IMPLEMENTATION OF TEAM SHOSHIN : AN EXERCISE IN PORTING AND MULTIPROCESS STRUCTURING OF THE KERNEL

By

HUAY-YONG WANG

BMATH, University of Waterloo, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

March 1986

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date MARCH 6, 1986.

Abstract

Team Shoshin is an extension of Shoshin, a testbed for distributed software originally developed on the LSI 11/23s at the University of Waterloo. This thesis presents a description of the implementation of Team Shoshin on the Sun Workstation. With wide disparity in the underlying hardware, a major part of our initial development effort was to port Shoshin to its new hardware. The problems and design decisions faced by the porting effort and how they were overcome will be discussed. The development of Team Shoshin has provided us the opportunity to investigate the use of multiprocess structuring techniques at the kernel level. We will describe the design and implementation of the proposed kernel multiprocess structure and the rationale behind it. The applicability of the proposed kernel multiprocess structure and its impact on operating system design will be discussed drawing from experience gained through actual implementation.

Contents

List of Figures

Acknowledgement

I would like to thank my supervisor, Dr. Son Vuong, for his advice and guidance on this thesis and Dr. Sam Chanson for reading the final draft.

Thanks are also in order for my colleague and friend, Donald Acton, who was responsible for the porting of the remote communication software and the initial implementation of the object based protocol. His help in testing and debugging Team Shoshin is particularly appreciated. I would also like to thank my fellow graduate students for making my stay at UBC a very pleasant one. Lastly, financial support from the University of British Columbia in the form of University Graduate Fellowship is also gratefully acknowledged.

Chapter 1

Introduction

This thesis describes the implementation of Team Shoshin distributed operating system on the Motorola 68010 [MOTO84] based SUN Workstation[1] interconnected by a 10 Mbps Ethernet.[2] The system is modeled after Shoshin[TOKU83a,TOKU84] with modifications and enhancements to facilitate efficient local and remote operations. The research reported here is based on the work done in porting, redesigning and developing the original Shoshin into its current state, namely Team Shoshin.

The original Shoshin's hardware environment was significantly different from the one we have here. With wide disparity in the underlying hardware, a major part of our initial development effort was to port Shoshin to its new hardware. Throughout the course of porting, we have the opportunity to try out new ideas and experiment with various operating system design techniques. In particular, we investigated the use of multi-process structuring[CHER79a] techniques at the kernel level through the

[1]SUN Workstation is a trademark of Sun Microsystems Inc.

[2]Ethernet is a trademark of Xerox Corporation

introduction of kernel team processes that execute in kernel space.

This thesis will first discuss the problems and design decisions faced by the initial porting effort and how they were overcome. Next, we will describe the design and implementation of the kernel team processes and the rationale behind it. As considerable experience was gained from this implementation, we will discuss the lessons learned and identify some potential problems. The design and implementation process has also led to insights in the applicability of the kernel multiprocess structure and its impact on operating system design. It is from all these lessons learned and experience gained through the implementation of Team Shoshin that form the basis for this thesis.

1.1 Motivations

The development of Team Shoshin on the SUN workstation was initially motivated by the poor performance in the original Shoshin. Part of this problem was due to the original Shoshin's underlying LSI 11/23 hardware that does not have good architectural support for context switching and memory management. This has motivated us to see how far performance can be improved through software restructuring in Shoshin to take advantage of the relatively superior hardware provided by the SUN workstations. This raises performance issues in porting operating system as to how much a limitation the new hardware can impose on system performance on the one hand and on the other hand how far one can improve system efficiency through software restructuring and

enhancements for the target machine.

Further motivations come from the porting process itself which has provided a golden opportunity to actually implement various experiments in operating system structuring and design. The application of multiprocess structuring in message-based operating systems was first introduced in the Thoth operating system[CHER79a] whose multiprocess structure is restricted only to the software layer above the basic kernel abstraction. Little work has been done in extending the multiprocess structuring concept to the kernel layer and until recently only some research work were carried out in this area[RAVI85a][RAVI85b]. It is apparent that the traditional kernel abstraction by its inherently passive nature somewhat enforces localised control as opposed to distributed control. This has motivated us to explore the idea of an active kernel by extending the concept of a process into the kernel. This can be accomplished by factoring the traditional kernel functions into kernel processes that execute within kernel space. The multiprocess structure of the kernel can also be extended to include functions that were ordinarily carried out by system processes for increased efficiency . We believe that this approach coupled with Shoshin's transparent remote and local message-based interprocess communication mechanism will greatly enhance distributed control and functionality.

1.2 Brief Taxonomy of Distributed Systems

In the following section, we will attempt to classify distributed systems in general and introduce some common definitions and terminologies. The main goal of this section is to provide a brief taxonomy of distributed systems so as to facilitate better understanding of the functions of Team Shoshin and its relationship with other existing distributed systems.

Distributed systems refer loosely to a whole spectrum of systems that have some characterization of distribution in their logical and physical features[LAMP81]. On one end of the spectrum we have the *tightly coupled* systems employing shared memory and centralised resource management and on the other end we have the *loosely coupled* systems that have no shared memory and are completely autonomous. Multi-processors like the Cray-1 is a good example of the former and for the latter the best known example is the *heterogenous* network ARPANET. In between the two extremes of the distributed systems spectrum are the multi-processor systems such as StarOS and *homogenous* networks such as MININET. A *heterogenous* network is one where the hosts are different as opposed to a *homogenous* network where all the hosts have the same architecture and run the same operating system.

To characterize Team Shoshin, it is worthwhile to consider resource-sharing networks and distributed-processing networks. As the name resource-sharing suggests,

this type of network allows the user at one location to utilize and share computer resources that may be physically dispersed. A resource-sharing network differs from the centralised uniprocessor system with remote terminals in that it allows users to access several hosts from different locations. A distributed-processing network facilitates problem solving by division of labour or functional specialization so that they can be carried out concurrently by several computer systems, each performing part of the total processing required. Logically, distributed-processing networks are built on top of resource-sharing networks. Resource-sharing networks can be roughly categorized into two major classes based on how computer resources are managed[ELOV74]:

1. Computer-communication Networks
2. Computer Networks

1.2.1 Computer-Communication Networks

Computer-communication networks are usually heterogenous and loosely coupled. From the user point of view, the network is characterized as a collection of several different computer systems with varying capabilities and access protocols. The user of the network is not protected from the idiosyncrasies of different hardware and system dependent interfaces, and the responsibility of managing distributed resources rest solely on the user. To utilize a resource in an computer-communication network, the user must first determine the system on which the resource resides and then establish a connection to that system. Next, all the system dependent commands and access

protocols needed to invoke the resource must be learned and familarised. Thus, the user must be conversant with each different system on the network if he/she wishes to access resources associated with these systems in the network.

1.2.2 Computer Networks

A Computer network is a much more powerful form of network as it is intended to isolate the user from the varying hardware and system dependencies that might complicate access to distributed resources. In effect, the user of a Computer network views the entire network as one large computer system. Here the responsibility of managing distributed resources in the network lies in the network operating system (NOS) that extends the programming environment to embrace remote accesses. In other words, accesses to local and remote resources by users and programs are transparently handled by the NOS. In short, the intend of the NOS is to provide the functions of a traditional single machine operating system on a multi-computer basis.

It is apparent that the development of an NOS is difficult especially when one has to deal with heterogenous networks. An NOS requires additional software to be written for each host so that local computing environment can be extended to handle remote operations with a uniform network wide access mechanism. Various issues have been raised in designing NOS and we will include some of them in this section[LANTZ80].

1. Basic system goal (general purpose versus specialized)

A NOS may be designed to be general purpose and "open-ended" so that it can support a wide range of applications. Alternately, a NOS may be built to solve a specific problem or a class of related problems with real-time and other performance constraints.

2. Implementation base (base level versus guest level)

   In the base level implementation, the system is built from bare hardware components. The advantage in this approach is that the NOS can be specifically designed to function efficiently and effectively with the underlying hardware. Moreover, the attainment of transparent local and remote accesses can be addressed right from the start during system design. In the case of guest level approach, the NOS is built as an extension of existing operating systems and usually utilizes the system services provided by the host operating systems. For transparent local and remote operations to be achieved in the guest level approach, some encapsulation interface must be provided to examine system requests and redirect remote requests to appropriate NOS routines. This will inevitably incur additional performance penalty on local services. Alternately, NOS services can be provided through a set of NOS primitives which are called explicitly by application programs using NOS resources. In this respect, it is only reasonable to take the base level approach if local services can be provided at least as cheap as in local operating system. (non-distributed)

3. Procedure-oriented versus message-oriented

   A procedure-oriented system is characterized by its dependence on some form of shared memory with system services being accessed largely through procedure calls. In contrast, message-oriented system depends on the explicit use of messages for interprocess communication and request for services.

4. Resource abstraction

   Resource abstraction refers to the concept of a meta-resource that possesses certain attributes commonly used and implemented. A meta-resource is a resource created by abstracting common attributes from system services and objects that would permit interchangeable use of different resources by different applications. Since the notion of a service in a NOS can be local or remote, resource abstraction has become an important issue as it helps to eliminate duplication of NOS services.

1.2.3 Classification of Team Shoshin

With respect to the above summary of distributed systems, Team Shoshin possesses the following characteristics:

1. general-purpose system

2. homogenous, loosely coupled, message oriented network operating system

3. base-level implementation on the SUN Workstations (with additional guest-level implementation of Team Shoshin IPC interface[ACTON85a] on other SUNs and VAXes running 4.2BSD UNIX[3]).

4. Resource abstraction over the Ethernet communication interface through transparent local and remote interprocess communication. This is provided as a basic system service.

The guest-level implementation of Team Shoshin IPC interface on SUNs and VAXes running on 4.2BSD UNIX allows UNIX processes to use Shoshin-like IPC primitives to communicate with other Shoshin and UNIX processes residing on remote machines. This has provided Team Shoshin the ability to access large number of services in UNIX with ease. This facility is extremely useful and important especially during the early stages of system development.

1.3 Thesis Outline

Chapter two deals with the impact of the new hardware on the development of Team Shoshin. Various aspects of the Sun Workstation hardware architecture that have bearing on operating system porting and development will be described. The software structure of Team Shoshin is then presented with emphasis on the problems faced and decisions made in the light of the new hardware.

[3]UNIX is a trademark of AT&T Bell Laboratories

Chapter three presents a detail description of Team Shoshin. The system has evolved considerably from its predecessor, the original Shoshin, and we will attempt to document such changes here. A synopsis of the history of Shoshin is first provided, followed by an overview of Team Shoshin and its system and user interfaces.

Chapter four investigates the idea of performing multiprocess structuring at the kernel layer. First, the rationale behind this investigation will be presented, followed by an description of the design and implementation of our proposed kernel multiprocess structure. The impact and applicability of this proposed structure are then discussed based on experiences gained through its use.

Finally, the last chapter concludes this thesis with a summary of the main results and suggestions for future work.

Chapter 2

Team Shoshin on New Hardware

Like all operating system implementations, one has to deal with the peculiarity of the target machine architecture. In this chapter, the impact of the Sun Workstation architecture on the development of Team Shoshin will be discussed. The focus will be on the problems faced and implementation decisions made in the light of the new hardware.

2.1 Hardware Architecture

This section will describe various aspects of the SUN Workstation's architecture that have some bearing on operating system porting and development. Most of the information below are for the SUN-2 Model which are obtained largely after many hours of experimentation of the SUN-2 hardware and partly from an assortment of available SUN-1 Model manuals[SUN82] [SUN83].

2.1.1 Memory management hardware

The memory management unit (MMU) of the SUN Workstation comprises of two context registers, a segment map and a page map. All memory accesses by the SUN's processor are virtual and are translated by the MMU into physical addresses according to the segment and page map entries. The basic unit is a page which is 2 Kbytes (2048). The segment size is 32 Kbytes and each segment maps logically onto a block of 16 consecutive pages. Memory address mapping is done with respect to the current context and up to 8 contexts can be mapped concurrently. The current context is determined by one of the two context registers, namely the supervisor and the user context registers. When the processor is in supervisor state, the current context is determined by the supervisor context register which is always set to the supervisor context (context 0). In the case where the processor is in user state, the user context register is used to determine the current context. Each context has a block of 512 hardware segment map entries giving the maximum logical address space for a context to be 16 Mbytes. In contrast, the number of hardware page map entries shared by all contexts is only 4096, limiting the maximum physical address space that can be mapped to 8 Mbytes. As a result, it will not be possible for a context to have all its segment map entries distinct. The hardware page map entries are grouped into blocks of 16 consecutives pages called segment groups ranging from 0 to 255.

As an illustration of how the MMU works, let us trace through the translation

process that is involved in an memory access [see Fig 2.1]. First the high 9 bits of the

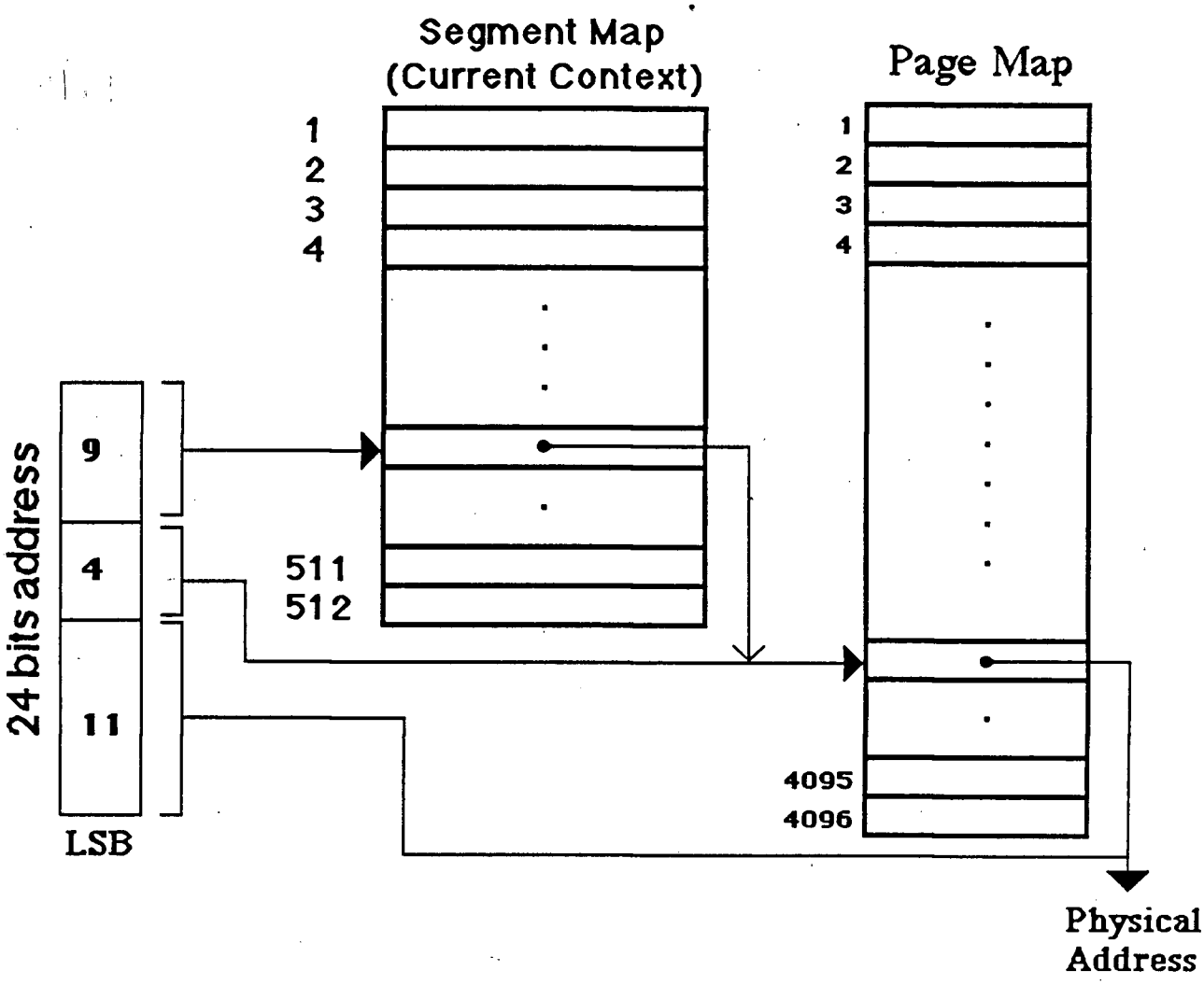

Figure 2.1: Memory mapping in Sun-2

24-bit virtual memory address is taken to index into the segment map of the current context and the segment group number is read from the indexed segment entry. The segment group number is then used to index into the corresponding segment group of the hardware page map entries and the next high 4 bits of the virtual address are used

to determine which page map entry in that segment group to be examined. Finally, the output of the page map entry is concatenated with the remaining low bits of the virtual address to form the corresponding physical address.

2.1.2 Handling of Peripheral Devices

The SUN Workstation Hardware does what is known as "memory-mapped" input/output in that accesses to peripheral devices is done exactly in the same manner as memory accesses[SUN84]. The only distinction between memory and peripherals is the presence of three special address spaces (apart from on-board memory space) that allow peripheral devices to be mapped. These three additional address spaces are namely the Multi-bus memory space, the Multi-bus I/O space and the On-board I/O space. The Multi-bus memory and I/O spaces are used to access devices that are attached to the IEEE-P796 Multibus or the VMEbus. Other special On-board devices like the TOD(timeofday) clock can be accessed through the On-board I/O space. By setting a special page type field in a hardware page map entry, the page can be declared to be in one of these address spaces. As a result, the Sun MMU can map any part of a program address space to the Multi-bus memory space, Multi-bus I/O space or the On-board I/O space.

2.2 Effects on System Architecture

2.2.1 Software Structure

Team Shoshin comprises of four layers of software executing under the two different processor states[see Fig 2.2]. The lowest layer is the kernel which executes only

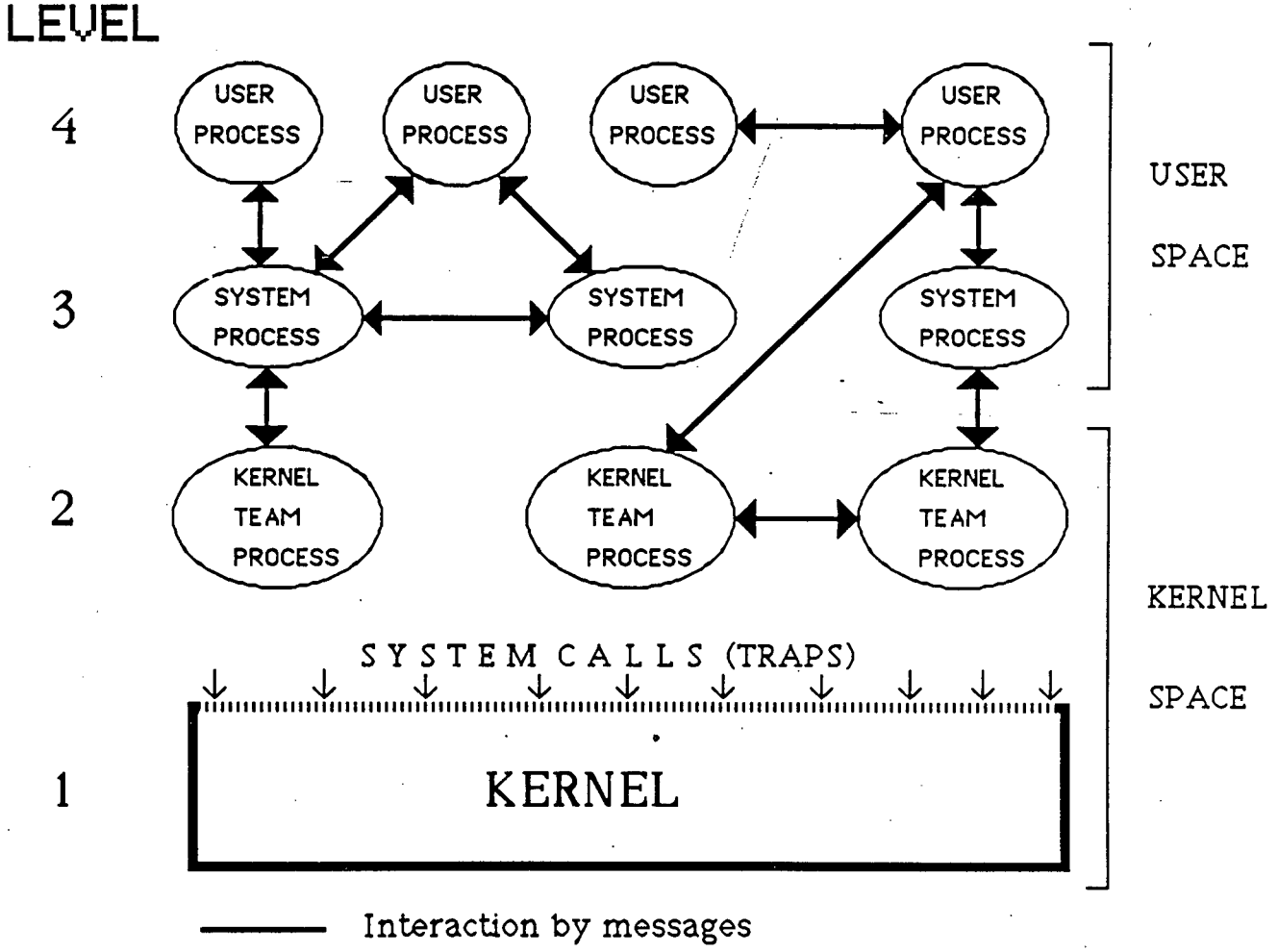

Figure 2.2: Software structure in Team Shoshin

in supervisor mode and provides the bare essential functions to support the higher software layers. It is guaranteed memory resident and resides in the supervisor context

whose segment map is set up to have access to the entire physical memory space. Next in the software hierarchy is the kernel team processes that execute in kernel space(see Chapter 4), followed by a layer of system processes that execute in user space. Both of these software layers execute under user mode and are responsible for implementing all other higher system abstractions. Finally, at the highest layer are the user application programs that run in user mode.

2.2.2 Contexts and Process Management

The concept of a context realized by the SUN's memory management hardware has allowed for rapid process switches, a crucial factor in attaining an efficient multi-task environment. By manipulating the user context register under supervisor mode, we can switch between 8 different segment maps allowing 8 contexts to be mapped concurrently. A Process switch to an existing process residing in a context is very cheap as there is no memory translation state information to be reloaded except for the user context register. This raises the question of how allocation of contexts to processes for multitasking is to be accomplished to fully exploit the SUN's MMU. One simple scheme to facilitate multitasking is to allocate one context per process. This scheme will work well as long as the number of executing processes do not exceed the number of contexts which is eight. For the case where there are more processes than available contexts, some context replacement strategies have to be adopted to displace

an existing process residing in context to make room for other processes. The cost for a context replacement is substantial since the segment map entries have to be reloaded for the new process. This performance penalty associated with a context replacement has prompted us to devise means to reduce the occurrences of context replacements.

2.2.3 Associating Teams with Contexts

One obvious way to reduce context replacements is to have a team of processes sharing a context. The team of processes must all be executing in the same address space requiring only one segment mapping. Consequently, process switches between team members will not require the loading of a new context. This observation has lead us to include Cheriton's team creation into Shoshin so that team of processes sharing the same address space can be created and be associated with a context.

The close association between teams and contexts has provided us with some interesting ideas in configurating system software. One of which is to dedicate some of the available hardware contexts to certain important process teams. The rationale for this is that the operations of certain system process teams have substantial bearing on overall system performance. Due to the design philosophy of Shoshin that advocates the design of a small kernel with higher system functions abstracted by server process, there is a large volume of message exchanges generated by user requests for services. As a consequence, an exceptionally high degree of process switchings are involved due to

the synchronous nature of Team Shoshin IPC primitives. In particular, a server process which provides a highly demanded service will be constantly blocked and unblocked most of the time receiving requests. If this server process team is not guaranteed to be context resident, chances are some process team will be mapped into its context when it blocks, increasing the probability of context replacements. Thus it is reasonable to have certain server process team that offers a highly demanded service to be context resident. The trivial case is the kernel which resides permanently in the supervisor context (context 0). To fully utilize the supervisor context, we also included a process team which comprises of kernel team processes sharing the same address space as the kernel. This multiprocess structuring of the kernel will be discussed in detail in Chapter 4. Next, the server process team called the communication manager is also configured to be guaranteed context resident in context 1. The communication manager is solely responsible for implementing the remote IPC interface and serves as the only link through which remote messages can be exchanged. Clearly, its services are highly demanded as rendered by the distributed nature of Team Shoshin.

2.2.4 Data Transfer between Contexts

Due to the nature of SUN's MMU hardware, access to data in a different context can only be done in supervisor state with respect to the current user context. In other words, data transfer between contexts is only limited between supervisor con-

text and the current user context. This poses a serious limitation in supporting data transmission between processes residing arbitrarily in different contexts. In the case of Shoshin's interprocess communication, this data transfer support is crucial as the message sent by the sender is copied directly into the receiver address space to avoid buffering.

To solve this problem, two special segment blocks each consisting of two consecutive hardware segment entries are reserved in the supervisor context. When data transfer between two different user contexts is required, the segment groups containing the data of one of the target context are mapped into the special segment block entries reserved in the supervisor context. Data transfer is then carried out between the supervisor context and the other target user context through special MC68010 instructions. This scheme will allow us to copy data of size ranging from 32 Kbytes to 64 Kbytes depending on where the target address lies with respect to the segment boundary. This limit can be easily increased by allocating more hardware segment entries to each of the special segment block in the supervisor context. The use of special MC68010 instructions to perform data transmissions between the supervisor context and the user context cost more than normal copy instructions which operate within a context. For efficiency, data transfer between different user contexts of size larger than 512 bytes will have the segment groups of both target addresses mapped into the supervisor context so that data copying can be done within the supervisor context. The 512 bytes optimal limit

is obtained by numerous iterations of trial and error.

2.2.5 Mapping of Peripheral Devices

The "memory-mapped" input/output support for peripheral devices in the SUN workstation has enable us to provide dynamic mapping of certain devices into user address space. Ordinarily, the absence of such mapping facility will require device managers to trap into the kernel space via system calls to gain access to the peripherals. This usually resulted in reduced efficiency due to the overhead in system call associated with each access to the device by the device manager. The cost can be extremely high especially when large amount of data is required to transfer into or out of the device frequently, as in the case of the Ethernet controller. Moreover, access control to the device has to be enforced each time the device is being accessed. In contrast, access control for the device is applied only once when mapping of the device is requested. Once this is done, the server process basically "owns" the device and access control is no longer required.

The dynamic mapping of devices is performed by the primitive **memmap** and currently only mapping of the 3Com Ethernet[3COM] controller is supported. Due to the device-dependent nature of the mapping, the actual routine that does the mapping is obtained by indexing into the *devtab* configuration table in the kernel using the device number argument passed to **memmap**. In this manner, access controls for the mapping

of a device can be specifically tailored for that device and enforced. Peripherals that are do not support dynamic mapping simply have null mapping routines in its entries in the device table.

Chapter 3

Description of Team Shoshin

An overview of Team Shoshin is presented in this chapter. The system has evolved considerably from its predecessor, the original Shoshin, especially in terms of increased system functionalities. As a result, various changes in the original Shoshin design had been made and we will attempt to document such modifications here. This chapter will first provide a synopsis of the history of Shoshin, its past and current state, followed by an overview of Team Shoshin and its system and user interfaces.

3.1 Past and Present State

The original Shoshin operating system was first written by Hide Tokuda and Sanjay Radia at the University of Waterloo in April 1984. It was initially implemented as a distributed software testbed running stand-alone on a collection of LSI 11/23s interconnected by a tailor made high speed bus called the *Schoolbus*.[TOKU83a] Software development was done in two PDP 11/45s running UNIX which also serve as fileserver

and boot server for downloading Shoshin to the LSI 11/23 machines.

The author commenced working on the original Shoshin in July 1984. The work includes porting the Shoshin kernel to the SUN workstations and rewriting and extending the kernel to its present state. Other extensions in the system accomplished by the author include user team creation based on Cheriton's Team concept[CHER79a], dynamic memory allocation using SUN's memory management hardware, facility to map certain selected devices into user memory space, timing services, multiple terminal support and a uniform distributed I/O interface. The remote IPC software written for the original *Schoolbus* [TOKU83a] hardware was ported by Donald Acton to the Ethernet environment. Much of the work on remote IPC facilities which is handled by a server process called the Communication manager was reported in [ACTON85a][ACTON85b].

At present, through a rudimentary UNIX-like command interpreter called the 'micro-shell', Team Shoshin features multi-tasking and transparent (local and remote), device-independent input/output of terminal and disk files. Currently, all disk files are adopted from the UNIX's filesystem and are remotely accessed from machines running 4.2BSD UNIX through guest-level implementation of Team Shoshin's IPC interface under UNIX.

3.2 Process Management

As in the original Shoshin, a process in Team Shoshin is a logical component of a distributed program. Each process has a unique 48 bit network wide process identifier (PID) which is assigned at process creation time. It consists of two parts, a unique 32 bit host identifier (HID) and a 16 bit local identifier (LID) assigned by the kernel to a process upon its creation. The unique HID is the internet host identifier and this allows interprocess communications to be extented to systems running 4.2BSD UNIX. Such static allocation of host identifiers is mainly adopted for the purpose of convenience and there are various problems associated with this fixed scheme. Currently, a new host identification scheme [CHAN86] is being studied and may be adopted in later version of Team Shoshin. A simple local identifier generation scheme described in [CHER79a] is used.

3.2.1 Process Creation

The only major difference in process creation in Team Shoshin as compared to the original Shoshin is the implementation of the team process concept. The following are the primitives for process creation that are available in Team Shoshin.

pid = **create**(pname,f_tree,rel_pri,at)

pid = **teamcreate**(func,rel_pri,ssize,nargs,arg1,arg2,..)

The **create** primitive is identical to the one in the original Shoshin. The process to be created is defined by the name "pname" which specifies the file name of the

load module. The created process can be attached or detached from its creator's family tree by passing ATTACH or DETACH as the parameter in "f_tree". This is for process destruction since only child processes that are attached to the parent will be destroyed when their parent dies. The "at" parameter indicates the user preference for the location of the new process. The priority of the child process is given by the formula, parent's priority + rel_pri, such that the lower the priority value, the higher the actual priority. In Team Shoshin, a process created by the **create** primitive is a team root which is analogous to the team root in [CHER79a]. A team root can be viewed as a *full-fledged* process that owns the process address space shared by its offsprings created in the team. Naturally, the death of a team root will result in the destruction of all its team processes.

The primitive **teamcreate** is identical to Cheriton's notion of creating a process on a team. This type of process creation is fast as it does not involve creating a separate address space, or locating the file containing code and initialized data. The created child process shares with its team root the same address space, code segment, data segment and memory free list. The entry point of the function to be executed as a new process is specified in "func" and the parameter "ssize" indicates the size of the new process stack to be statically allocated from the memory free list. Due to the fixed allocation of the process stack, it is the user responsibility to ensure sufficient stack size is specified in "ssize". In contrast, the process stack of a team root is dynamic

and grows from high memory to low memory. The "nargs" parameter specifies the number of arguments passed to the function and "arg1","arg2".. are the arguments to be passed. A process created by **teamcreate** is automatically attached to its creator and is destroyed when its parent terminates.

The following system primitive is identical to that of UNIX's and is provided to facilitate the passing of command line argument of C [KERN78] in Team Shoshin.

execv(pname,argv)

The text and data segments are found in a file called "pname" and "argv" is an array of character pointers to strings. Upon successful execution of **execv**, the caller process will be transformed into a new process with its core image overlayed by the text and data segments specified in "pname". The argument list consisting of strings specified in "argv" will be made available to the new process. The use of the primitive **execv** is forbidden in team roots since **execv** will result in destroying the original core image that might be shared by other processes. Upon completion of the **execv**, the process will be detached from its parent and becomes a team root.

3.2.2 Process Destruction

The destruction of a process is usually accomplished by either an explicit call to the primitive **exit**, or an implicit one when the process "falls off its code". Process destruction can also be explicitly initiated by other processes using the **kill** primitive. In any case, all resources owned by the terminated process are reclaimed and will be

available for future use. In particular, the process stack allocated for a terminated team process is also recovered and returned to the memory free list shared by surviving team mates. This recovery is crucial as it facilitates dynamic process creation and destruction within a team without exhausting the memory free list all the time. It is interesting to note that such resources if not reclaimed will be ultimately recovered when the team root dies. When a process terminates, a process destruction wave is initiated such that all its attached descendents are automatically killed.

3.2.3 Process Scheduling

A *ready* process is one that can execute when the CPU is available. A process that is *active* or currently having the CPU shall relinquish the processor under the following conditions.

1. process time quantum expires.
2. preempted by a higher priority process that has become *ready*.
3. process blocks for synchronization.
4. process terminates

The *ready* processes are grouped into priority queues such that processes belonging to a particular priority queue have the same priority associated with that queue. The priority queues are then arranged into a *ready* queue in descending order of priority.

When the *active* process relinquishes the CPU, the first process on the *ready* queue will be scheduled to get the CPU.

Due to the introduction of team concept, additional considerations have to be made in process scheduling to ensure team processes execute indivisibly within their team. This assurance that no *active* team process is preempted by its team mates is vital in maintaining data consistency within the team. In order to accomplish this, the following additional restrictions are enforced. First, no team process can have higher priority than its team root. Second, a process can only be preempted by a higher priority process of a different team. Third, upon expiration of the time quantum, the *active* process will only relinquish the CPU if the next process to be scheduled belongs to a different team and have priority equal or higher than itself. Finally, a process that goes into a *ready* state from an *active* state will be entered into its team root priority queue such that no processes of the same team come before it. In the normal case where a process unblocks and becomes *ready*, it will be entered FIFO into the priority queue associated with its priority.

3.3 Interprocess Communication

The interprocess communication facility of Team Shoshin provides a simple, yet extensible set of IPC primitives. As far as the user process is concerned, both local and remote communications share the same IPC interface and is totally transparent. In any

case, the user process simply supplies the PID of the process it wants to communicate with to the desired IPC primitive. Message lengths are arbitrary and the maximum message length is only limited by the maximum buffer size at the receiver site. The Team Shoshin IPC model provides both blocking and non-blocking types of receive primitives but only blocking type send primitives. The flow-controlled send **fsend** [TOKU83b] primitive was not implemented.

3.3.1 IPC Primitives

A process can send a message using the three direct IPC send primitives. They are defined as follows:

nr = **request**(topid,&msg,m,&buf,n,mtag)

ns = **reply**(topid,&msg,m)

ns = **bsend**(topid,&msg,m,mtag)

The "topid" indicates the destination PID, "&msg" and "&buf" indicate addresses of message and buffer areas. The two parameters "m" and "n" indicate the byte sizes of the message and reply message respectively while "mtag" specifies the message tag value to be used in Selective Receive. [TOKU83b] The values "ns" and "nr" represent the number of bytes to be sent and received respectively.

As in the original Shoshin, the **request** and **reply** primitives are used for establishing a "client-server" model between processes. When a process sends a message using the **request** primitive, the requestor will be blocked until the reply message is

returned from the receiver. In the case of the **bsend** primitive, the sender is blocked until the receiver receives the message. There is no need for a reply message from the receiver to unblock the sender.

The receive primitives provide both blocking and non-blocking modes of operation:

nr = **brec**(frompid,&buf,n)

nr = **nrec**(frompid,&buf,n,mtag)

xpid = **brecany**(&buf,n,mtag)

xpid = **nrecany**(&buf,n,mtag)

Where the sender's PID is specified in the parameter "frompid" and the return value "xpid" is a special structure that contains the sender's PID and the number of bytes received. Descriptions of the other parameters are given in earlier definitions.

The blocking **brec** primitive blocks the receiver until the message arrives from the desired sender. The message tag parameter "mtag" is removed in **brec** since in Team Shoshin a sender can send at most one message. This is not true in the case of the original Shoshin where non-blocking flow-controlled send **fsend** is supported. In the case of blocking receive any (**brecany**) the receiver is blocked until a message with the specified tag arrives. All other non-blocking receive primitives are based on querying the message queue of the process and retrieving the specified message if it exists.

3.3.2 Relaying of Remote IPC

The handling of remote IPC is carried out by a special server process called the communication manager, which is solely responsible for all message exchanges between

machines. By examining the location of the destination process, the kernel upon detecting a remote IPC invocation will relay the IPC request to the communication manager. The relaying of remote IPC requests in Team Shoshin differs considerably from its predecessor, the original Shoshin. For remote invocation of IPC primitives, the original Shoshin takes the approach of having the caller build a request packet in kernel space and then sending, via the **request** IPC primitive, the request to the communication manager. When the communication manager receives the request it performs the requested actions and replies the result directly using IPC to the caller process. Advantages of this approach arises from the apparent adoption of a straightforward service access protocol, where user processes simply send explicit requests to the servers relying heavily on the local IPC facilities. However, this advantage might be deceiving as one has to consider the complexity involving kernel stack management to facilitate user processes to build request packets in kernel space and the unwinding of kernel stack upon the unblocking of the caller process. Ironically, this approach appears to be procedure-based, since the kernel in this case will have to run as an extension of the user process and has to be blocked on behalf of the user process. It can also be costly as a result of inefficiency due to the redundancy associated with the nature of the service primitive.

In particular, for remote IPC, the caller who invoked the **request** primitive to a remote process has to build a request packet in kernel space and invoke the **request**

primitive again (recursively) to send the request to the network server, in this case the communication manager server. Clearly there is a great deal of redundancy associated in the above design since the second recursive call to **request** will go through checks and tests which are not necessary. Every remote IPC invocation in the original Shoshin actually involves two calls to the local IPC facilities thus making it rather expensive to use.

To remedy this problem with remote IPCs, we decided to remove the dependency on local IPC routines for the relaying of remote IPC requests to the communications manager. Instead, the relaying of remote IPC requests are carried out directly at the kernel level. When a user process invokes an IPC primitive, the activated routine in the kernel checks the location of the destination process in the specified process identifier. If the destination process is on a remote host, the process descriptor of the invoking process is then queued directly to the message queue of the communication manager server with additional remote IPC related information entered in the descriptor. The communications manager server obtains the remote IPC requests from the kernel through a special kernel primitive called **getcommreq**. Upon completion of a remote IPC, the communications manager returns the IPC status directly to the user through another special kernel primitive, **remipcdone**. With this approach, the user processes are viewed as blocking and resuming execution at the point of entry into the kernel. The support for kernel software to run as an extension of a user process and

to be able to sleep on the behalf of it is no longer required. In effect, such approach enforces a clear boundary between user and kernel execution threads which greatly reduces complexity and enhance understanding in kernel software.

A possible objection to the above solution is the large potential overhead involved in maintaining remote IPC information in the process descriptor. Fortunately we find the fields that were originally designated in the process descriptor for local IPCs are sufficient, and can be easily be extended to cater for remote IPC.

3.4 Input/Output Facilities

Team Shoshin I/O interface is implemented by server processes with client processes communicating their I/O requests via IPC. The I/O interface is distributed by the fact that Team Shoshin supports transparent local/remote IPC. Consequently, as far as the servers are concerned, there is no difference in local and remote requests thus allowing location independent accesses of objects (files,devices). Under this arrangement, the I/O system has provided Team Shoshin a reasonably uniform interface with local and remote peripheral devices by facilitating the indiscriminate use of standard I/O library routines on such devices. The data transfer mechanism for I/O operations is achieved by adopting Cheriton's "connectionless" object-based protocol[CHER81]. Under this protocol, the concept of a file is generalized to that of a view of an object or activity abstracted by a server. To access an object, the object must be first opened

by

objdescrpt_ptr = **OpenObject**(objname,mode)

where "objname" specifies the pathname of the object with access mode indicated by "mode". The primitive **OpenObject** returns a pointer objdescrpt_ptr to an object descriptor which contains information about the opened object. This object descriptor is a non-sharable per-process data structure and the pointer to it serves as an identifier to the accessed object.

Read and write operations can now be performed on the accessed object through the objdescrpt_ptr explicitly or implicitly. By explicitly, we mean that the object descriptor pointer has to be specified as a parameter in the operation. The primitives involved in these explicit read/write operations are

byt_read = **ReadObject**(objdescrpt_ptr,buf,bsiz)

byt_wrote = **WriteObject**(objdescrpt_ptr,buf,bsiz)

ReadObject attempts to read "bsiz" bytes from the object specified by objdescrpt_ptr into the buffer "buf". The number of bytes actually read is returned. In the write operation, **WriteObject** will attempt to write "bsiz" bytes starting from address "buf" into the specified object. Similarly, the number of bytes actually written is also returned.

For implicit read/write operations, the accessed object has to be selected as one of the standard I/O units or streams of the process. In Team Shoshin, each process

is created with three standard I/O units, STDOUT_U, STDIN_U and STDERR_U corresponding to standard input, standard output and stardard error respectively. Each of these standard I/O units is used to index into an object descriptor pointer which is initially inherited from the parent process. The accessed object can be selected to be a standard I/O stream by

oldobjdescrpt_ptr = **resetiounit**(iounit,objdescrpt_ptr)

where "objdescrpt_ptr" is a pointer to the accessed object to be selected as stardard I/O unit "iounit". A pointer to the previously selected object descriptor for "iounit" is returned. Read/write operations on stardard I/O units are performed by the following primitives.

putchar(ch)

_perror(ch)

ch = **getchar**()

The primitives **putchar** and **_perror** will attempt to write the character "ch" into standard output and stardard error respectively. In **getchar**, a character from the standard input is read and returned.

To facilitate efficient input/output, device dependent buffering schemes are implemented with the read/write operations. In order to ensure that data output is written out onto the device, the primitive

flush_buffer(objdescrpt_ptr)

flushes out all buffered data onto the object pointed by "objdescrpt_ptr". Finally, the primitive

CloseObject(objdescrpt_ptr)

flushes out all previously buffered output associated with the object specified by "objdescrpt_ptr" and terminates access to the object.

3.5 Other System Facilities

3.5.1 Dynamic Memory Allocation and Mapping

The following UNIX-like primitives are available for allocating memory from the memory free list which grows from low to high memory.

memory_vec = **malloc**(msize)

memory_vec = **valloc**(msize)

A pointer to a contiguous array of memory of "msize" bytes will be returned to memory_vec. In the case of **valloc**, the block of memory allocated is guaranteed to start on a page boundary. Memory allocated can be returned to the free list by using the primitive

free(memory_vec)

To facilitate the mapping of objects into user address space, the following primitive is provided.

memmap(obj,uaddr,len,offs)

The pages starting at "uaddr" and continuing for "len" bytes are mapped onto the object specified by "obj" at offset "offs". The parameter "uaddr" must be on page boundary and the "len" and "offs" parameters must be in multiples of page size.

3.5.2 Timing facilities

The system clock abstraction of Team Shoshin supports the following primitives.

gettimeofday(date_and_time,tzone)

settimeofday(date_and_time,tzone)

gettime(time)

delay(secs,clks)

The time and date of the day can be obtained by calling the primitive **gettimeofday**. The date/time and timezone values are returned by copying into the structures "date_and_time" and "tzone" respectively. The primitive **gettime** provides a similar function but it attempts to read the timeofday hardware clock thus providing more accuracy. It is used mainly for performance measurements. The date/time and timezone of the system can be set by the primitive **settimeofday**. Finally, the primitive **delay** allows the invoking process to sleep for "secs" seconds and "clks" clicks. A click is equivalent to one clock interrupt and it is machine dependent.

3.5.3 Miscellaneous System Primitives

In addition to the basic IPC, process creation and I/O services supported by Team Shoshin, the following system services are also provided to look after various aspects

of Shoshin programming environment.

pid = **self**()

pid = **parent**()

pid = **whois**(logname)

pexist(lid)

The primitive **self** returns the process id of the caller process. A process can find out its parent's process id by invoking the **parent** primitive. The primitive **whois** allows user processes to find out process ids of *well-known* server processes using their logical names. Lastly, the primitive **pexist** checks the existence of a process specified by its local process id "lid". A non-zero value is returned if the process in question exists, otherwise a zero value will be returned.

Chapter 4

Multiprocess Structuring of the kernel

This chapter describes the design and implementation of the kernel multiprocess structure. The main objective here is to explore the idea of extending the concept of a process to the kernel level by experimenting with kernel team processes that execute within kernel address space.

4.1 The Rationale

One of our foremost concerns in the implementation of Team Shoshin has been efficiency. Our approach in this aspect is not to change the original goals of Shoshin as a flexible and modular distributed system, but to find ways to accomplish these goals in a more efficient manner.

One common approach is to put more functionalities into the kernel for increased efficiency. In fact, we are observing a growing trend in current system architectures

where an increasing number of system functions ordinarily abstracted by server processes are now being implemented directly into the kernel. Unfortunately, such migrations of system functionalities into the kernel are often accomplished in an adhoc manner which inevitably increase the complexity and size of the kernel. In addition, such approach reduces the use of server processes in carrying out system functions, thus seriously curtailing system flexibility in terms of ability to change its functionality and configuration by adding or deleting server processes. What one would like to have is some systematic means which allows one to implement system services in the kernel with relative ease. Such scheme should also be flexible enough so that it permits easy deletion of an existing service if one chooses to have that service implemented outside the kernel. In this way, the kernel is allowed to have the facility of adding functionality without increasing complexity.

Having all these considerations in mind, we decided to investigate the use of multiprocess structuring techniques at the kernel level as a basis for such a scheme. The proposed multiprocess structure is the creation of kernel team processes that execute in kernel space. With this kernel team concept, we now have the means of migrating selected server processes into the kernel thus preserving their process abstraction without having the need to implement their functions directly in the kernel. We view this extension of the concept of a process into the kernel to be very useful since it has the potential of providing the ease in adding/deleting functionalities into/from the kernel

by simply adding/deleting kernel team processes systematically. In terms of efficiency, the use of kernel team processes is by no means better than the direct implementation approach (that is implementing services directly into the kernel). But it clearly has the advantage of simplicity and flexibility in ease of reconfiguration while improving performance by cutting the overhead in process switching and scheduling since kernel team processes are guaranteed to be memory resident. Another advantage arises from the sharing of the kernel data segment by all kernel team processes. The ease of access to certain kernel data structures is sometimes crucial in the performance of certain processes. In addition, devices especially memory-mapped ones are usually mapped into kernel space and kernel team processes that handle such devices can access them with ease.

The extension of multiprocess structuring concept to the kernel also provides a means of factoring traditional kernel functions into kernel team processes. This allows us to explore the idea of an "active" kernel as opposed to the convectional kernel abstraction that passively implements its traditional set of so called *kernel* functions. To illustrate this point, consider process creation which is traditionally implemented as a kernel function. Accessing it will require the user process to perform a trap to the kernel via a system call. Due to the localized nature of system calls, a separate mechanism will be needed to handle remote requests for process creation. One technique commonly used here is the Remote Procedure Call(RPC)[BIRR84] mechanism.

In contrast, if process creation is abstracted as a kernel process where accesses for services are being realized by IPC, then there will be no distinction in the access protocol for local and remote invocation with transparent local and remote IPC support. It is apparent from this illustration that conventional kernel abstraction by its inherently passive nature somewhat enforces localised control as oppose to distributed control. In many aspects, the introduction of the kernel multiprocess structure actually allows the kernel size to be logically reduced by factoring out kernel functions into processes. In effect, we have a logically smaller kernel.

4.2 The Design

From the discussion in the previous section, it is clear that for the kernel multiprocess structure to achieve its goals, certain design criteria must be observed.

1. The kernel multiprocess structure should be simple.

2. The kernel multiprocess structure must allow easy addition or deletion of kernel team processes.

3. The functionalities abstracted by kernel team processes can be interchangeably implemented by ordinary server processes with relative ease.

The application of these criteria to the design of the kernel multiprocess structure has resulted in the static creation of kernel team processes during system initialization

time. This fixed creation avoids the problems associated with a dynamic scheme which inevitably requires some form of resource allocation/reclaimation strategy within the kernel. Such fixed scheme allows for the preallocation of all resources required by the kernel team processes during the crucial system startup phase. Specifically, the preallocation of resources occurs before system resources are being sized up and accounted so that there will be no interference involving operations carried out by existing kernel resource management routines. To completely avoid the problems of resource reclaimation upon possible death of a kernel team process, the attribute of immortality is imposed upon the created kernel team processes. The introduction of this immortality attribute is consistent with the whole rationale behind the kernel multiprocess structure which is to be viewed as a tool for structuring operating systems, in this case, the kernel. This kernel multiprocess structure can be regarded as an integral component of the kernel and thus any failure in a kernel team process is viewed as a failure in the kernel. Of course, one can adopt various scheme in handling failures in the kernel team processes to achieve some level of fault tolerance. The discussion of fault tolerance in this aspect is beyond the scope of this thesis.

The design criterion to permit easy interchangeable use of kernel team processes and ordinary server processes for realizing system functions has resulted in making the kernel team process abstraction very similar to that of an ordinary process. A kernel team process like an ordinary process is an independent entity which is uniquely identi-

fied by its process id assigned at creation time. As far as other processes are concerned, it is just an ordinary process that can be communicated to via IPC. Access to system services by kernel teams is performed in the similar manner as ordinary processes. All stardard system library routines except for memory allocation (see below) in Team Shoshin are shared and used by both kernel teams and ordinary processes. With this uniform treatment in terms of access to system services for kernel team processes, interchangeable use of kernel teams and ordinary server processes can be facilitated. For example if one decided to reconfigure the system by migrating an existing system function realized by a kernel team process to a server abstraction outside the kernel, little changes will be required since the system interfaces used in the kernel team implementation are basically similar to that of an ordinary process. The same applies in the reverse situation. With this scheme, an operating system designer can restructure the system by bringing it down and reconfigure it with ease through adding/deleting kernel team and ordinary server processes. Note that in the case of memory allocation, a separate set of primitives are provided for the kernel team processes. This is necessary since standard memory allocation primitives are only applicable to user space. To provide such capability, memory buffers are preallocated at system startup time and accesses to these memory buffers are done through calls to the special memory allocation routines provided.

One possible objection to this uniform treatment is that the advantage of accessing kernel primitives directly by the kernel-resident processes is not tapped. Instead, the kernel primitives are accessed in the same manner as an ordinary process via a kernel trap with its associated overhead using standard system library routines. Apart from the sake of achieving uniformity as described above, the decision to abandon the direct access of kernel primitives was also based on observations that led to the conclusion that the efficiency advantage might be deceiving. First, almost all of the kernel primitives are highly important functions that require access to privilege machine instructions. Of the most common usage of these instructions is the raising and lowering of interrupt levels to manage critical sections which occur in almost everywhere in kernel primitives. Since these machine instructions are privileged, the execution of them will require the kernel-resident process to be run under supervisor mode. The problem now is that under supervisor mode the kernel stack frame will have to be used, which consequently require the adoption of some kernel stack management scheme for kernel team processes. To make matter worse, the execution of some kernel primitives may result in the invoking process being blocked requiring the kernel to sleep on the behalf of the process. Such additional complexities associated with facilitating direct access of kernel primitives could be too costly to fully reap the benefeits of reduced overhead through direct access. Moreover, there will be little distinction in the kernel and kernel team processes execution threads which basically defeat the whole purpose

of introducing the concept of a process in the kernel.

Kernel team processes are created according to two configuration tables namely, the *kteamconftab* and the *devtab*. During system startup, the entries in these two configuration tables are interrogated and for each non-null entry, a kernel team process might be created according to the specifications in that entry. The *kteamconftab* configuration table contains specifications of all machine independent kernel team process that must be created. Each entry in *kteamconftab* [see Fig 4.1] comprises of three fields,

```
/* Structure declaration for kteamconftab configuration table */
typedef struct {
    short kteampri;      /* process software priority */
    int (*kteamproc)(); /* entry point for kernel team process */
    int logname;         /* logical name */
} KTEAMCONF;
```

Figure 4.1: *kteamconftab* Configuration Table

kteampri, *kteamproc* and *logname*. The entry point of the kernel team process is specified in *kteamproc* and it is created with software priority *kteampri*. The field *logname* is an integer-valued logical service name used to index into the kernel name cache. Upon creation, the pid of the kernel team process will be entered into the kernel name cache via its *logname* so that registration for the offering of services is immediate.

To cater for creation of machine-dependent kernel team processes, the *devtab* configuration table is used. The *devtab* contains specifications of kernel team processes whose existence are used solely for the handling of peripheral devices. These kernel team processes fall basically into the category of device workers that operate directly on devices. For each peripheral device handled by the system, there is an entry in the *devtab*. Each entry [see Fig 4.2] consists of numerous fields, of particular interest

```
/* Structure declaration for the devtab configuration table */
typedef struct {
    PD *dev_hdrpd;      /* pointer to kernel team PD */
    short hdr_prio;     /* hardware interrupt level of device */
    int (*dev_hdr)();   /* entry point for kernel team process */
    int dvcsr_addr;     /* start address of device */
    int (*deviint)();   /* routine for handling input interrupt */
    int (*devoint)();   /* routine for handling output interrupt */
    int (*dev_probe)(); /* routine for probing the device */
    int (*dev_mmap)(); /* routine to perform memmap for device */
    int (*dev_init)();  /* routine to initialize device */
    int dev_unit;       /* device unit number */
    int logname;        /* logical name */
} DEVSW;
```

Figure 4.2: *devtab* Configuration Table

here are the fields *dev_probe*, *dev_hdrpd*, *hdr_prio*, *dev_hdr* and *logname*. During system startup, each entry of the *devtab* is examined and the device specified in the entry is

probed for existence using the device probe function in *dev_probe*. If the device exists, a kernel team process will then be created with entry point specified in *dev_hdr*. With this arrangement, we can ensure that these kernel-resident device workers are only created for peripherals devices that exist. Note that in the case where the *dev_hdr* has a null entry, no process will be created for that device. Once created, the kernel team process will be assigned a software priority that reflects the device hardware interrupt priority level specified in *hdr_prio*. This assigned software priority will be one of the 7 highest software priorities corresponding to the Motorola's 7 interrupt levels. The created process is always executed under that interrupt level to prevent itself from being interrupted by the device that it manages.

4.3 The Implementation

Currently, the *kteamconftab* is configured to create two kernel team processes, the *ttyserver* and the *nameserv* [See Fig 4.1]. The *ttyserver* is responsible for the handling of all terminals attached to the Zilog[1] serial communication ports provided by the Sun Workstation. For each serial device, the *ttyserver* manages two globally declared circular buffers for receiving and transmitting characters. Every time the serial device interrupts to accept a character for transmission, the interrupt handling routine in the kernel simply removes a character from the appropriate transmit circular buffer and

[1]Zilog is a trademark of Zilog Inc.

writes it to the device. Similarly in the case of receiving a character from the serial device, the interrupt handling routine concerned simply reads the character from the device and enters it into the receive buffer. Clearly with this scheme, actual character I/O on the serial devices can be made almost completely asynchronous as a result of close co-operation between the device interrupt handling routines and its device manager, the *ttyserver*. Such close co-operations were fostered through data sharing made possible by implementing the *ttyserver* as a kernel team process. Next, the kernel team process *nameserv* is implemented as a local name server. The *nameserv* process is responsible for managing the kernel name cache data structure used in name/address translations. Requests from user processes are accepted for translation of service names into corresponding server's pid. Registration of server processes into the name cache are also accepted and the name cache is then updated.

The *devtab* configuration table presently creates two kernel team processes as device workers [See Fig 4.2], the *zsdev_hdr* and the *ecdev_hdr*. The sole function of these device worker processes is to serve as a synchronization medium between its device managers and their corresponding devices. For example, the *zsdev_hdr* process spends all its time waiting for an interrupt from the Zilog serial controller and once unblocked, it will signal to the *ttyserver* using the **bsend** IPC primitive and proceed to wait for the next interrupt. In this way, the *ttyserver* never waits on the Zilog controller and all outstanding I/O requests are queued to be act upon later when the *ttyserver* is

notified by *zsdev_hdr*. The unblocking of the *zsdev_hdr* process is carried out directly by the Zilog interrupt handling routines and this only occurs when the transmit buffer is empty and there is an outstanding write request pending or a character just arrived with a outstanding read request pending. From our experience, such conditions exist rather infrequently and in most cases the *ttyserver* spends a large portion of its time accepting user requests and filling/depleting circular buffers leaving the bulk of the actual I/O operations to be carried out asynchronously by the interrupt handling routines. In the case of the *ecdev_hdr*, it serves as a device worker for the 3Com Ethernet controller which is managed by the communication manager process. The 3Com controller generates four different interrupts each requiring extensive handling. All these interrupts are generated through one interrupt line and are distinguishable only by reading the interrupt bits from the 3Com control status register. To cater for such elaborate interrupt services, the *ecdev_hdr* is dispatched whenever the Ethernet controller interrupts. It then proceeds to disable the controller and send the interrupt bit sequences obtained from the 3Com control status register to the communication manager which does the actual servicing. When servicing is completed, the communication manager will issue a reply to the *ecdev_hdr* process which will then enable the Ethernet controller according to the reply message and resume waiting for interrupts.

4.4 Retrospect

The static creation of kernel team processes from the *kteamconftab* and *devtab* configuration tables has been successful in achieving easy addition/deletion of kernel team processes by manipulating the entries in the configuration tables. With this scheme, a limited capability is provided to change kernel functionalities without seriously curtailing system flexibility and reconfigurability.

The catering of machine-dependent creation of kernel team processes through the use of *devtab* had increased the applicability of the kernel multiprocess structure for handling device-dependent tasks. Indeed, experiences gained from the implementation of device workers and managers as kernel team processes had demonstrated remarkable close co-operation with the device interrupt handling facilities in the kernel. One direct advantage in fostering such close co-operations is that it facilitate the use of only one device worker to handle synchronization of many similar devices with its device manager. This was illustrated in the implementation of the *zsdev_hdr* process which serves as an synchronization worker process for all the serial ports handled by the *ttyserver*. In contrast, the *ttyserver* in the original Shoshin uses two worker processes to synchronize input/output on a serial port.

The implementation of the *nameserv* process to manage the kernel name cache suggests a possibility of allowing network-wide sharing of kernel data and state infor-

mation. As demonstrated in the use of *nameserv*, the kernel name cache which is a kernel data structure can be accessed locally and remotely by user processes via IPC.

Finally, the use of the kernel multiprocess structure is not without problems. One main problem is that there is a strong tendency for the indiscriminate access of kernel primitives and data structures when writing kernel team processes. The unleashing of the power to create processes in kernel space that permit sharing of kernel data structures must be controlled in order to achieve our design criterion of simplicity. Some form of software methodology must be adopted when designing kernel team processes to achieve a simple, yet powerful kernel multiprocess structure. This software engineering aspect of writing kernel team processes is beyond the scope of this thesis.

Chapter 5

Concluding Remarks

5.1 Summary

In this thesis a description of the implementation of Team Shoshin operating system is presented. The emphases here are the development effort to port Shoshin onto the Sun Workstation and the subsequent investigation into multiprocess structuring of the kernel. Various aspects of the Sun Workstation hardware architecture are presented and its impacts on Team Shoshin software development are discussed. To underscore how much Team Shoshin has evolved from its predecessor, the original Shoshin, a detail description of Team Shoshin is provided in Chapter 3. In this chapter, the system and user interfaces of Team Shoshin are described with particular emphasis on documenting the changes made on the original Shoshin with respect to its design and implementation. In Chapter 4, the investigation into multiprocess structuring of the kernel is presented. The rationale for multiprocess structuring of the kernel is first discussed. This served as a basis for formulating the set of criteria which are

used in designing the proposed kernel multiprocess structure. Various implementations involving the use of the proposed kernel multiprocess structure are described. The applicability of the proposed kernel multiprocess structure and its impact on operating design are then discussed drawing from the experiences gained through its actual use.

One major conclusion drawn from this thesis is that the extension of the process concept to the kernel is worthwhile and deserves further investigation. We do not contend that our proposed kernel multiprocess structure is "the solution" or the only approach to use. What is important here is that the work done in the kernel multiprocess structure has provided us a great deal of insights in its applicability and in its potential as a tool for structuring operating systems. Based on our experience in the implementation of the kernel multiprocess structure, the proper usage of the kernel multiprocess structuring concept is not trivial. It requires further studies and understanding and may even need the development of new techniques in software engineering to fully exploit it.

Finally, the development of Team Shoshin has resulted in a great deal of software restructuring of the kernel. The kernel text size has grown to 43 kbytes compared to 24 kbytes in the original Shoshin, most of which are due to the introduction of the kernel multiprocess structure. Currently, local IPC in Team Shoshin is typically 4.5 to 5.5 times faster than the original Shoshin. Apart from the fact that Team Shoshin has better underlying hardware support, a substantial part of this improvement in IPC

performance is an consequence of kernel restructuring. We estimated that message exchange time using the original Shoshin kernel software structure would be about double the time with kernel restructuring reported in [ACTON85a].

5.2 Future work

Many of the possible topics for future work that stemmed from this thesis presentation is centered on furthering ideas developed on multiprocess structuring of the kernel. In fact, many of these ideas are still not fully explored or even tested. So far we have not investigated the use of the kernel multiprocess structure to factor out traditional kernel functions into processes. A prime candidate for this investigation is to implement local process creation as a kernel team process. Due to the transparent handling of local and remote IPC in Team Shoshin, the impact of this could be substantial since requests for process creation can be initiated locally or remotely. Consequently with this scheme, a separate facility for remote process creation may not be required.

Another possibility is to look into the use of the kernel team processes to facilitate the network-wide sharing of kernel state information. By using kernel-resident processes that "own" such state information, local and remote processes can query or access them with ease via IPC. Alternatively, changes in kernel state information can be also used to affect the behaviour of some kernel team processes so as to reflect the

effects of such changes. For example, by implementing a death notifier as a kernel-resident process that constantly monitors the process table, network-wide notification of a process death can be facilitated by dispatching the death notifier whenever the process table changes.

The development of Team Shoshin on the Sun Workstation has been an on-going project and there are still a great deal to be done to make it a *full-fledge* operating system. We will attempt to mention a few of them here. Currently, all Shoshin programs have to be downloaded with the operating system before they can be executed. It would be nice to have some form of dynamic loading facility that reads in load modules on demand from the disk. One possible way to achieve that is to read load modules from disks in existing Unix systems using our ability to communicate to Unix processes. In fact, a Unix fileserver that accept requests from Shoshin processes is already implemented.

Finally, a virtual terminal management interface that allows a user to view and control the activities of its local and remote processes on the Sun Workstation bit-mapped display will definitely be welcome. Such interface will greatly enhance distributed computing and control at the user level.

Bibliography

[ACTON85a] D. Acton, *Remote Interprocess Communication and its Performance in Team Shoshin*, Master Thesis, Technical Report 85-16, University of British Columbia, November 1985.

[ACTON85b] D. Acton, H. Wang, S. Vuong, *Experience in Interprocess Communication in the Distributed Operating System Team Shoshin*, to appear in Proceedings of IEEE's International Conference on Communications'86, Toronto, June 22-25, 1986.

[BIRR84] A.D. Birrell, B.J. Nelson, "Implementing remote procedure calls", ACM Transaction on Computing Systems, Vol 2, Number 1, February 1984.

[CHAN86] *Host identification in reliable distributed kernels*, S.T. Chanson, K. Ravindran, Technical Report 86-4, University of British Columbia, 1986

[CHER79a] D.R. Cheriton, *Multi-process Structuring and the Thoth Operating System*, Ph.D Thesis, University of Waterloo, 1979.

[CHER79b] D.R. Cheriton, *Interactive Verex*, Technical Report 79-1, University of British Columbia, September 1979 revised January 1983.

[CHER79c] D.R. Cheriton, *Zed Programming Manual*, Technical Report 79-2, University of British Columbia, September 1979 revised January 1983.

[CHER81] D.R. Cheriton, *Distributed I O using an Object-based Protocol*, Technical Report 81-1, University of British Columbia, January 1981.

[CHER83] D.R. Cheriton, T.P. Mann, "The Distributed V Kernel and its Performance for Diskless Workstation", *Proceedings of the Ninth Symposium on Operating System Principles* pp. 128-140 ACM, October 1983.

[ELOV74] H.S. Elovitz, C.L. Heitmeyer, "What is a Computer Network?" IEEE 1974 NTC Record, pages NTC 74-1007 to 74-1014

[KERN78] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1978.

[LAMP81] Edited by B.W. Lampson, M. Paul, H.J. Siegert, *Distributed Systems Architecture and Implementation An Advanced Course*, Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1981.

[LANTZ80] Keith A. Lantz, *Uniform Interfaces for Distributed Systems* PH.D. Thesis, University of Rochester, 1980.

[METC76] R. Metcalfe, D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM* vol.19 #7 pp. 395-404, July 1976.

[MOTO84] Motorola, *M68000 16 32-bit Microprocessor Programmer s Reference Manual* fourth edition, Englewood Cliffs, New Jersey, Prentice-Hall, 1984.

[RAVI85a] K. Ravindran, S. Chanson, *On process aliases in distributed kernel design*, Proceedings of the 6th International Conference on Distributed Computing Systems, Boston, Mass, May 1986.

[RAVI85b] K. Ravindran, S. Chanson, *State inconsistency issues in local area network-based distributed kernels*, Proceedings of the 5th Symposium in Reliability of Distributed Systems and Databases, Los Angeles, January 1986.

[SUN82] *Programmer s Reference Manual for the Sun Workstation* Version 1.0, SUN Microsystem Inc., October 1982.

[SUN83] *Sun 68000 Board*, Revision B, Sun Microsystem Inc., February 1983.

[SUN84] *System Internals Manual for the Sun Workstation*, Revision C, Release 1.1, Sun Microsystem Inc., January 1984.

[TOKU83a] Hideyuki Tokuda, Sanjay Radia, Eric Manning, "Shoshin OS: a Message-based Operating System for a Distributed Software Testbed", *Proceedings of the Sixteenth Annual Conference on System Sciences 1983* pp. 329-338, 1983.

[TOKU83b] Hideyuki Tokuda, Eric Manning, "An Interprocess Communications Model for a Distributed Software Testbed", *Proceedings of SIGCOMM '83 Symposium on Communications Architectures and Protocols* pp. 205-212, March 1983.

[TOKU84] Hideyuki Tokuda, *Shoshin A Distributed Software Testbed* Ph.D. Thesis, University of Waterloo, 1984.

[3COM] *Model 3C400 MULTIBUS Ethernet ME Controller Reference Manual* May 1982.

Appendix A

Kernel Data Structures

This appendix provides descriptions of various important data structures used in Team Shoshin kernel. These descriptions are all in the form of structure declarations using the C [KERN78] programming language.

```
/* Basic type declarations */
typedef unsigned long ulong;
typedef unsigned short ushort;

/* Descriptor type - message descriptor or process descriptor */
typedef enum { IS_PD, IS_MD } PD_OR_MD;

/* Process status */
typedef enum {
    WAITING,
    READY,
    DEAD,
    DELAYING
} PSTATUS;

/* Structure declaration for process id */
typedef struct {
    ulong hid;  /* host id */
    ushort lid; /* local id */
} PID;
```

```
/* Message wait condition */
typedef struct {
    PID    w_pid;       /* pid waited upon */
    short  w_tag;       /* message tag */
    STYPE  w_stype;     /* the ipc type waited upon */
    ulong  w_base;      /* message base address */
    ulong  w_length;    /* message length */
} WCOND;

/* Structure declaration for the Process table entry */
typedef struct PROCTAB {
    char p_name[MAXSPATH]; /* pathname of module */
    ushort p_mode;         /* mode */
    ushort p_uid;          /* user id */
    ushort p_gid;          /* group id */
    int p_tsize;           /* text size in bytes */
    int p_dsize;           /* data size in bytes */
    int p_bssize;          /* bss size in bytes */
    int p_tseg;            /* start of text seg */
    int p_dseg;            /* start of data seg */
    int p_entry;           /* entry point */
    short p_refcnt;        /* reference count */
} PROCTAB;

/* Structure declaration for a message descriptor - for remote IPC    */
/* Note that message descriptors (MD) and process descriptors (PD) */
/* share the 1st few entries allowing PDs to be queued as MDs.       */
typedef struct MSGDSC {
    struct MSGDSC *qnext; /* pointer to next descriptor */
    PD_OR_MD pd_or_md;    /* is it PD or MD */
    STYPE m_sendtype;     /* message sending type */
    ulong m_length;       /* message length */
    short m_tag;          /* message tag */
    ulong m_base;         /* message base address */
    PID p_pid;            /* sender's pid */
} MSGDSC;
```

```
/* Structure declaration for a primary process descriptor */
typedef struct {
    struct PD *qnext;              /* pointer to next PD */
     PD_OR_MD pd_or_md;            /* is it PD or MD */

      /* the following is used when PD is used as a message descriptor*/
      STYPE m_sendtype;            /* message send type */
      ulong m_length;              /* message length */
      short m_tag;                 /* message tag */
       ulong m_base;               /* message base address */

    PID p_pid;                     /* own pid */
    ushort p_sr;                   /* status register */
    ulong p_pc;                    /* program counter */
    ulong p_regs[MAXREG];          /* general registers */
    ushort p_context;              /* process context */
    char p_status;                 /* process status enum PSTATUS */
    char p_pri;                    /* process priority */
    PID p_ppid;                    /* parent's pid */
    struct PD *p_teampd;           /* team pd */
    PID p_child[MAXCHILD];         /* Children pids */
    char p_ctlflag;                /* process control flags */
    PROCTAB *p_proc;               /* pointer to the process table */
    ushort p_segmap[MAXSREG];      /* segment map */
    ulong p_dbase;                 /* base address of data segment */
    ulong p_dsize;                 /* size of data and bss seg */
    ulong p_bssize;                /* size of bss (nearest page) */
    ulong p_lastbssptr;            /* end of used bss area */
    ulong p_stbase;                /* base address of stack seg */
    ulong p_stsize;                /* size of stack seg */
    ulong p_spdpge;                /* page entry of spdbase */
    MDSC *p_msgq;                  /* message queue */
    WCOND p_wcond;                 /* waiting condition */
    ushort p_cpu;                  /* cpu time */
    ulong p_tolcpu;                /* total cpu time used */
} PD;
```

```
/* Structure declaration for kteamconftab configuration table */
typedef struct {
    short kteampri;      /* process software priority */
    int (*kteamproc)(); /* entry point for kernel team process */
    int logname;         /* logical name */
} KTEAMCONF;

/* Structure declaration for the devtab configuration table */
typedef struct {
    PD *dev_hdrpd;       /* pointer to kernel team PD */
    short hdr_prio;      /* hardware interrupt level of device */
    int (*dev_hdr)();    /* entry point for kernel team process */
    int dvcsr_addr;      /* start address of device */
    int (*deviint)();    /* routine for handling input interrupt */
    int (*devoint)();    /* routine for handling output interrupt */
    int (*dev_probe)(); /* routine for probing the device */
    int (*dev_mmap)(); /* routine to perform memmap for device */
    int (*dev_init)();   /* routine to initialize device */
    int dev_unit;        /* device unit number */
    int logname;         /* logical name */
} DEVSW;
```