its effect on performance would be large since all system calls would involve an extra message send and receive, as well as additional processing in the Schizo server.

The second alternative, and the one chosen for the prototype, was to modify the OSF/1 emulator such that the fork system call is sent to a different port than the BSD request port. This is accomplished by creating a port for which Schizo holds the receive right, and changing the newly created Mach task's exception port to be a send right for the new port. Since all OSF/1 processes start executing in the emulator in an initialization routine, it was easy to arrange for the exception port to be read, and saved away for later use. Whenever the process issues a fork system call, the request message is sent to this port instead. Additional code in the emulator re-reads the value of the exception port after subsequent forks. In this way, all children of the original process are scheduled by Schizo, and have a separate port to which fork system calls are sent.

6.1.1.2 schizo_migrate

The other Schizo request shown in Figure 6.1 is the `schizo_migrate` request. This request allows a user to migrate a process using its process ID number. This is mostly a debugging feature, but on occasion, is useful for doing load balancing by hand. Typically, processes are migrated by the scheduling subsystem (described in Section 6.2) when they fail to make adequate progress, or by the exception subsystem (described in Section 6.1.3) when a node is about to be rebooted.

6.1.2 Fork System Call Requests

As discussed above, the Schizo server intercepts the fork system call requests made by its processes. After arranging for the new process to be created on a (possibly remote) node, the request message is forwarded to the OSF/1 server which does the actual work of creating the new process.[1] In this way, the Schizo server is acting as both a client and a server of the fork request; hence it must link in both the client and server stubs created by MIG. Figure 6.2 shows the MIG definition file used by Schizo. To avoid duplicate symbol definitions, the server routine (the one accepting requests from the user process) is prefixed with `schizo_`, while the client routine (the one called to forward the request to the OSF/1 server) is prefixed by

[1]Throughout this chapter, all occurrences of the term "OSF/1 server" refer to the one that "owns" the Schizo server process.

```
ServerPrefix schizo_;
UserPrefix   bsd_;

skip;   /* bsd_execve */
skip;   /* bsd_exec_args_set */

routine fork(proc_port          : mach_port_t;
        out  interrupt          : boolean_t;
        in   new_state          : thread_state_t;
        out  child_pid          : int);
```

Figure 6.2. MIG definition file for BSD fork system call.

`bsd_`. The rest of the file is identical to the MIG definition contained in the OSF/1 server. The two `skip` statements are needed since Schizo does not need definitions for those routines, but must ensure that the ID number of the request is advanced properly by MIG.

When Schizo receives a fork request message, a `process` structure (see Figure 6.3) is allocated to hold information about the new process. The node on which to schedule the process must then be determined. If the tree depth of the new process would exceed some threshold (a command line argument to the Schizo server), the process is scheduled on the same node as the parent. This is helpful for process level parallel programs that need to limit fan-out of its children. Otherwise, a call is made to the scheduling subsystem to determine where in the cluster to place the process. Once the node is known, a microkernel call is issued which tells the NORMA system on which node to place child tasks of the task making the fork request. The fork request is then forwarded to the server which creates the UNIX process and underlying Mach structures. When the OSF/1 server issues the necessary `task_create` call to create the child task, the NORMA system arranges for it to be created on the requested node. Forwarding the fork request message to the OSF/1 server in this manner maintains the UNIX parent-child relationship of

```
typedef struct process {
        queue_chain_t   exc_chain;   /* For exception port hash */
        queue_chain_t   bsd_chain;   /* For BSD request port hash */
        mach_port_t     exc_port;    /* Exception port */
        mach_port_t     task_port;   /* Task kernel port */
        mach_port_t     bsd_port;    /* BSD request port */
        mach_port_t     thread_port; /* Thread kernel port */
        struct mutex    proclock;    /* A lock */
        int             node;        /* Node scheduled on */
        int             pid;         /* Unix PID */
        queue_chain_t   proc_chain;  /* List of tasks on a node */
        queue_chain_t   migr_chain;  /* Migration queue */
        int             badness;     /* Stall count */
        int             canmigrate;  /* FALSE for "joined" tasks */
        int             migrations;  /* How many migrations */
        int             level;       /* Recursion level */
        int             usage;       /* CPU usage at last check */
        int             state;       /* Run state at last check */
        int             hint;
} *proc_t;
```

Figure 6.3. Process Structure definition.

the new process; the parent of the new process is the one that issued the original fork request, not the Schizo server.

Once the server has completed the fork operation, a reply message is sent back to Schizo indicating that a new UNIX process has been created. This process is in a suspended state. Schizo then initializes the process structure, allocating a new exception port for the underlying Mach task. The fields `exc_chain` and `bsd_chain` are used to link the structure into two hash chains. All of the requests received by Schizo on behalf of the process will come in on either the `exception` port (see Section 6.1.3), or will come in on the BSD request port (see Section 6.1.4). The hash tables are used to quickly map from the port value to the associated `process` structure.

A number of other values are also cached in the process structure. The `task_port` and `thread_port` fields store the value of the kernel port for the underlying Mach task and thread of the new process so that they may be manipulated by Schizo. The UNIX process ID and the node on which the process is currently executing are also recorded. The `proclock` field is initialized to the unlocked state. This lock is used to prevent concurrent access to the process structure by Schizo's two threads (the main loop and the scheduling loop).

Once the initialization is complete, the scheduling priority of the process is depressed, and the process is resumed. Control then returns to the fork system call stub within the process.

6.1.3 Exceptions

The exception subsystem is responsible for receiving asynchronous exceptions sent from the microkernel to the task exception port of the task underlying the UNIX process. An example of such an exception message would be one informing the OSF/1 server of a segmentation violation or bus error. Normally, these messages are sent directly to the OSF/1 server for processing. Since the Schizo server has replaced the task exception port so that it can intercept fork requests, it is also the case that *all* exception messages for the task are received by Schizo. When Schizo receives an exception message, it uses the value of the exception port to map to the corresponding process structure. It then looks at the type of the exception, forwarding the message to the OSF/1 server if it is not an `EXC_MIGRATE` exception (see Section 5.2.2). This is done rather easily since all UNIX processes use the same exception port, so it is simply a matter of forwarding the message to the exception port of the Schizo server. If the type of the exception message is `EXC_MIGRATE`, the Schizo server must arrange for the process to be migrated away from the node it is currently executing on. Further, since the machine is going to be rebooted, the migration must be performed eagerly to ensure that all of the pages belonging to the process are moved across before the machine is no longer able to respond. This is described more fully in Section 6.2.2.

6.1.4 Notifications

The Schizo server needs to be informed whenever a process terminates so that it can release the resources it has associated with the process. The scheduling system also needs to know when a process terminates so it can factor that into future scheduling decisions. Since Schizo processes are not actually children of the Schizo server, the usual UNIX methods for the determine when a child process has exited (the `wait` system call) cannot be used. Fortunately, Mach provides the necessary capability in the form of a notification that a particular port has been destroyed, called a *dead name notification*.

When Schizo creates a new process, it makes a request to the microkernel that it be notified (via a message sent to the port Schizo allocated during its initialization phase) when the BSD request port for that process is destroyed. Later, when the process exits, the OSF/1 server will destroy the port as part of the normal cleanup mechanism. The microkernel will then send a dead name notification message to Schizo. When Schizo receives this message, it uses the BSD request port contained in the message to map to the appropriate process structure. Schizo then deallocates the request port, tells the scheduling system that the process has terminated, and deallocates the process structure.

6.2 Scheduling System

The scheduling system is responsible for determining where to place processes when they are created or when they need to be migrated. In addition, the scheduler also determines when a process needs to be migrated due to lack of progress. The scheduling system runs in a separate thread, periodically waking up to query the state of all the nodes in the cluster, and the state of each Schizo process. If a node has left the cluster since the last iteration, the microkernel will return an error to Schizo, which then removes that node from its internal data structures. The following sections describe the scheduling system in further detail.

6.2.1 Scheduler

When Schizo starts up, it calls an initialization routine that asks the microkernel for the *host privilege port* of each machine in the cluster. The host privilege port for a node is used to perform special operations or request information about the node. Since the Schizo server runs as a "root" process, it can request the host privilege port for the node it is running on from the OSF/1 server. Schizo can then use that as a capability that allows it to request the host privilege port for each of the other machines in the cluster. The Schizo server uses the host port to request load information from each node in the cluster. This load information is computed by the microkernel during each scheduling interrupt and is an array of integers representing the number of processes in the run queue, averaged over the last 30 second, 60 second, and 5 minute intervals.

The scheduling system uses a delay loop that performs a "sleep" operation on each iteration. The duration of the sleep is a configurable option to the Schizo server, and has a default value of three seconds. Each time the scheduling system wakes up, it requests the load information from each node, and saves it in a node specific data structure. Although crude, this information is still a good indication of how busy a node is, and whether it has enough idle cycles to handle another process. When Schizo needs to schedule a process, the scheduler looks for the node that is the least busy. If all of the nodes are too busy, the process is temporarily scheduled on the core node.

6.2.2 Utilization

In addition to collecting load information, Schizo tracks the progress of existing processes, moving them from overloaded machines to underutilized machines. This includes processes that were previously scheduled on the core node because an underutilized machine could not be found. Each time the scheduling loop wakes up, it goes through the list of Schizo processes, requesting its thread status information. The status information includes two fields, `cpu_usage` and `cpu_state`, which are used to determine if a process is making adequate progress. The `cpu_usage` field is a measure of the amount of CPU time the process is using. The `cpu_state` is a collection of bits that indicate the status of the process. For Schizo, several states are of interest:

- The process is receiving a lot of CPU time. This is the ideal situation for a process.
- The process is receiving very little CPU time, and is in the `BLOCKED` state. This indicates that the process is blocked, perhaps waiting for an I/O request to complete. There is reason to expect that once the I/O request is satisfied, the process will continue running.
- The process is receiving very little CPU time, and is in the `RUN` state. This indicates that the process wants to run, but cannot get CPU time, most likely because local processes are competing for CPU.

This last case is the one that causes Schizo to react. These idle processes are candidates for being migrated to other nodes in the system, so that they may potentially make forward progress. For each one, Schizo determines if there is another candidate node, and if so, migrates the process to that node. If an idle node cannot be found, the process is migrated to the core node in the hope that it can make forward progress there.

There are two problems with this approach. The first is potential *ping-ponging* as the result of transient load increases on the remote machines. The second is another of form of ping-ponging in which a process on the core node is migrated to an idle node, followed by the migration of a process from a busy node to the core node because there no other free nodes. Instead, a single migration should be done, moving the process from the busy node to the idle node. In addition to the obvious waste of cycles spent in extra migrations, there is also the impact that the workstation owner might see as his or her machine responds to page faults, messages, etc. when the migrations are performed.

To avoid these problems it was necessary to extend the algorithm slightly. Instead of migrating processes as soon as they cease to make forward progress, a counter is employed that keeps track of how long the process has been stalled. When the time exceeds a constant threshold, the process becomes available for migration. Second, processes on the core node are considered only after processes on remote nodes. This ensures that processes are migrated from one remote node to another before ones on the core node are migrated. This prevents the second form of ping-ponging described above.

6.2.3 Migration Module

The migration module was implemented by Milojicic [11], and subsequently modified for Schizo to do eager migration in addition to lazy migration.[2] Other than the small modifications made to the microkernel (see Section 5.2.3), the migration module is implemented entirely at user level. This is possible because any process with a send right to the kernel port of a task can manipulate that task in almost any way it wishes. The Mach abstractions provide a simple and elegant means by

[2] A kernel-resident implementation by Dejan Milojicic *does* eager migration, but was not available.

which a task can be migrated to another node. The general task migration strategy is:

1. The task and its associated threads are suspended, and then aborted to put them into a clean state. This ensures that any system calls in progress will be restarted when the thread is resumed.

2. Interpose the kernel ports of the task and thread(s).

3. A new task and appropriate number of threads is created on the destination node. The state of the threads is transferred using the `get_thread_state` and `set_thread_state` microkernel calls. Lastly, some task specific information such as priority, exception port, etc. are transferred.

4. The address space is transferred.

5. Ports and capabilities are transferred using the `mach_port_extract_right` and `mach_port_insert_right` microkernel calls. It should be noted that the extraction and insertion of the rights causes the NORMA system to do the actual migration of the rights.

6. Interpose again the kernel ports of the task and thread(s) on the destination node.

7. The task and its threads are resumed.

Of these, address space transfer warrants a more detailed description. The address regions of a task can be roughly divided into *internal* regions (such as those allocated by `vm_allocate`), and *external* regions that are the result of mapping an object. Internal regions can further be divided into those that are shared with other tasks, and those that are not. These last regions are the easiest to transfer. When the destination task is created, it is created as a child of the source task, thus all of the nonshared internal regions are inherited automatically as a result of the normal microkernel task creation mechanism.

Shared internal regions and externally mapped regions are harder to transfer because Mach's builtin pagers are designed for only a single kernel, and the current NORMA DSM system does not yet allow internal regions to be read/write shared between nodes.[3] However, the modifications made by Dejan Milojicic to the microkernel provide for a workaround. For these two types of regions, a capability representing the underlying memory object is exported using the new microkernel system call `memory_object_export`. This capability is then used to `vm_map` the object into the destination task on the remote node. The proper protection and inheritance values are then transferred using the `vm_inherit` and `vm_protect` system calls.

Once the destination task is resumed, it continues running at the point at which it was suspended. Any threads that were within a Mach system call automatically restart the call. As the thread runs, it accesses its pages of memory for the first time since the migration. These pages still reside on the node where the original task was before the migration, and are faulted across the network as they are touched. For this reason, this form of migration is designated as *lazy*. This method is preferred for load shifting migrations as it spreads the work of migration over a longer period of time. The workstation owner would likely notice the extra page faults if they occurred as a burst. Section 7.4.2 discusses the effects of these remote page faults on the workstation.

Processes that are migrated as the result of an `EXC_MIGRATE` exception need to be handled differently. The memory regions for these processes must be moved eagerly to the destination node before the source node goes down. To do this, the migration module was modified to touch all of the pages in the destination task

[3]Internal regions *can* be shared Copy On Write between nodes.

using `vm_read`. Each page is read into a buffer from the remote task, causing the page to be faulted across the net into the destination task, so that `vm_read` may then read it into the provided buffer. The result is that all of the pages are faulted across to the destination node in a burst. This has an obvious performance effect on the source node, and will be discussed in the following chapter.

CHAPTER 7

RESULTS

A series of experiments have been performed and their results are described in this chapter. These experiments are intended to:

- characterize the kind of applications that are appropriate for the Schizo system;
- demonstrate the functional capabilities of the system;
- quantify the effects of remote execution and migration on the applications;
- quantify the effects of the guest applications on the workstation tasks;
- show that Schizo does not saturate the underlying network.

The Schizo prototype has not yet been deployed on personal workstations, so these experiments were performed using two clusters of dedicated machines. One cluster consists of 4 HP 730 (76 Dhrystone 1.1 MIPS) workstations with 32 MB, ethernet interconnect, and local disks on SCSI controllers. The other cluster consists of 4 PCs with I486 DX2/50 processors (22 Dhrystone 1.1 MIPS), 16 MB, ethernet interconnect, and local disks on SCSI controllers. All of the machines are running version 13.16 of the NORMA microkernel, and version 1.0.4 of the OSF single server. To minimize external network traffic effects, the two clusters of machines coexist on a single ethernet subnet with only one other host which serves as a gateway to the outside world.

7.1 Characterizing Appropriate Applications

Two characteristics of appropriate applications can be quantified: minimum practical size, where size is a function of compute time and program text size, and compute-intensiveness, which is a function of the kind and frequency of system calls it employs.

Table 7.1 displays the times for `fork` and `exec` system calls executed remotely (i.e., the process calling `fork` is on the same node as the OSF/1 server, but the child is to be created on a different node).[1] The times for the local case are also provided for comparison. The paging activity involved is minimal, as the child's pages are copied on demand (lazily) and the program being exec'ed by the child is a one-page program. The exec'ed program simply exits. Table 7.2 shows the time to fault a page of memory across the network. The test program ensures that the pages in question are already memory resident by touching them. It then forks, with the child assigned to a different node. After the fork, the pages are touched again by the child. Taking these two sets of measurements together, it can be postulated that "running" a program on another node will (on the I486) cost at least .56 seconds plus approximately 15 milliseconds per page of text. This constitutes a reasonable, but by no means maximal, lower bound on the compute time that a Schizo application would need to consume to be considered "practical."

Table 7.3 lists a selection of system calls and the elapsed time for each as a local or remote call. The `getpid` and `sbrk` calls are handled by the emulator in the calling process' context; thus their times are essentially the same in both cases. `nop` is an "empty" call of no arguments with no associated actions; it is used simply to measure the communication and context switch time involved in a system call. `stat` takes roughly five times longer in the remote case, and `read` and `write` are each two

[1] `Fork` and `exec` may be expected to improve in the OSF/1 AD server, which has provisions for distributed process management.

Table 7.1. Times for fork and exec system calls.

	Fork Time		Exec Time	
Architecture	Local	Remote	Local	Remote
HP730	.014	.330	.029	.700
I486	.022	.270	.032	.290

Times are in seconds.

Table 7.2. Time to fault a page across the network.

Architecture	Fault Time
HP730	6480
I486	15240

Page size is 4096 bytes. Times are in microseconds.

Table 7.3. Remote and local system call timings on HP730 and I486 clusters.

System Type	HP730		I486	
System Call	Local	Remote	Local	Remote
nop	100	1240	348	3170
getpid	13	13	20	20
read (4K bytes)	812	83170	1120	116830
write (4K bytes)	496	49900	659	50020
stat	269	1450	769	3710
sbrk	22	22	36	39
chown	273	1380	752	breaks
create	555	133430	1376	166950
signal	375	245680	912	300330

Times are in microseconds.

orders of magnitude slower in the remote case. Coupling this with the rather large times for remote `fork` and `exec`, it can be concluded that a program with small amounts of computation and large amounts of I/O is currently an inappropriate application for Schizo. As can be seen in Table 7.4, the time to compile a program remotely is much greater than the local case. For small programs such as these, the amount of I/O that is performed is much greater than actual computation. More generally, any program performing large amounts of reading and writing and making frequent use of signals is an inappropriate application.

The next experiment was performed using a ray tracer implemented in Scheme[25]. The ray tracer is a long-running, compute intensive, single-threaded program. Once it completes an initialization phase, it computes intensively; the length of the computation is easily adjustable by specifying the desired number of pixels. Test scenarios were developed with short (several minute) computations, and then with much longer runs of the same program. The program is also a "real" application, rather than a synthetic benchmark. Table 7.5 lists the local and remote elapsed times for small and moderate sized runs of the ray tracer. The reason for the (apparently anomalous) speedup achieved by executing the program remotely in unknown. Though relatively large, the program's virtual memory size did not exceed the available real memory and little paging occurred after the relatively short initialization phase. This program is used in later tests described in Sections 7.3 and 7.4.

Table 7.4. Remote and local C compile times on the I486.

Test case	Lines	Local	Remote
null.c	0	1	8
cpio.c	1400	4	32

Times are in seconds.

Table 7.5. Remote and local ray trace times on the I486.

Test size in pixels	Local	Remote
1156	12.9	13.3
13000	96.8	88.5

Times are in minutes.

The results presented thus far indicate that Schizo would not be suitable for running system-call intensive process-level parallel applications such as parallel `make`. However, it *is* reasonable to expect that compute intensive parallel applications would exhibit performance improvements when run on more than one node. To show this, an experiment was performed using a program that implements an exhaustive tree search algorithm. The program, called `triangle`, computes all the solutions for a simple board game whose object is to remove all but one peg from a board containing some number of pegs, and 1 hole. A peg is removed when it can be leap-frogged by another peg. Table 7.6 shows the elapsed times for a 20 peg triangle program on both the I486 and the HP730. The "depth" indicates to what level the tree was searched; searching the entire tree would take an estimated 200 years on the HP730! Times for the sequential version and for a parallelized version are given. The parallelized version uses three child processes to divide the work

Table 7.6. Triangle program timings on HP730 and I486 clusters.

System Type	HP730			I486		
Depth	One Node	Three Node	Efficiency	One Node	Three Node	Efficiency
10	111	48	77.1%	198	79	83.5%
11	805	292	91.8%	1511	544	93.2%

Times are in seconds.

up, synchronizing at completion to report the results. The efficiency measurements indicate that compute intensive parallel applications would indeed be suitable for Schizo.

7.2 Demonstrating Functionality

The desired functional behavior for Schizo encompasses the following:

- the ability to run programs on participating nodes, with automatic node selection by Schizo, and without requiring any modification, recompilation, or relinking of the application;
- the containment of any child processes within Schizo's purview;
- the migration of processes as necessitated by increases in load on host machines;
- the migration of a process as necessitated by the orderly shutdown of the host machine on which it resides.

To demonstrate this, Utah Common Lisp (UCL) was started via Schizo. It ran a simple loop,[2] which allocated and discarded objects on the heap, coincidentally causing periodic garbage collections. With the help of some artificial load generated on the host workstations, the UCL program moved from node to node making and collecting garbage. The ability to perform most of the functions listed above (except for handling shutdown), is implicit in the experiments outlined in the next two sections.

7.3 Performance Effects on the Guest Application

To prove useful, the Schizo system must ensure that programs entrusted to it make reasonable progress when resources are available within the system. The

[2]Of course, in Common Lisp, a simple loop can touch hundreds of text and data pages.

location at which these resources are available is expected to shift as workstation owners come and go and as they shift from one task to another. In addition, the user should be able to form a reasonable estimate of completion time for his program. This estimate would include assumptions about the availability of resources and expected "normal" time to completion for the task. It is assumed that the user can observe the load on the workstation base before forming his expectations, or that he has a model, perhaps from prior experience, of the load to expect.

There are other effects on runtime that are artifacts of running on a distributed system. Some of these effects were addressed earlier, in particular system call and paging times. When migration is taken into account, it can effect runtime both directly, in the form of unavoidable paging delays and time lost during migration, and indirectly, in the form of additional page fault latency due to load on the machine the task is migrated *from*. Schizo will have selected a lightly loaded destination for the task, so the process will usually be able run immediately upon migration.

Table 7.7 shows the results of an experiment intended to quantify the latter cost. An artificial load program was constructed that repeatedly loops, imposing a high compute load for a given number of seconds (the "load" column in the table) and then becoming quiescent (sleeping) for some other specified number of seconds.

The experiment was run with three nodes, one serving as the Schizo core and two as participating "workstations." The load program was started on the two workstations, 180 degrees out of phase, viewing the loaded/idle states as a clock. The test program (the ray tracer described earlier) was started via Schizo, which placed it on whichever workstation was currently idle. When the load program on that machine entered its load imposition phase, the ray tracer was migrated to the other machine. The time to completion and number of migrations for varying migration intervals is reported. As can be can seen, the ray tracer, which took

Table 7.7. Effects of source host load on a migrated task.

Load Period	Migration Interval	Elapsed time in minutes	Migrations
30	180	20.36	6
30	240	18.46	6
30	300	16.66	4
0	150	16.88	6
0	210	15.81	4

approximately 13 minutes as a normal, local job, is significantly degraded when the node it migrates from is very busy. This can be attributed to time spent fetching pages from the busy machine. As the migration interval grows, more time is available to work with the pages once they are faulted over. This conclusion is borne out by the last two cases, where no load was imposed on the source machine; the degradation is much less.[3] The only time lost is due to the actual migration (approximately 2 seconds to transfer the task and thread state on the I486), and paging delays as pages are transferred across the network.

7.4 Performance Effects on the Host Workstation

The performance effects of Schizo tasks on the host workstations and their tasks has already been identified as an important consideration. Schizo attempts to mitigate these effects in two ways:

1. by depressing cpu priority;
2. by migrating a task to another host when the current host becomes too busy.

The following experiments demonstrate how effective the first strategy is in minimizing the impact of guest tasks, and what, if any, lingering effects a migrated job has on the source node.

[3] Since no load was imposed on the source machine, the migrations were artificially induced by sending migrate messages to the Schizo server.

7.4.1 Limiting CPU Consumption

An experiment was conducted in which a script of "typical" user's activities, including editing and sleeping, was run as a foreground task. As an example of a Schizo guest task, a Cholesky factorization was run as a background task to determine its effects on the time required to complete the script of user activities. Table 7.8 displays the results. A baseline time was established corresponding to a private workstation with no Schizo tasks. Then a "worst case" time was obtained by running the background task with no controls, i.e., as another foreground task. An appreciable degradation occurs in this case. Next the background task is run with its cpu priority depressed. As the results show, the degradation of the foreground task lessens. This indicates that depressing CPU priority helps in preventing Schizo processes from having an adverse effect on local tasks. However, using depressed CPU priority alone can cause page thrashing. Experiments conducted in [21] show that depressing virtual memory priority of Schizo tasks further reduces the amount of degradation in the foreground task. The results of these experiments are given in the last two rows of Table 7.8.

7.4.2 Residual Effects

Section 7.3 shows that the lazy faulting of pages that occurs after a migration has a noticeable effect on the progress of the migrated job. The experiment was

Table 7.8. Effects of background task on foreground activity on the HP730.

Background Activity	Foreground task Elapsed time in seconds	% Increase in Time
None	133	–
Cholesky, no control	151	13.5
Cholesky, depressed cpu	145	9.0
Cholesky, depressed VM	141	6.0
Cholesky, depressed cpu and VM	136	2.2

modified slightly by adding a counter in the busy loop of the artificial load program, as a measure of the "work" it performed in the given number of seconds. Table 7.9 shows the amount of work performed by the load program in an idle Schizo system (no migrating tasks) and the amount performed immediately after a migration occurs. The impact of the migration on the workstation is significant. It is also intuitively reasonable. Page faults, including those caused by the migrated process, are serviced by kernel threads, which run at a higher priority than user threads, There may also be a second order effect if the pages for the migrated task have been paged out to disk and need to be paged in to satisfy the faults. No attempt has been made to isolate these effects, if present. These results lend support to the intuition that some additional work on depressing Schizo activity will be required to meet the goal of nonintrusiveness.

7.5 Performance Effects on the Network

The underlying network used in these experiments is a ten megabit per second Ethernet. Ten Mbps represents the theoretical maximum bandwidth, not the raw speed of the network. In other words, two machines rarely transfer data between them at this speed. Rather, many machines on the network can generate individual loads that consume the total bandwidth. Given this framework, it is necessary to show that Schizo does not generate individual loads that consume an excessive amount of bandwidth. Doing so would have an adverse effect on all machines on the network, and could potentially make Schizo less attractive to workstation owners. This discussion examines the principal areas in which Schizo causes network activity.

7.5.1 Paging

The single largest component of network activity caused by Schizo is remote paging, both when a process is started and whenever one is migrated due to

Table 7.9. Amount of "work" performed in 30 seconds on the I486.

	Counter value after 30 seconds elapsed time	% of baseline
Baseline (no Schizo tasks)	245624169	–
Schizo task faulting pages	162747867	66.2

lack of progress. As previously mentioned, the NORMA microkernel uses UDP packets to transfer data; a 4096 byte page will require three UDP packets. With header information and the acknowledgement message, a total of 4268 bytes will be transferred across the network. In the absence of any software overhead, this amount of data would take 427 microseconds to transfer. As the results in Table 7.2 show, the time to fault a page is considerably more; 93 percent of the time can be attributed to software overhead. The conclusion is that although the total number of pages transferred might be large, the amount of bandwidth consumed is rather small due to the amount of software overhead necessary to transfer those pages. Although there is room for improvement in the software, it is unlikely that the amount of overhead could be reduced to the point where it would be possible to saturate the network. Additionally, the results in Table 7.2 are for "warm" pages; disk latency resulting from loading nonresident pages would further throttle paging activity.

7.5.2 Load Information

As Section 6.2 describes, the Schizo scheduling loop periodically wakes up to request load information from each node in the cluster. The amount of data transmitted for each request is 548 bytes; 168 bytes for the request message, 188 bytes for the reply message, and two acknowledgement messages of 96 bytes each. Table 7.10 shows the number of microseconds it takes to synchronously make the request and receive the reply. In the absence of software overhead, the network

Table 7.10. Roundtrip time for load information request.

Architecture	Microseconds
HP730	1033
I486	3375

transfer time should be 55 microseconds. Since the actual time is 19 times greater (for the HP730), it is safe to assume that these periodic bursts of message traffic should have little effect on the network. In addition, these times indicate that the number of nodes can be scaled up into the hundreds without having to increase the scheduling period; the default 3 second period permits (on the I486) 900 nodes to be queried on each iteration.

CHAPTER 8

CONCLUSIONS AND FUTURE RESEARCH

The Schizo prototype has demonstrated that Schizo can effectively distribute processes to participating workstations. Further, it can automatically shift these processes throughout the system as the load on participating workstations changes. The class of applications that are suitable for Schizo have also been identified. These are long running, computationally intensive programs that are not time-completion critical, as well as compute intensive process-parallel applications. The large communication costs of the Mach microkernel effectively rule out system-call intensive applications such as parallel `make`. Future research should include investigation into why the communication costs are so large, and whether they can be reduced. Further, the possibility of hosting Schizo on other operating systems should be explored.

The fragility of the underlying OSF microkernel, as well as a lack of user level support (backups, mail, etc.), has prevented Schizo from being deployed on personal workstations. Great effort was made in trying to isolate and fix the problems in the underlying microkernel, but the system never reached the level of stability that is necessary for deployment on desktops. However, the experiments that have been conducted thus far indicate that Schizo's goals of efficiency and ease of use have been met. Future research should seek to verify these experimental results by deploying Schizo on personal workstations. Further, the goal of security can be verified only when the system is running real applications under a more varied set of conditions and users.

The goal of reliability has not been met by the Schizo prototype. While Schizo can migrate processes from a workstation that is undergoing planned shutdown, there are occasional problems with system lockup. These problems are most likely due to residual dependencies on the node after it has left the cluster. These residual dependencies are a result of algorithms in the NORMA microkernel that depend on distributed state. Future research should attempt to isolate and remove these residual dependencies. In addition, the prototype makes no attempt to deal with system crashes; when a node crashes, the entire Schizo system locks up, and all Schizo processes are lost. This problem needs to be addressed, since the level of fault tolerance dictated by the Schizo design is necessary in order to convince workstation owners to be part of the Schizo system and to convince users to run their applications under Schizo. Workstation owners must be assured that their system is not going to be less reliable when part of Schizo, and users need to be confident that Schizo can run their applications to completion.

The ability to handle heterogeneous nodes is another area in which future research can be done. A first approach would be to allow the memory of nodes to be shared between different architectures. Since a page is ultimately just a collection of bits, paging to a node of a different type should be feasible and might allow applications that were paging to disk to page to other nodes. The next step of allowing forking and migration of tasks across different architectures will require significant operating system and also compiler work (for example, at well defined points, it is possible that a task could snapshot its state and transform that data to another architecture and restart).

REFERENCES

[1] Barrera, J. S. A Fast Mach Network IPC Implementation. In *Proceedings of the Second Usenix Mach Symposium* (1991), pp. 1–12.

[2] Comer, D. E. *Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1991. 2nd Edition.

[3] Cooper, E. C., and Draves, R. P. C Threads. Tech. Rep. CMU-CS-88-154, Carnegie Mellon, 1988.

[4] Forin, A., Barrera, J., and Sanzi, R. The Shared Memory Server. In *USENIX Conference Proceedings* (January 1989), pp. 229–243.

[5] Gao, C., Liu, J., and Railey, M. Load Balancing Algorithms in Homogeneous Distributed Systems. In *Proceedings of the International Conference on Parallel Processing* (1984), pp. 302–306.

[6] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[7] Krueger, P., and R., C. The Stealth Distributed Scheduler. In *Proceedings of the 11th International Conference on Distributed Computing Systems* (May 1991), pp. 336–343.

[8] Leffler, S., McKusick, M., Karels, M., and Quarterman, J. *4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Reading, MA, 1989.

[9] Li, K., and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* (November 1989), 321–359.

[10] Litzkow, M., Livny, M., and Mutka, M. Experience With the Condor Distributed Batch System. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems* (October 1990).

[11] Milojicic, D., Zint, W., Dangel, A., and Giese, P. Task Migration on Top of the Mach Microkernel. In *Proceedings of the Third Usenix Mach Symposium* (April 1993).

[12] Mullender, S., Rossum, G., Tanenbaum, A., Renesse, R., and Staveren, H. v. Amoeba - A Distributed Operating System for the 1990s. *IEEE Computer Magazine* (May 1990), 44–52.

[13] Nichols, D. A. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems principles* (November 1987), pp. 5–12.

[14] Open Software Foundation and Carnegie Mellon University. *MACH 3 Kernel Interface*. Cambridge MA, January 1992.

[15] Osterhout, J., Cherenson, A. R., Douglis, F., Nelson, M., and Welch, B. B. The Sprite Network Operating System. *IEEE Computer 21*, 2 (February 1988), 23–36.

[16] Rashid, R. F. Threads of a New System. (Mach, a Multiprocessor Operating System). *UNIX Review 4*, 8 (August 1986), 37–49.

[17] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W. Overview of the CHORUS Distributed Operating Systems. Tech Report 90-25, Chorus Systemes, April 1990.

[18] Schaar, M., Efe, K., Delcambre, L., and Bhuyan, L. N. Load Balancing with Network Cooperation. In *Proceedings of the 11th International Conference on Distributed Computing Systems* (May 1991), IEEE.

[19] Shapiro, M. A Fault-Tolerant, Scalable, Low-Overhead Distributed Garbage Detection Protocol. In *IEEE 10th Symposium on Reliable Distributed Systems* (October 1991).

[20] Shivaratri, N., and Krueger, P. Two Adaptive Location Policies for Global Scheduling Algorithms. In *Proceedings of the 10th International Conference on Distributed Computing Systems* (May–June 1990), pp. 502–509.

[21] Swanson, M., Stoller, L., Critchlow, T., and Kessler, R. The Design of the Schizophrenic Workstation System. In *Proceedings of the Third Usenix Mach Symposium* (April 1993).

[22] Theimer, M., and Lantz, K. Finding Idle Machines in a Workstation-based Distributed System. In *Proceedings of the Eighth International Conference on Distributed Computing Systems* (1988), pp. 112–122.

[23] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (October 1983), pp. 49–70.

[24] Wilkenloh, C. J., and Ramachandran, U. The Clouds Experience: Building an Object-Based Distributed Operating System. In *USENIX, Distributed and Multiprocessor Workshop* (October 1989), pp. 333–347.

[25] Yih, B., Swanson, M., and Kessler, R. Persistent Immutable Shared Abstractions. In *Second Workshop on Parallel Symbolic Computing* (October 1992). to appear in Springer-Verlag Lecture Notes in Computer Science.

[26] Young, M. W. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Tech. Rep. CMU-CS-89-202, Carnegie Mellon University, November 1989.

[27] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., and Netterwala, D. An OSF/1 UNIX for Massively Parallel Multicomputers. In *USENIX Conference Proceedings* (January 1993), pp. 449–468.

[28] Zhou, S., Wang, J., Zheng, X., and Delisle, P. UTOPIA: A Load Sharing Facility for Large Heterogeneous Distributed Compter Systems. Tech. Rep. CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.