A Multiple Processor Implementation of the TRIX Operating System

by

David Goddeau

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science

at the

Massachusetts Institute of Technology

September 1983

Signature of Author _____

Department of Electrical Engineering and Computer Science
August 29, 1983

Certified by _____

Thesis Supervisor

Accepted by _____

Chairman, Departmental Committee on Graduate Students

A Multiple Processor Implementation of the TRIX Operating System

by

David Goddeau

Submitted to the Department of Electrical Engineering and Computer Science

on August 29, 1983 in partial fulfillment of the requirements for

the degree of Master of Science

Abstract

This thesis describes a version of the TRIX operating system modified to run on multiple processors. The system is designed for an environment in which several processors share physical memory and devices over a common bus. It is organized to treat all processors symmetrically to provide maximum throughput. The issues involved in running a multiprocessor TRIX are discussed and details of the implementation are described.

The performance of the system is measured and analyzed. A model is given to predict the maximum throughput of a multiprocessor TRIX from system parameters.

Name and Title of Thesis Supervisor:

Stephen A. Ward,
Associate Professor of Electrical Engineering and Computer Science

Key Words and Phrases:

multiprocessor operating systems

ACKNOWLEDGMENTS

Many thanks to Steve Ward, Chris Terman, Jon Sieber, all of the members of RTS for their advice, encouragement, and formatting expertise.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research (Contract Nos. N00014-75-C-0661 and N00014-83-K-0125).

TABLE OF CONTENTS

1. Introduction

This thesis describes a version of the TRIX operating system modified to run on multiple processors. The system is designed for an environment in which several processors share physical memory and devices over a common bus. It is organized to treat all processors symmetrically to provide maximum throughput. The issues involved in running a multiprocessor TRIX are discussed and details of the implementation are described.

1.1. Motivation

With the increasing use of microprocessors as computing engines, the processor represents a smaller fraction of the total cost of a computer system. This is particularly the case in smaller machines and personal computers. The items which make up most of the cost of a small system are main memory (in increasingly large amounts), display and communications hardware, secondary storage, and packaging. This latter includes the power supply, the backplane hardware, and the box containing the system. Given the low cost of processing modules relative to that of system overhead, a single system supporting multiple processors should be a more cost effective source of computing power than a number of single processor units. The challenge is to create a system which can take advantage of the additional processing power a second or third processor provides.

It is not immediately apparent that adding another processor to a system will provide an increase in performance. This will depend on both the hardware used to interconnect the processors and the way in which the software utilizes the additional processor. The general problem of how to focus multiple processors on a task and manage their interaction efficiently is very difficult and as yet unsolved.

There are, however, some specific situations in which it is fairly clear how multiple processors can be used to improve system performance. One is the execution of sequential processes running in a multiprocessing environment. In general, processes specify a separate thread of activity unconstrained by any other process. This implies that all such processes could be run concurrently. In a time-sharing environment the concurrency is supplied by a single processor, but if the processes are run simultaneously on separate processors (or time sliced onto several processors if there are more tasks that processors), there should be an increase in throughput.

1.2. Goals

The system described in this thesis is an attempt to apply multiple processors to the parallelism implicit in multiprocessing. It is an adaptation of the TRIX operating system capable of running on multiple processors. It has been implemented on a personal computer which uses MC68000 microprocessors as the computing element. The processors operate over a common backplane and share a single physical address space.

The main goal of this project was to create a working system that uses several processors effectively. To achieve the greatest efficiency, the processors must be as independent as possible. Thus it is important to avoid both specialization of processor function and "artificial serialization" of processor activity. A secondary goal of the project was to examine the various factors which affect the performance of the system and its effectiveness in a personal computer environment.

TRIX is an operating system designed to efficiently support concurrency and communication between processes. Its semantics lead conveniently to an extension which utilizes multiple processors. Furthermore, the structure of the TRIX kernel is such that an adaptation of this sort is possible without disrupting the underlying framework of the system.

1.3. Overview

Since this thesis describes an adaptation of the TRIX system, most of the issues dealt with are specificly concerned with TRIX. It does not claim to provide solutions to the general problems of multiprocessor organization; rather, it presents the application of some of these principles to in a particular system.

The next three sections of this thesis provide some background information. Section 2 describes other multiprocessor operating systems with goals similar to those of TRIX. Section 3 describes the computer hardware on which the system was implemented. Section 4 briefly discusses the semantics of the TRIX operating system. It also describes in some detail the structure of the TRIX kernel as this is relevant to its adaptation to multiple processors. Section 5 discusses the major issues that arise in this adaptation. It examines possible alternatives and describes some of the important details of the implementation. Some additional semantic issues and the consequences of certain system extensions are also considered. Section 6 is concerned with the performance of the system, in both its current and future incarnations. The

results of experiments with the current implementation are given and various factors affecting its performance are discussed. A model is presented to aid in predicting the throughput of the system on different hardware.

2. Previous Work

This section describes two systems which are very similar to TRIX in their goals. These are the HYDRA system, written at CMU, and a multiple processor implementation of UNIX[1] produced at Purdue University. The focus of the discussion is on the relative similarities and differences between these systems and multiprocessor TRIX.

2.1. HYDRA

The HYDRA system[Wulf73, Wulf81], developed at CMU to run on the C.mmp multiprocessor, has many goals in common with TRIX. Among these is the desire to provide a minimal kernel which supports the fundamental mechanisms of HYDRA. These primitives permit the majority of "system" facilities to be implemented as user level programs. Also, the HYDRA system is designed to run on a multiprocessor, in particular C.mmp. Both TRIX and HYDRA are built on a computational model of concurrent, communicating processes. Both provide facilities for "message passing" and process scheduling and synchronization. However, HYDRA is deeply concerned with the issues of object-oriented programming languages and with protection and sharing at the object level. It uses capabilities extensively to implement object protection. TRIX does not deal with protection at the language object level and does not use capabilities.

The communication mechanisms used by HYDRA and TRIX differ considerably. HYDRA implements a message passing system with implicit synchronization. The communication mechanism is TRIX is an inter-domain procedure call (a procedure call between protected address spaces). The thread of execution continues in the new domain, possibly concurrently with other threads in that domain. This allows asynchrony between the various "messages" and the corresponding "replies", avoiding artificial serialization of message processing.

Multiprocessor TRIX encounters some of the same issues as HYDRA in implementing processor synchronization. The problem is somewhat more extreme in HYDRA where the decision was made to use a very "fine-grained" synchronization structure, in which there are many independently locked data objects. Since TRIX does not deal what objects at that level, the locking can be much coarser.

---

[1] UNIX is a trademark of Bell Laboratories.

2.2. Purdue UNIX

Purdue UNIX[Goble81] is a dual processor adaptation of UNIX implemented at Purdue University. It runs on a DEC VAX/780 modified to run two processors and is intended to provide better throughput to users in a multi-user timesharing environment. The details of the hardware modification are not important here except for the fact that both processors operate on the same system bus and therefore share the same physical memory. The project was undertaken because one dual processor VAX (using the Purdue modifications) is considerably less expensive than two single processor VAX's. It is therefore quite cost effective if the same computing power can be obtained in either case. Another reason given is that the system maintainence and accounting is easier with one system than with two, for example only one set of passwords and login ID's are needed.

The two processors in Purdue UNIX operate in a master-slave relationship. The master processor is by definition the one that was booted first and initialized the system. The slave is subsequently booted if both processors are to be used. The slave processor is restricted to user mode processing. All system calls and interrupts are handled by the master processor. If a user process running on the slave processor makes a system call, that process is rescheduled and the slave processor searches for a new process to run. This organization was adopted because of the difficulty of interlocking the UNIX kernel to run simultaneously on more than one processor. Interrupts are handled exclusively by the master processor and are locked out of the slave processor by running at a higher hardware priority than the device interrupts.

The performance of Purdue UNIX is affected by several issues. One set of factors is due to the fact that the system was expected to run more users and more processes than a single processor system as it is expected to do the work of two VAX's. It is therefore very sensitive to the amount of main memory present (too little caused excessive swapped and paging), the size of the file system block cache and the block size (more users means more active files and more disk transfers). Memory contention is another factor affecting system performance. The performance decrease from this contention is usually small (about 5%) because of the memory caching and instruction prefetch buffer on the processors. Programs which did intensive memory write operations (bypassing the cache) were observed to slow the system performance to the level of a single processor system. A serious performance problem is the restriction of one of the processors to user mode computation. The increased number of running processes

produce an increased number of system calls; kernel processing can become the bottleneck of the system. It is recommended that the heavily used application programs be reprogrammed to reduce the frequency of system calls. In a job mix composed of various compilations and text processing tasks, an improvement of 85 to 90 percent was reported over the throughput of a single processor system.

3. Hardware

The testbed hardware for this initial implementation of TRIX is a personal computer designed at the MIT Lab. for Computer Science, called the Nu computer[Ward80b]. The idea behind the Nu computer is to provide a flexible, modular system built around a general bus which can support a variety of processing engines. The system is also designed to support multiprocessor configurations, allowing a user to increase the available computational power available without additional expenses in power supplies, backplanes, and memory.

. The heart of the Nu computer is a bus supporting 32 bits of address and data. The arbitration and data transfer protocols of the bus are not dependent on the characteristics of any particular processor. Several different microprocessors can be interfaced to the bus and use the same memory and I/O hardware. All bus masters share a common physical address space and each has access to all memory and devices on the bus.

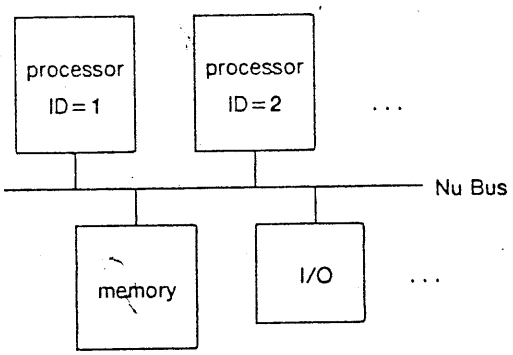


Figure 3.1. *Nu hardware configuration*

The bus provides for the operation of either homogeneous or nonhomogeneous multiprocessor systems. There is an arbitration for each bus cycle which decides which requesting bus master has use of the bus for that cycle. This arbitration phase can be overlapping the previous data transfer cycle to conserve bus time. Bus requests are granted through a daisy-chain mechanism starting at one end of the bus. Processors (and other possible masters) geographically closer to that end of the bus therefore have higher priority for bus cycles. However, the bus is designed to be fair; no processor can execute two bus cycles before any other requesting processor has access to the bus. This prevents processors at the far end of the bus from being starved of bus cycles and establishes a maximum latency that a

requesting bus master can wait before being granted a bus cycle.

Interrupts are implemented over the bus by a special mechanism called an *event cycle*. These are special 32 bit transfers which all processors monitor. Each interrupting device has an *event register* which can be loaded with an arbitrary data word. When the device requires service, it obtains the bus through the usual mechanism and broadcasts the contents of its event register. By convention the lowest four bits of the event data specifies the ID number of the processor to be interrupted. Processors with a different ID number ignore the cycle. The processor which matches the ID number reads the event data from the bus. Normally this data contains the vector number for the proper service routine. Thus a processor is assigned to a device interrupt by initializing the proper event register with its ID number and vector data.

The processor used in the initial implementation of the Nu computer is a Motorola MC68000 microprocessor running at 4 MHz. The processor module contains the microprocessor chip, memory management hardware, a queue for event data, and an interface to the system bus. Processor modules determine their identity by reading a four bit processor ID at a fixed physical address. This ID number is set by switches so each processor module can be given a unique ID number.

The MC68000 processor provides a Test and Set instruction for implementing synchronization primitives. This instruction appears on the bus as an atomic read-modify-write cycle. The integrity of the Test and Set is therefore preserved by the bus.

The memory management hardware on each processor consists of single level page table which performs the virtual to physical address translation. The processor is unable to properly deal with page faults so the system does not support demand paging. The current processor interface does not have a cache, although such an addition would be very useful in a multiprocessor configuration to reduce bus contention.

Upon initial reset, all processors execute code out of a bootstrap program stored in ROM. Each processor checks its ID number and branches to a different sections of code accordingly. One processor (ID number 1) enters a simple debugger and waits for a system initialization command. Other processors enter a loop polling a flag location in memory. Each processor waits until the flag contains its ID number and then begins execution at a location found in another reserved location in memory. Thus processor number 1 can selectively start up other processors by putting the desired starting address in the reserved location and writing the

processor's ID number in the flag location.

The performance of this particular hardware, although adequate, is not optimal for running a multiprocessor TRIX. The bus utilization of a processor module is fairly high. This is due to some peculiarities of the MC68000 and the lack of any caches. The result is that bus contention is a serious performance issue. Another drawback of the testbed hardware is relatively slow disk storage hardware.

4. TRIX

TRIX is an operating system designed to deal with the issues of communication and concurrency in a distributed environment. It is oriented towards users of personal computers interconnected by a network. It is organized to facilitate both inter-machine and intra-machine communication. The treatment of concurrent activity in TRIX is such that an extension to multiprocessor implementation can be implemented without impacting the semantics of the system. The following section discusses briefly the ideas behind TRIX and some structural details relevant to implementing a multiprocessor system. A more complete treatment of the issues addressed by TRIX can be found elsewhere[Ward80a,Sieber83].

4.1. Philosophy

Here are, briefly summarized, the main ideas behind the TRIX system.

- The operating system should provide a minimal yet efficient set of primitives supporting "process" management, communication, and scheduling control. These primitives can be considered an instruction set extension of the underlying machine, which implements the basic TRIX abstractions. Most of the functionality commonly associated with operating systems should be implemented with user level programming. Only the machine extension and interrupt level device interfacing need to be part of the system kernel.
- The communication mechanism of a system must be built in at the lowest level. Attempts to graft these mechanisms onto the semantics of existing operating systems have proven unsatisfactory for a variety of reasons[Sieber83]. In TRIX, the inter-process communication mechanism is a central part of the system abstraction and is supported at the level of the basic kernel primitives.
- The communication mechanism must properly provide for possible asynchrony between processes. TRIX was designed to deal with communication between processes on separate nodes in an environment in which processing nodes are loosely connected by a network. Messages and replys traversing a network are likely to suffer greater delays than intra-machine transactions. It is therefore crucial that the communication mechanism does not artificially serialize tasks not otherwise interdependent (for example by processing messages one at a time, not acknowledging new message until the current message has been completely serviced).

The current implementation of TRIX is an attempt to realize these goals. The next section provides an overview of the semantics of the TRIX system, introducing the TRIX model of

computation. It is followed by a description of some of the implementation details relevant to adapting the system to run on multiple processors.

4.2. Semantics

The object of TRIX is to provide a small but powerful set of mechanisms, essentially a machine extension, that can be used to build interesting systems. The machine extension manages a database of system objects provides the basic communication mechanism. Around this kernel are a very few *system handlers* which provide such functions as "process" synchronization and device interfacing. The majority of what is normally considered "system" functions are implemented at the user level.

4.2.1. Objects

In TRIX, what is conventionally though of as a "process" is divided into two objects, a *domain*, and a *thread*. Domains consist of an address space and a set of *handles*, essentially pointers to other objects protected and maintained by the kernel. Domains are passive objects; there is no stack, program counter, or register state associated with them. They may, however, have state in the sense of program data. Associated with a domain is a structure in the kernel data base which contains information about the state of the domain and the mapping of the domain address space. Domains are referred to via *handles*, which are kernel maintained pointers to a *port* (entry point) in a domain. Handles on domains are used to read from and write to domains, or to load the domain (from a handle on a file for instance). They also encapsulate the permitted communication channels at any instant. Everything in the system, files, directories, programs, etc., are referenced through handles. A powerful consequence of this is that various functions can be transparently interposed. For example, a remote file could seem to a user as handle on any other file, though the handle may actually be on a network server. The remote file server will process the same messages as a local file handle and will be transparent to the user.

Domains are created by a MAKE_DOMAIN kernel call. This call allocates and initializes a domain structure in the kernel data base. The domain is created empty, with a null address space. The address space can be later loaded though requests to the *domain handler*. Domains are not explicitly freed; rather, they are garbage collected when there are no longer any open

handles on the domain (at which point no further access to the domain is possible). Handles are closed by the CLOSE kernel call.

A thread is a single locus of execution in TRIX. It contains the execution stack, the program counters and current state of the task. Threads execute code in a domain, and are always resident in some domain. There are no restrictions on the number of threads in a domain. A domain can exist with no active threads. A single user activity is likely to include several domains (a file system and I/O interface are likely to be separate domains, for example) but only a single thread. Threads are created by a SPAWN kernel call, which instantiates a new thread in the current domain executing at a specified point. The new thread is created with an empty stack (in contrast to the UNIX fork() in which the stack is duplicated). There is no naming system for threads in TRIX; there is no way for one thread to explicitly reference another. Communication between threads must take place through a common domain; either though both threads executing in a single domain, or though data exchange through windows on shared domains. Threads are freed when they execute a REPLY from the same context in which they were SPAWNed.

4.2.2. Communication

The basic communication mechanism in TRIX is an inter-domain procedure call and return. These are accomplished through the kernel calls REQUEST and REPLY, respectively. A thread executing in a domain can transfer itself into another domain (perhaps to use some capability provided by code in the new domain) by means of the REQUEST kernel call. The return point in the calling (*requesting*) domain is saved on a protected request stack. The thread resumes execution at this point when a REPLY kernel call is encountered. The thread enters the new domain with an empty stack. The stack from execution in the previous domain is still existent but inaccessible. When a REPLY returns the thread to the requesting domain, the stack is restored to its previous state (before the request). Some information can be passed to the new domain. This includes a single handle (one of the handles owned by the requesting domain), a small number of arguments (uninterpreted data words), and a *data window*. This data window can be into any portion of the requesting domains address space, into the part of the thread stack active in that domain, or into the data window passed into the requesting domain by that thread. This window allows the new domain access to external data and is the basic data transfer

mechanism in TRIX. Values can be returned by a REPLY call through a similar mechanism.

Below is a list of some of the kernel calls and a brief description of their function.

MAKE_DOMAIN	Creates an empty domain and returns a handle. A domain is garbage collected when all handles on it are closed.
MAKE_PORT	Creates a port on a domain a returns a handle on it.
SPAWN	Creates a new thread with an empty execution stack. Threads are destroyed when they REPLY from the context in which they were created.
CLOSE(*handle*)	Frees the handle specified by its argument.
REQUEST(*handle*)	Transfers the executing thread to the port named by the argument handle, pushes the return point onto the threads request stack and protects the current execution stack.
REPLY	Returns a thread from the last REQUEST.
RELAY	Transfers the executing threads to the specified port without pushing any return information.

4.3. Structure

The structure of a TRIX system can be analyzed in three sections, the kernel, the system domain, and the user domains. The TRIX kernel provides the virtual machine model, which is a functional extension of the underlying hardware. It manages the fundamental TRIX objects; domains, thread, ports, and handles, as well as providing a set of communications/control flow primatives as kernel calls. Kernel calls run on a single kernel stack for all threads rather than maintaining a separate kernel stack for each process. TRIX kernel calls have the property that they do not block; the calling thread is never suspended in the midst of executing a kernel call.

The kernel maintains a data base consisting of two main sections. The first contains the state of all of the various kernel objects, domains, threads, etc., as well as that of physical resources such as memory. The second contains scheduling information, which threads are scheduled to run in which domains and which threads are waiting on some event. Most of the function of the kernel calls is to manipulate this data base.

The *system domain* is a special entity in TRIX. It appears to threads as a normal user

domain and is accessed through handles on ports by the same REQUEST/REPLY calls. It is considered separately because it directly accesses the kernel data structures. The system domain provide functions which need this coupling to the kernel. Handlers in the system domain include the *domain handler*, *sync handler*, and the low level device handlers (that portion of the device programming which interfaces the user with the interrupt level service).

The domain handler performs certain operations on domain objects. Among these are READing and WRITEing the domain address space, and LOADing the domain address space from a file (actually a handle). The domain handler uses the kernel data base in the WRITE and LOAD operations, for example, as these can allocate additional physical memory in order to expand target domain segments.

The other handler which closely interacts with the kernel is the sync handler. Synchronization of threads in TRIX is accomplished through two requests *sleep(address)* and *wakeup(address)* on a sync handle. When two threads wish to synchronize, they must request and share a sync object. The *sleep* request removes the thread from the list of scheduled threads in its current domain and puts the thread on a special list of sleeping processes. It records the address on which the thread is sleeping and the sync object (handle) the *sleep* used. The thread stays on the sleeplist until a *wakeup* occurs on the address using the same sync handle. A *wakeup* replaces any thread on that address-handle pair back on the active schedule lists. The *wakeup* will have no effect on a thread sleeping on a different sync handle or a different address. The address slept on is by convention the address of a shared object to be locked. The semantics (though not the implementation) of this mechanism are similar to those of the UNIX **signal()** and **wait()** system calls[Thompson78].

Within the system domain are also the handlers which interface the user to the I/O device interrupt processing. User threads treat these handlers like any other and the REQUEST/REPLY mechanism is used. These handlers are in the system domain because the user side must be carefully synchronized with the interrupt processing. Actually, two levels of synchronization are necessary. The first prevents indeterminacy due to user-interrupt conflicts on data structures (an interrupt occurring when an important structure is in an inconsistent state due to incomplete user processing). These conflicts are resolved by means of a hardware priority mechanism with which the user thread can lock out interrupts during the execution of a critical section (the interrupt does not need to lock out user processing as this is implicit). The

second level of I/O synchronization results from threads waiting on devices for an I/O transaction to complete. This is accomplished via a mechanism with semantics identical to the *sleep* and *wakeup* described earlier. The implementation is slightly different because the interrupt servicing is done in the kernel rather than by a legitimate thread, and therefore cannot REQUEST into the sync handler.

The user domains contain most of the code and data of the system. This includes a large amount of what is commonly considered "system" functionality, such as teletype drivers and file systems. This organization reduces both the size and complexity of the kernel and allows users to supply their own programs providing this functionality.

4.3.1. Scheduling

Active threads are scheduled to run in a specific domain. Each domain, as part of the associated kernel data structure, has a list of threads scheduled to run in it. When free, the processor searches these lists for runnable threads, with the system domain always searched first. If there are no runnable threads in the system domain, scheduling proceeds in round robin fashion through the list of user domains.

Within a domain, threads are run according to a priority system. Associated with each thread is a priority word, which determines its scheduling properties. There is a similar word for each domain, and the priority of a domain is set to that of the thread currently running in it. The priority word contains a numerical priority and two property bits. The rules regarding scheduling within a domain are:

(1) only the highest priority thread(s) in a domain are selected to run,

(2) a high priority thread will never be pre-empted by a thread with lower priority, and

(3) a high priority thread will be run before any lower priority thread not currently running, but is not guaranteed to pre-empt a lower priority thread already running.

The two property bits specify additional information regarding pre-emption. One bit, when set, prevents the thread from pre-emption by another thread at the same priority level. The other bit is used to "lock in" a thread's priority into a domain. Even if the thread requests into a

different domain. the "locked" domain behaves, with respect to scheduling, as if the thread were still running in it. Thread priorities are raised and lowered by means of the SPL kernel call (kernel calls are atomic with respect to thread scheduling and pre-emption), which sets the thread priority to the specified level.

4.4. Status

The current implementation of TRIX operates as a single user system. It runs a UNIX-like command interpreter and many of the UNIX utilities. A set of C libraries and write-arounds allow most UNIX programs to run on TRIX with no changes. The system runs on the 68000 based computer described earlier and runs UNIX utilities (for example ls) about 90% as fast as UNIX on the same hardware.

5. Multiprocessor TRIX

The inspiration behind multiprocessor TRIX is straightforward. A TRIX thread represents a separate path of activity. Therefore it should be possible to execute several threads simultaneously on different processors. The goal of this project is to produce a multiple processor implementation of TRIX which efficiently exploits this concurrency. In order to realize this goal, several issues which determine the organization of such a system must be resolved. The two major strategy issues which distinguish the system are:

- How should the system's tasks should be divided amongst the processors and,
- How the processors are to communicate with each other.

The philosophy of multiprocessor TRIX is that all processors are created equal. The system is symmetric with respect to the way it deals with processors. Any processor can perform any task in the system, including executing kernel calls, handling I/O requests, and servicing interrupts, as well as running user code. There are other possible approaches to this issue. One alternative is to establish a fixed pairing between processors and tasks. Under this discipline, processors might be dedicated to kernel processing or interrupt handling, or perhaps to running some commonly used subsystems, such as the file system or network server. A major disadvantage of this organization is that processors will often be idle because there is no demand for its assigned activity. The percentage of processing power devoted to each task is built into the system and is not able to adapt to the needs of users. A symmetric organization allows processors to perform and task necessary. The percentage of processing devoted to particular activities will follow the pattern of demand from the user.

Another approach to managing multiple processors is to use a "master-slave" organization as in the Purdue UNIX system. In this scheme, one of the processors is the "master" and performs all "kernel" processing, that is all kernel calls, interrupt servicing, and I/O processing. The "slave" processors are restricted to user level processing. There are two disadvantages of this organization. First, kernel level processing can easily become a performance bottleneck. Although all of the processors can be generating system calls, only a single processor can service them. In a busy system, all of the processes could be waiting for kernel access leaving the slave processors nothing to do. The other problem with the master-slave organization is that

slave processors must suspend execution of a process when it requires kernel access. Slave processors must therefore perform a context switch at every system call, since it must find a new process to run. This high rate of context switching decreases the effective performance of the slave processors. The main advantage of the master-slave organization is that it is fairly straightforward to implement. It does not require the system kernel to be interlocked for simultaneous execution by multiple processors. This interlocking is difficult to impose on UNIX. Because of this, Purdue adopted the master-slave organization in spite of its disadvantages. The structure of TRIX is very different from that of UNIX. Interlocking the TRIX kernel and low level I/O is quite feasible. This makes the master-slave system much less attractive since the symmetric organization is more efficient.

The second major design issue is how the processors should communicate. In this version of multiprocessor TRIX there is no direct interprocessor communication. All interaction is moderated by the system through shared memory. No processor is able to refer to any of the others; no processor is aware of the existence of any others (except for delays in access to kernel objects). This has a number of nice features. For example, the number of processors is not built into the system; it can be booted with any number of processors in the backplane. Also, a system which relies of direct communication will have processors waiting for replies from other processors, which not only wastes processor cycles but can lead to deadlocks. It may, however, be useful at some time to add an special mechanism by which one processor can cause an interrupt in another, thus allowing high priority threads to pre-empt running lower priority threads.

In order to implement the decisions described above, several modifications must be made to the basic TRIX system. The following additions are needed:

- A framework which allows multiple processors to run in the kernel.
- A means of synchronizing several processors executing simultaneously in the kernel.
- A scheduling system to assign processors to runnable threads.
- Mechanisms for synchronizing threads (at user level).
- A strategy for handling interrupts.
- A scheme for initializing the TRIX system and gracefully bringing all processors into operation.

Each of these issues requires both a policy (or general philosophy of operation) and a mechanism (particular implementation).

5.1. Kernel Structure

Since all processors are treated symmetrically, each processor must be prepared to handle kernel calls. Each processor has its own kernel stack mapped into the same section of virtual address space. Each processor handles traps and interrupts independently on this stack. All processors share the same kernel code and data base, mapped into the same virtual address space. This data base contains the state of all kernel objects, and information regarding the scheduling of threads in domains. It is through this data base that the processors know what is going on in the system, and can find out what task to perform.

Each processor has a certain amount of local data. This includes some memory locations built into the CPU hardware, in particular a Processor ID register, a Bus Error register and a page map. Each processor also has a structure containing some system data:

```
struct proc_data{
        pointer T_CURRENT;
        pointer T_MAPPED;
        pointer D_CURRENT;
        pointer D_MAPPED;
        int dosched;
}
```

This structure contains pointers to the current thread run by the processor and the current domain in which it is executing. It also contains pointer to the domain and thread (if any) that are currently mapped in processor's page map, and a flag indicating that the processor needs to be rescheduled (set during some kernel calls). Ideally this "per processor" data structure would be kept at some level in kernel stacks. However, this is inconvenient in the C language so they are implemented as arrays in the kernel data area. Each processor uses its ID number (from the register on the CPU) as its index into the arrays. Another alternative is to keep this data in separate physical pages, mapped to the same virtual address.

The major resources in a TRIX system are the physical memory and devices and the kernel objects: threads, domains, handles, ports, and synchronizers. These are allocated and freed (or collected) by the TRIX kernel and handlers of the *system domain*. In most cases the allocation or de-allocation is requested explicitly by a running thread and can therefore be performed by the processor executing the thread. In order to implement this resource management, the kernel must be properly interlocked to insure that the data base describing the various kernel objects remains self-consistent.

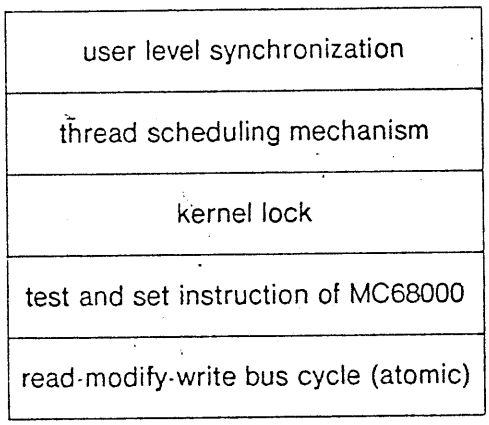
Each thread has its own stack and state information. This information is kept independently from the kernel stacks; thus a single thread can be run alternately by many processors (for example, a thread waking up from an I/O wait need not run on the same processor that put it to sleep).

5.2. Synchronization

There are two separate synchronization issues in multiprocessor TRIX. One involves the interaction of concurrent threads, the other concerns the synchronization of processors executing in the kernel. When two concurrently running threads are executing the same code or accessing the same data, there is a need for synchronization to avoid indeterminacy and data access conflicts. This problem also occurs in single processor TRIX and is dealt with by the

user with synchronization primitives provided by the system. It must be approached with special care in a multiple processor environment since often programs implicitly assume that there is only a single processor. Additional primitives may be found useful for conveniently managing thread interactions. The second issue, processor synchronization, is only relevant to multiprocessor TRIX. Since all processors can execute in the kernel, it is essential that it be properly interlocked. Any failure in this level of interlocking could result in a complete deadlock or system crash.

This system uses several levels of synchronization or mutual exclusion primitives. Each level is built from the primitives beneath and serves as a foundation for higher level functions.



The lower levels of this structure are dependent on the hardware. Atomicity at the lowest levels is essential. The kernel lock is the basic processor synchronizer in multiprocessor TRIX. It is uses a binary semaphore implemented with the test and set instruction of the processor. This locking mechanism keeps the kernel data base consistent and mediates the progress of threads as they transfer between domains through the kernel. The scheduling mechanism, by enforcing the pre-emption discipline, provides for synchronization at the thread level. The *sync handler* provides the **sleep** and **wakeup** requests described earlier. The following section examines the interprocessor kernel interlocking, thread synchronization is discussed later.

5.3. Kernel Synchronization

There are several design constraints on the implementation of kernel interlocks. First, and most important, the kernel must be safe from any possibility of deadlock. Deadlocks are caused by processors waiting for each other to relinquish resources. Were this to occur, the processors involved would cease useful computation until the system was restarted. A deadlock in the kernel would quickly freeze all of the processors in the system. Another constraint is that the kernel data base must be kept in a valid and consistent state at all times. Inconsistencies due to write-write conflicts or read-write conflicts are fatal to the system.

In order to maximize the performance of the system, it is desirable to maintain as much parallel activity as possible. Task execution should only be serialized in response to the constraints of that activity. This is difficult in the kernel, since all processors running in the kernel are competing for the same database. There is a legitimate mutual exclusion constraint between them. Another efficiency issue is the amount of time a processor spends requesting resources and waiting on them. This should be kept as small as possible.

5.3.1. Loci of Interaction

Most of this interaction is competition for access to two databases, the kernel object database and the thread scheduling lists. This section details the various points in the TRIX kernel where processor interaction can occur.

One place where processor synchronization is necessary is in TRIX kernel calls. These allocate physical memory and kernel objects such as domains, threads and handles. Access to the the corresponding data structures must be interlocked. Kernel calls also modify the active threads lists used for scheduling threads in domains. For example, the SPAWN call creates a new thread and schedules it in the current domain. Calls such as REQUEST and REPLY unschedule a thread in its current domain and reschedule it in the target domain. These lists of scheduled threads must be maintained in a consistent state. Therefore, these sections of code must be kept mutually exclusive.

In addition to kernel calls, a small number of handlers in the *system domain* also change the kernel data base. The domain handler may allocate physical memory when handling LOAD and WRITE messages. Processors executing in the *domain handler* must therefore be synchronized with those performing kernel calls. The *sync handler* alters the lists of active and

sleeping threads as is accepts **sleep** and **wakeup** requests.

Also in the kernel are the lowest level device drivers. These consist of the interrupt routines for the devices and the handlers which interface to them on the user's side. There are two possible sources of synchronization problems in these routines. The first is the possibility of read-write conflicts on data objects if both the interrupt routine and user handler are executing simultaneously. The second source of problems in the I/O routines involves communication between the interrupt and user sides. The user side occasionally **sleep()**s, waiting for the completion of some I/O transaction, an event which is signaled by a **wakeup** from the interrupt handler.

User - interrupt exclusion is necessary in a single processor system, but presence of additional processors complicates its implementation. The user thread can no longer lock out interrupts by raising the hardware priority of its processor since the interrupt may be handled by a different processor, one over which the thread has no control. Conversely, the interrupt handler can no longer assume that all user processing stopped since it will continue on other processors.

The fact that both user and interrupt processing on the same device can occur simultaneously also creates problems for their synchronization. Typically the user handler tests some condition, such as buffer full or buffer empty, and **sleeps** until the interrupt routine performs a **wakeup**. The problem arises when both are executing simultaneously. The **wakeup** may be performed after the user side checks the deciding condition but before the corresponding **sleep** is accomplished. The result is the thread will **sleep** forever, waiting for an event that already occurred. The "test and sleep" on the user side of a user-interrupt interaction must be made an atomic operation to avoid such deadlocks.

5.3.2. Some Implementation Issues

One use of the kernel locking mechanism is to make mutually exclusive the sections of code where processor conflict can occur. There are several issues to be resolved in implementing this mutual exclusion. One is the granularity of interlocking. Kernel calls generally have several critical sections of code interspersed with non-critical sections. These could be interlocked separately or together as a group. Furthermore, some critical sections in these calls deal with resource allocation and others with scheduling. It is possible to use different locks for

each type of critical section or to group them together.

The advantage of separately locking each critical section is that by locking only those sections absolutely necessary the latency of waiting processors is reduced. The disadvantage is that a processor will have to request a resource several times in the course of executing a single call. This increases the overhead of interlocking, and the waiting time of the processor.

The possible advantage of using separate locks for each type of critical section is that it increases the potential parallelism. The difficulty with using several different locks in the kernel is that it introduces the possibility of deadlocks. If a kernel call requires more than one resource (as most do), it must either release one before requesting the other or request both at the outset. Releasing resources is impossible since this would leave unprotected data in an inconsistent state. Requesting both locks is a serviceable approach, but if two locks are always requested together, there is no advantage to having different locks. In the interest of simplicity, they can be combined with no loss.

5.3.3. Kernel Lock Routines

The kernel locking mechanism uses two global variables for each lock, an integer which holds the processor ID of the processor which currently holds the lock and a binary semaphore. The processor spins in a tight loop until it obtains the lock.

The interlocking of the kernel is accomplished using two routines **lock()** and **free()**. The **lock** routine returns the previous value of the lock (presumably this is 0 or the ID of the processor owning the lock). It does not return until the processor is able to obtain the lock. Processors waiting for resources are looping in the **lock** routine. The value returned allows the locking processor to return the lock to its previous state when freeing the resource. It is important that a processor running interrupt routines does not blindly free locks that may have been locked by that same processor executing user code or kernel calls. The routine **free(old)** frees the lock only if **old** is zero. If **old** is different from zero **free()** merely checks that the processor trying to free the lock is the one that owns it, but otherwise leaves the lock unchanged.

A simple semaphore is insufficient for implementing the lock routine. There are circumstances under which a processor will try to lock a data base it has previously locked. For example, this can occur if a processor receives an interrupt while executing a kernel call (

during which the data base is locked). The interrupt handler may also need to insure that the data base is locked to prevent conflicts with other processors. If a semaphore is used to implement the mutual exclusion, the interrupt side will find the data base locked, even though the same processor in user mode locked it. This situation results in a deadlock since that processor is waiting for itself to release a resource. A solution to this problem is to keep a variable in the lock which holds the processor ID of the current holder of the lock. This will prevent any processor from waiting on itself. Of course if it is essential that a critical section of code not be affected by interrupts, the processor's priority must also be raised to prevent it from taking interrupts. This is not done by the **lock()** routine since it is only necessary in a few sections of code and would greatly increase interrupt latency.

The test and set on the lock variable must be atomic. This is insured by using a binary semaphore to implement mutual exclusion on the lock variable. The code for the lock routine is given below in **C**.

```
int locknum;          /* owner of lock */
char sem;             /* semaphore for lock */
lock(){
     if(locknum == mynum()) return(locknum);

     while(1){
          n = spl7();
          if(locknum == mynum()) s = locknum;
          wait(sem);
          if(locknum == 0){
               locknum = mynum();
               s = 0;
          }
          signal(sem);
          splx(n);
          if(locknum == mynum()) break;
     }
     return(s);
}
```

The signal() and wait() routines are the standard binary semaphore routines, **P** and **V**, implemented using the test and set instruction of the MC68000. This instruction provides the low-level atomicity for the synchronization mechanism. The semaphore is used to provide mutual exclusion around the test and set of the ownership variable **locknum**. The **spl7()** sets the hardware priority of the processor to 7 (the highest priority). The **splx()** restores the priority to

its previous value. Raising the priority is necessary to prevent interrupts from occurring after the execution of the wait() but before the execution of signal(). Were this to happen and the interrupt handler tried to acquire the lock (in order to wake up a sleeping thread), a deadlock would occur, since the processor would be waiting on itself. Interrupts are only disabled while waiting to execute the test of **locknum**. Since there are only a few instructions in this critical section, the amount of time interrupts are disabled is small. If the lock is in use, interrupts are re-enabled before trying again.

The value returned is the previous value of the lock. This is used by the **free(old)** routine to restore the lock if necessary. The **free(old)** routine checks that the processor really owns the lock, then clears the **locknum** variable if its argument is zero. No mutual exclusion is needed in the free routine.

5.4. Scheduling

There are two problems introduced into the TRIX scheduling structure by multiple processors. The first is simply how the processors are to be paired with runnable threads. Key issue are safety and fairness. The implementation must insure that no thread is permanently ignored, and that no thread is run by more than one processor at a time. The second problem is to maintain the TRIX semantics of priority and pre-emption when it is possible for several threads to be running simultaneously in a domain.

In order to insure the safety of the system, the multiprocessor scheduling has been designed with a state model of threads. Threads can be in one of three states, SLEEPING, RUNABLE, or RUNNING. Threads are RUNABLE if scheduled to run at some priority in some domain, RUNNING if currently paired with some processor, SLEEPING if waiting on another thread (or I/O).

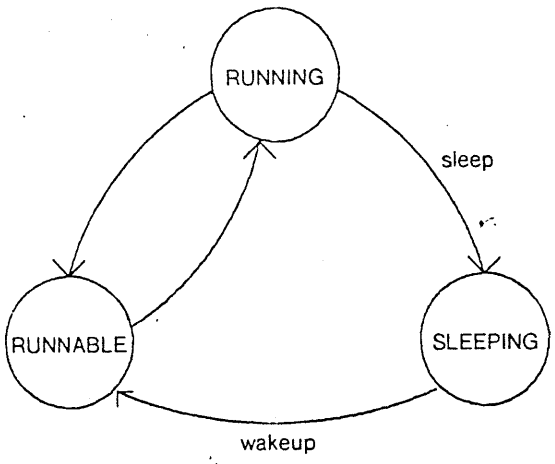


Figure 5.2. *State diagram for threads*

Threads change from RUNABLE to RUNNING when a processor in the scheduling loop begins to run the thread, from RUNNING to RUNNABLE when that processor is rescheduled. State transitions into and out of SLEEPING occur when threads are put to sleep or awakened by the sync handler or I/O routines. Threads executing a **sleep** change from RUNNING to SLEEPING. A **wakeup** returns threads to the RUNNABLE state, but these threads are not run until paired with processors.

The state transitions of domains and threads must be carefully protected. Otherwise it is possible, for example, for two processors to find the same thread RUNABLE, and attempt to begin running that thread at the same time. This will invariably crash the system as the thread state will become hopelessly garbled. The key to protecting these transitions is to insure that the operation of testing and conditionally changing a thread's state always atomic (with respect to rescheduling and pre-emption) and only performed by one processor at a time. This is implemented using the kernel locking mechanism since transitions only occur from the system domain or during kernel calls.

Idle processors wait for tasks in a loop, searching through the schedule list of each domain for a runnable thread. Each processor looks for the next RUNABLE thread to be executed (searching round robin through domains and checking the highest priority thread in each domain). If a suitable thread is found, the state of that thread is set to RUNNING and the processor begins to execute code for the thread.

The presence of multiple processors introduces some complications into the scheduler implementation, which are necessary to preserve the TRIX semantics. For example, one consequence of the pre-emption rules is that a running thread is guaranteed that no thread of lower priority is also running in the same domain. This is an important feature since it allows for mutual exclusion between threads in a domain (A thread running at the highest priority should be the only thread running in its domain). However, in a multiprocessor system a thread at high priority may request into a domain in which a lower priority thread is already running on a different processor, which would result in a violation of the guarantee stated above. Similar problems can arise if one of two threads running in a domain executes an SPL to a high priority.

A solution to this problem is to monitor all points at which the priority relationships within a domain can change. These include priority changes through SPL and thread transfer because of requests. Threads rescheduled in a domain by a wakeup are not a problem since they are not RUNNING when they appear. In order for a thread to be run by a processor, it must not only be the highest priority RUNABLE thread, but the domain in which it is to run must not contain any RUNNING threads of lower priority. This condition is checked in the kernel before a thread is selected to be run. The priority of a domain is also checked when a thread REQUESTS into it. A lower priority thread will be placed on the schedule list when entering a high priority domain but will be set to RUNNABLE and not run until all higher priority threads are gone. Similarly, a high priority thread entering a domain in which a lower priority thread is running will be marked RUNNABLE and its processor rescheduled. When the currently running threads stop (due to being put to sleep or the occurrence of a scheduling tick), it will be the next thread to run in that domain. Threads changing their priority with an SPL call are also returned to the RUNNABLE state if it is not appropriate for it to run at the new priority. This system preserves the pre-emption semantics of single processor TRIX, allowing mutual exclusion between threads within a domain.

5.5. Thread Synchronization

The issues concerning thread synchronization are largely the same as in single processor TRIX. In both cases, threads meet in a common domain to coordinate activity. The problems of simultaneous execution exist in both cases since the order of execution of threads with the same priority is not defined. In a multiple processor system both may be executing the same

instruction at the same time. Philosophically, the synchronization of independent threads in TRIX is entirely the responsibility of the user. The semantics of TRIX enforces no policy in this area; indeed much of the flexibility of TRIX comes from this fact. It does, however, provide sufficient mechanism for interlocking thread activities.

5.5.1. Priorities

The two mechanisms for synchronizing the threads are the sleep/wakeup requests to the *sync handler*, and the SPL kernel call. The sleep and **wakeup** requests can be used to order computational events between threads. Presumably, one thread checks for the completion of an event and sleeps on it (actually on some associated address). Another thread, upon causing the event will wake up all threads sleeping on it. A danger arises when the sleep and **wakeup** are to be performed by threads running simultaneously in the same domain. It is possible for a **wakeup** to occur while a thread is deciding to sleep on the same event. In this case, the sleeping thread may never be run again, since the **wakeup** it is waiting on has already occurred. For this type of synchronization to work properly it is essential that one thread be able to exclude any other thread from performing a **wakeup** (or anything else) while it is deciding to sleep.

A similar problem exists in the interaction of the low level I/O handlers with interrupt service, requiring locking at the kernel level. Thread-thread mutual exclusion is achieved by use of the SPL kernel call. A thread using SPL to set its priority to the highest possible (and keep threads on the same priority from pre-empting) will be the only thread running in its domain after it returns from the call. The call itself is atomic with respect to pre-emption. Because of the scheduling rules, even if two threads (on different processors) begin executing the call at the same time, only one will return from the call still RUNNING. That thread will retain control of the domain.

It is possible for a thread to lock a domain across a request into another domain. This is done by setting the priority bit which "locks in" the current threads priority. Other threads are prevented from running in that domain until the requesting thread returns and lowers the priority. Since this mechanism allows a thread to retain one resource (domain) while requesting another, it can lead to deadlocks. For example, if two threads simultaneously lock their current domain and request into each others domain, it is possible that neither one will be able to run again.

Since such a situation will not crash the system, but merely inactivate the threads involved, it is not precluded by the system. It is the responsibility of the user to see that it does not occur.

5.5.2. Other handlers

One disadvantage of using the SPL to implement mutual exclusion is that it locks the entire domain. It may be useful to be able to lock specific data structures or sections of code within a domain while allowing free access to others. In a teletype handler, for example, requests on different devices can be processed concurrently, but each device can only deal with one request at a time. One possible mechanism for dealing with interlocking at this level is a semaphore handler built into the *system domain*. It would accept requests which implement the basic **signal** and **wait** semaphore primitives, as well as **create** and **close** requests. Using the same *sleeplist* as the *sync handler*, **wait**ing threads would be put to sleep so as not to consume processor cycles. The corresponding **wakeup** would come from the **signal** request.

5.6. Interrupts

Device interrupts can be handled by any processor. The details of the mapping of interrupts on to particular processors is highly dependent on the hardware. The hardware on which TRIX is currently implemented allows each interrupting device to be programmed with the ID of the processor to be interrupted as well as a service routine vector. An interrupt by the device will only be detected by the processor with that ID. All processors share the same table of interrupt vectors and handling routines. The devices are usually programmed with the ID number of the processor which initializes them and this processor will service the interrupts.

Other hardware schemes can also be used. The system could be set up such that all processors respond to all interrupts but use different vector tables. All but one processor would immediately return leaving the proper processor to service the interrupting device.

Interrupts are handled using the kernel stack of the servicing processor, so several processors can take different interrupts without interfering. However, as discussed in the previous section, proper synchronization between user and interrupt sides of a task is critical.

5.7. System Initialization

A exception to the rule that all processors are considered equivalent is the startup sequence. Turning all processors loose on the system on power up would result in chaos. In the present hardware, on system reset one processor (the one with processor ID 1) executes a bootstrap routine while all other processors wait in a tight loop. The active processor starts the TRIX system. This task includes initializing the kernel data base, creating a domain for the system handlers, and starting up an initial thread. It also initializes the devices and file system, although this could actually be done at a later point by any processor. When the system is in the proper state, with a legitimate thread running in a domain, the initializing processor wakes up another processor which has been looping on some location since reset. Each new processor executes some initialization code (which sets up its page map and clears interrupt fifo's), wakes up the next processor, and enters the scheduling loop looking for something to do. Each processor wakes up the processor with the next higher ID number until all processors are active. It is therefore not necessary for the system to have built in the number of processors currently running. Of course the processors must have sequential ID numbers for this to work.

5.8. Garbage Collection

The TRIX used in this thesis is, at present, an experimental system. Several features must be added to make TRIX useful to a larger user community. Some of these features impact multiprocessor operation and this must be considered during their implementation. The most important of these is garbage collection.

Garbage collection on TRIX refers to the reclamation of system resources no longer in use, particularly the collection of domains. A domain can be recycled when it is impossible for any references to be made to it. This is the case when there are no outstanding handles on the domain (any of its ports). Garbage collection involves checking the active handles list, marking the referenced domains and collecting the rest. The main issue here is that the handle data base could be changing during this process due to the activities of processors other than the one doing the garbage collection. Since there is no inter-processor communication mechanism, other processors cannot be halted for the garbage collection. However, one property of the system is that if there are no outstanding handles on an inactive but uncollected domain, no handles can be created on that domain. Therefore no inactive(collectible) domain will ever

become active again. Garbage collection can therefore proceed independently from other activities. If the garbage collector is carefully constructed in the way it examines the handle data it is not necessary to lock the data during garbage collection. At worst, a newly closed domain will remain uncollected until the next time the garbage collector is run.

5.9. Language Issues

There are a number of issues brought up by the interface of TRIX semantics with programming languages. TRIX is written in the language C[Ritchie78], the system language of the UNIX operating system. C is essentially an unscoped language; all data is either local to a given subroutine, and resides on the stack, or global to all routines, and resides in the data segment. This conflicts somewhat with TRIX semantics. A TRIX "process" is split into a thread and domains. Ideally the domains, supposedly passive objects, should contain only that data essential to the domain function. This includes the domain text and those data objects which are shared by all threads running in the domain. Context dependent data, data which each thread needs independently, should be associated with threads rather than the domain. This data should reside on the thread stack. The C language imposes the scoping issue on this decision since all variables which are to be accessible to several levels of subroutine are put into the domain. This mixing context dependent and context independent data in the domain has several consequences. One is that the domain must either be restricted to run a single thread at a time, or provide separate copies of context dependent variables for each thread.

A similar problem arises when the TRIX kernel is run on multiple processors. In this case the context is not associated with threads but with processors. There is a set of data objects which must have an instantiation for each processor. Ideally, these could reside on the kernel stacks. However, due to the global significance of these objects (many kernel subroutines need them), C places them in the data segment. It is therefore necessary to maintain an array of these objects with one entry for each processor.

Aside from the problem of duplicating data, there is also a semantic issue. TRIX domains are supposed to be static objects, although they do contain some state. The static nature of domains contributes to the ease with which they can be swapped. This static nature is weakened by the addition to the domain of non-essential state. One solution to the problems outlined above is to implement TRIX and the code in the user domains in a lexically scoped

language that allows a global area for shared data.

Another issue involving the interface between TRIX and programming languages concerns the semantics of SPAWN. A new thread is created through the SPAWN call with an empty stack. This makes the instantiation of threads very efficient since no copying is involved (as in the UNIX fork() call). However, it could create problems when used with some types of languages. Since new threads are created completely empty, they have no context in which to resolve non-global references. This is not a problem in C if threads always begin running at the start of a function. In a lexically scoped language, though, it could be disastrous if a thread were SPAWNed running in one of the inner levels of the lexical nesting. This thread would be unable to properly resolve references to variables in any of the surrounding levels. Essentially, threads must be SPAWNed to legitimate entry points into the domain, either explicit or implicit ports. This constraint, however, limits the usefulness of SPAWN as a multiprogramming primitive as it cannot be arbitrarily invoked, to provide parallel execution, at any point of the program.

6. Performance

The current TRIX systems, both single and multiple processor, are experimental versions to demonstrate the feasibility of the TRIX concepts. Although structured to run efficiently, the TRIX code has not been extensively optimized. The particular hardware on which multiprocessor TRIX has been developed is not very well suited for running multiple processors in a production environment. Memory contention is a serious performance problem due to the length of the processor bus cycles and the lack of any caching on the processor modules. This particular implementation of the Nu computer was used for reasons of expediency rather than performance; the system was available and supported multiple processors in a reasonable manner.

These considerations aside, it is nonetheless important to characterize the performance of the system and attempt to predict its performance on some more suitable hardware. In the ideal, a system running with *n* processors would have *n* times the throughput of a single processor system. Unfortunately this is often not the case. Due to several factors, mostly involving the processor interaction, the real performance of a multiprocessor processor system falls somewhat short of the ideal. This section attempts to analyze the major factors which affect the performance of multiprocessor TRIX. The goals of this discussion are threefold,

(1) To characterize the performance of the current implementation;

(2) To predict the probable performance of TRIX on other hardware, and,

(3) To discuss various optimizations which could improve the system.

The significant factors affecting system performance can be divided into two sets. One contains the effects caused by the contention of multiple processors for various shared resources. These are bus contention, and contention for use of locked sections in the kernel. These contention issues are very important because they decrease the effective throughput of each processor. Thus a two processor system will not achieve, in some sense, the computing speed of two separate one processor systems. The second set of performance factors are those that affect the level of processor utilization. It includes contention and synchronization at the thread level. These latter effects are very difficult to quantify since they depend in large part on the programming style and system use patterns of the user.

The effect of processor contention on system performance is characterized by considering the resulting decrease in effective throughput of each processor in the system. That is, each processor will actually be able to provide a certain percentage of the throughput (instructions / sec) it could in a single processor system. This percentage describes the effect of contention on that processor. The effective computing power of an N processor system would therefore be $Np_b p_k$, where p_b and p_k characterize the effects of bus contention and kernel lock contention respectively. The two factors p_b and p_k are, in general, functions of N.

6.1. Bus Contention

Bus contention significantly affects the throughput of multiprocessor TRIX on the current hardware. The degree to which it slows the processors depends on the frequency of bus accesses, which it turn depends on the programs they are running. The effect of bus contention on the current Nu hardware was measured running a compute bound program, *PUZZLE*. The program was timed on a single processor system and again on a two processor system. On the two processor system, both processors ran the *PUZZLE* program but only one was timed. The ratio of computation times, two processor over single processor was 1.4. Due to bus contention, the throughput of each processor is reduced to 71% (1/1.4) of its value without contention. The maximum throughput of a dual processor system, if both processors are utilized completely, is therefore 142% of that of a single processor system. While this is an increase over a single processor system, it is unacceptably low.

6.1.1. Caches

One technique for reducing bus contention is the use of caches on the processor modules. Caches reduce both the average memory latency seen by the processor and the amount of bus traffic it generates. However, with multiple processors there is the problem of maintaining consistency between multiple copies of data.

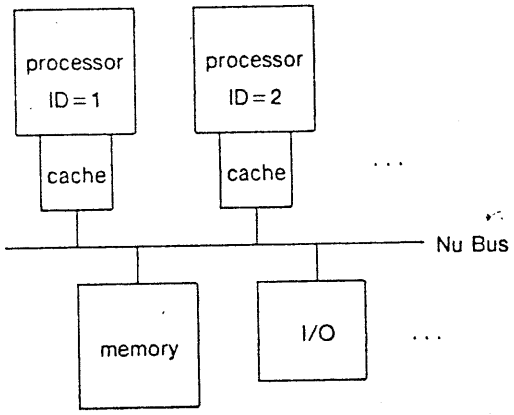


Figure 6.3. *Cached Processors*

There are several strategies for dealing with this issue. One is to design the caches to monitor all transactions over the bus and update or invalidate cache entrys whenever the cached location is written. This of course implies that all caches must be write-through. The disadvantage of this scheme is that it greatly complicates the cache hardware necessary. A simpler mechanism is to not cache all writable shared pages. In the TRIX environment this can be done fairly conveniently. In general, domain text and thread pages can be cached with the exception of those passed in a REQUEST data window. The cost of context switching is increased by such caching since the processor's cache must be invalidated (or at least part of it) when transferring into a new domain or executing a new thread. The text segment of the kernel is read only and hence can be cached, but the kernel data segment cannot be cached since it is shared by all processors and frequently altered. Because the number of shared, writable pages is relatively small and it is known which pages are shared, the expedient of not caching these pages should be effective.

6.1.2. Modeling the Effect of Contention

Bus contention causes an increase in the average bus cycle time seen by the processors. This increase is due to two different sources. One is that the bus is characterized by a maximum bandwidth (number of bus cycles per second). If the number of cycles per second desired by the processors exceeds this, cycle requests will be queued by the bus arbitration mechanism. This will cause the average length of bus cycles to increase and therefore cause the effective speed of the processors to decrease.

The second effect depends not on the average rate of memory accesses, but on the variation of the access rate over short periods. If all processors in a shared memory system had a constant, identical access rate and their total access rate did not exceed the bus bandwidth, they would not experience any increase in cycle time at all. During an initial startup transient, the requesting times of the processors would synchronize. After this phase, each processor would have its own time slice of the bus and would not face any further competition for its cycles.

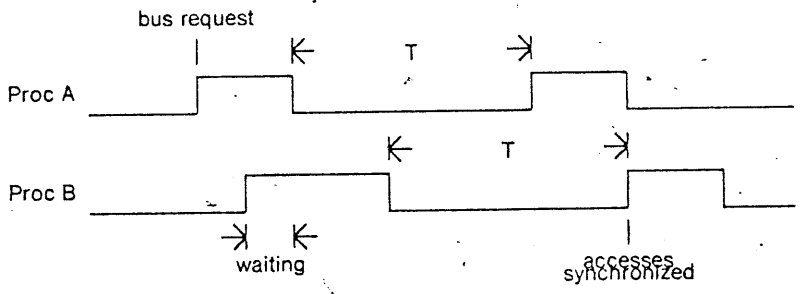


Figure 6.4. *Processors with constant inter-access time T*

Processors are slowed by bus contention even on unsaturated busses because their access rates are actually subject to variation. This means some fraction of a processors cycle requests will occur when the bus is busy. When this occurs, the requesting processor must wait until the current bus master is finished.

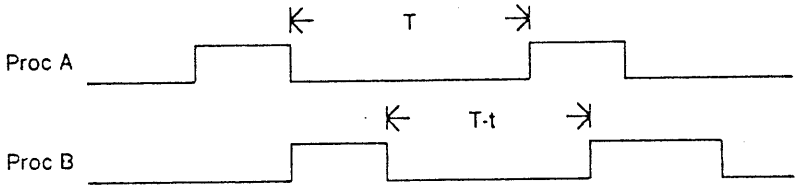


Figure 6.5. *Effect of variable inter-access time*

It is also possible that this processor may have to wait for the requests of other queued processors to be serviced. Some variation is found with an uncached processor because of the

differences in execution times and operand accesses among instructions. The variation is greater with a cached processor since a cache filters the stream of memory requests from the processor, sending only a fraction of them on to the bus. Cache misses tend to occur at random intervals as seen from the bus.

Models predicting the behavior of a bus contention system can be mathematically very complex. However, a model which is both mathematically tractable and applicable to interesting systems can be developed using results from queueing theory. The model assumes an unsaturated bus serving requests from multiple, cached processors. The bus is treated as a service facility which provides memory cycles to queueing customers, the processor modules. The arrival process (processors requesting bus cycles) can be approximated by a Poisson process[1] with an arrival rate of Nr, where N is the number of processors and r is the rate at which a single processor generates bus cycles. The service time, s of the bus is its cycle time and is assumed to be constant (This is reasonable assuming memory is all approximately the same speed). These processes characterize an M/D/1[2] queue[Kleinrock75,Coffman73]. Given that the bus is not saturated ($Nrs < 1$), the average number of customers in such a system (those waiting and currently being served) is given by

$$Q = Nrs + \frac{(Nrs)^2}{2(1-Nrs)}. \tag{6.1}$$

The average length of a bus cycle as seen by a processor is

$$T = s + \frac{Nrs^2}{2(1-Nrs)}. \tag{6.2}$$

The graph below shows the normalized average bus cycle time seen by each processor as a function of the bus utilization, Nrs. The curve increases slowly when the bus utilization is low,

---

[1]Given the details of the cache architecture and the sequence of memory accesses, the pattern of cache misses can be completely determined. However, from the point of view of the bus, cache misses occur at random intervals. Furthermore, the occurrence of misses is a memoryless process; that is, a miss is equally likely to occur at any point in time.

[2]The three characters refer to the arrival process, the service process, and the number of servers respectively. The M signifies an arrival process with a Poisson distribution. The D signifies a constant (Deterministic) service time.

but extremely rapidly as the bus becomes saturated. In this region, from *Nrs* =.8 and up, the model begins to break down. As the bus utilization approaches 100%, the model predicts that the length of the queue and therefore the average waiting time become infinite. This is because the model does not take into account the fact that processors waiting for or using the bus do not generate additional requests. This effect is greatest when the average number of processors in the queueing system is close to the total number of processors. The model is therefore inappropriate for analyzing systems close to saturation (as is apparent from the graph above). The model also predicts a small amount of contention in the single processor case due to the same effect. For unsaturated systems, the effect is small and results in the model predicting a slightly longer average bus cycle than would actually be observed.

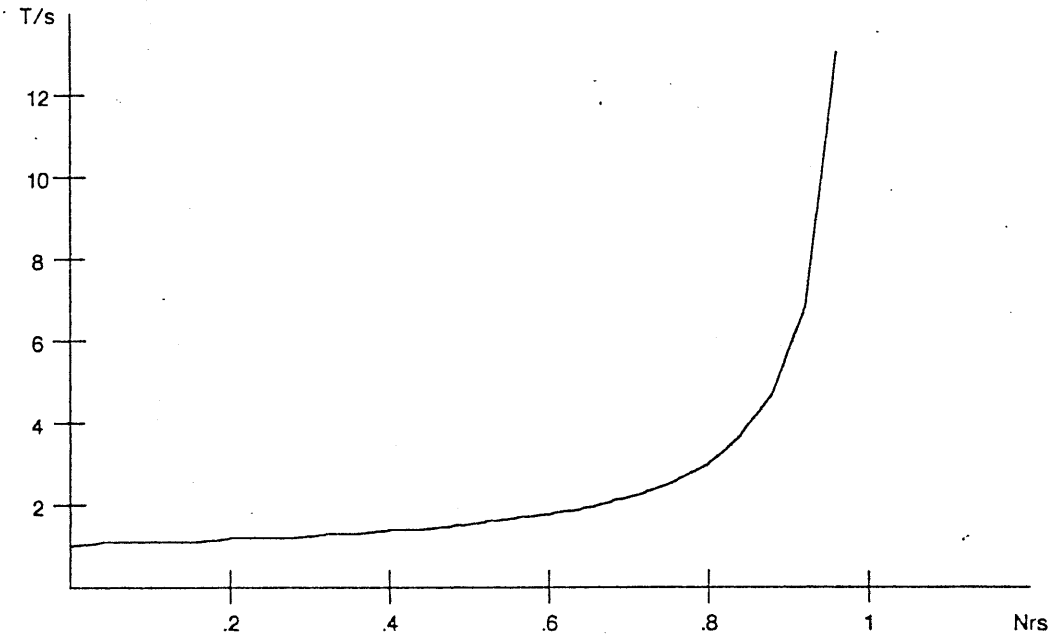


Figure 6.6. *Average bus cycle time vs. Percent saturation*

When the bus is saturated, processors will accumulate in the queue, waiting for cycles, until the average access rate of the remaining processors is less than the service rate (bus bandwidth). In addition to the queueing cause by variations in inter-request times, there is a fixed overhead due to these waiting processors. In the limit, $Nrs \gg 1$, the average bus cycle time seen by any processor will approach *Ns* if queued requests are serviced in FIFO order.

Thus the true curve describing effective bus cycle length will not exhibit the rapid increase in the vicinity of $Nrs = 1$ seen in the curve above.

The model can be applied to Nu computer system on which TRIX currently runs. The bus utilization for each processor is about 38%. Using this value, the model predicts a throughput for each processor which is 63% of that of the same processor running without bus contention. The performance degradation predicted by the model is greater than that observed. This is unsurprising since the total bus utilization is 76%, approaching the limits of the models applicability.

Caching decreases the effect of bus contention on processor throughput in two ways. First, by reducing the rate of bus accesses it reduces the amount of actual contention. Second, it somewhat insulates the processors from the fact that the effective bus cycle length has increased since only a small fraction of memory accesses actually require bus cycles. These effects can be seen in the table below. It gives the predicted bus cycle length (normalized) and the processor degradation factor for various configurations of a system in which each processor contributes a bus utilization of 10%.

N	Nrs	Average queue length	Average cycle length	Effective processor throughput
2	0.20	0.23	1.12	98%
3	0.30	0.36	1.21	98%
4	0.40	0.53	1.33	97%
5	0.50	0.75	1.5	95%
6	0.60	1.05	1.75	93%
7	0.70	1.52	2.1	89%

Because of the dual effect of caching on bus contention, the amount that processor performance is reduced is actually very small. Even in a heavily loaded system, bus contention only reduces throughput by 11%.

6.2. Kernel Contention.

Another situation where processors compete for access to shared resources is found in the kernel. Since the kernel data base is interlocked, only one processor can use it at a time. If several processors simultaneously desire access to the locked structures, all but one will spend time waiting in the lock routines. This waiting time decreases the effective speed of the processor in much the same way as bus contention, though at a higher level.

The degree to which this kernel contention affects performance depends on the fraction of time each running thread (and therefore each processor) spends in the kernel. This fraction is, in turn, dependent on the program executed. The table below shows the results of profiling some typical programs. Computation intensive programs such as *PUZZLE* spend very little (less than 1%) time in the kernel. The C compiler, which uses a fair amount of I/O, spends about 10% of its time in the kernel. Small utility programs such as the directory list program, ls, can spend up to 60% of their time in the kernel. This large percentage of kernel time comes from the fact that these programs are very short and do very little computation. In the case of ls, a significant part of the time spent in the kernel is due to the overhead of reading in and executing the command file.

Program	User	System	Kernel
puzzle	98%	0%	1%
c68	90%	1%	9%
ls	18%	23%	58%

The degree to which waiting for access to kernel structures decreases the effective performance of processors can be estimated using the queueing model developed for bus contention. The model is somewhat less apt in this case since the service times for kernel accesses (average amount of time spent in a locked section) is not constant. It can still provide, however, an idea of the magnitude of this effect. The table below shows the amount of

performance degradation expected from kernel contention assuming a mix of tasks which spend 15% of their time in locked sections.

N	Effective throughput
2	97%
3	94%
4	90%

6.2.1. Optimization

There are several ways in which the system can be optimized to reduce the amount of time programs spend in the kernel, which in turn will reduce the performance effects of lock contention. The most straightforward of these is to optimize the kernel routines so that they run as efficiently as possible. This is worthwhile even for a single processor TRIX and must be done when TRIX is converted to a production system. Aside from this, there are some structural changes to the kernel which should reduce the frequency of locking.

Probably the most frequent requests for the kernel data base are due to processors looping in the scheduler. Idle processors (those not currently running any thread) loop, searching through the lists of scheduled threads. This has to be done while the data base is locked, otherwise threads and domains could be inserted or deleted from these lists while they were being read causing confusion. This can be a large overhead on the kernel interlocking, delaying processors which need to lock the kernel to do something useful. An optimization can be made to the scheduling loop code which will greatly reduce this overhead. After one pass through the list of threads, if no suitable thread is found, idle processors should sit in a loop (having unlocked the kernel) testing a NEW_WORK flag. Occasionally the processor would time out of the loop, and repeat the search (hoping the situation has changed). The NEW_WORK flag would be cleared after an unproductive search for a thread to run and set when a new

thread is spawned or an existing thread awakened, immediately breaking processors out of the idle loop to run newly available thread (this keeps the processors from waiting for the time out when there is work to be done).

Another method of reducing frequency of kernel calls is to decrease the number made by commonly used subsystems. One way this can be achieved is by designing the I/O structure with a minimum number of "levels of abstraction". Each level typically makes multiple requests to the next lower level, and each request requires kernel intervention. Careful design of these utilities can reduce this overhead. Another approach is allow very low-level, frequently used, trusted systems to bypass the normal request/reply mechanism and use direct procedure calls instead (system utilities can probably be trusted more than typical user code).

Decreasing the granularity of kernel interlocking is not likely to decrease contention since each processor must still spend the same amount of time in locked sections. Although the time spent in each locked section would decrease, the number of times locked sections would be requested would increase. Their product, the utilization of the locked sections, would remain about the same as would the average waiting time. In fact, overall performance is likely to decrease since the effect of locking overhead will have increased. Another possible approach is to use several separate interlocks in the kernel, each lock protecting a separate section of the data-base. For example the scheduling and unscheduling of threads can be interlocked separately in each domain. A processor executing a **wakeup**, for example, would not interfere with processors in other domains. However, most kernel calls must use several resources so the advantage of separate locks is unclear.

6.3. Utilization

The remaining factors which affect system performance are functions of the user's programming and working styles. They include the degree of contention between threads for access to I/O, and the manner in which threads are used as programming constructs. These affect perceived performance by determining the extent to which multiple processors are utilized.

One issue is the degree to which user domains support multiple running threads. A given domain, say a file system or network server, can be kept always at high priority, so that only one thread at a time can run in it. This assumption simplifies writing the code of the domain since

data objects and critical code sections do not have to be explicitly interlocked. However, this means threads may occasionally have to wait to start executing in the domain because another thread are already running in it. This in itself is not a problem since the processor that was running the thread immediately searches for a new thread to run. However, the overhead of switching contexts in order to run a new thread will decrease that processor's productivity. In addition, it is possible that the processor will remain idle because there are no other runnable threads. It is worthwhile, especially in commonly used subsystems (file systems, terminal interfaces, etc.), to code the domains so that they can support more than one thread at a time.

6.3.1. Device Contention

At the lowest level of I/O, transactions are naturally seriatized. Only one physical disk block can be accessed at a time; only one character can be sent to a TTY at a time. As threads compete for access to IO devices, bottlenecks will occur since these tasks are processor independent. If a two processor system is supporting twice the number of users as a single processor system, the amount of I/O and disk transactions can be expected to double (more or less). This can lead to bottlenecks unless the I/O resources have doubled. Often increasing the size of the various system caches (such as the disk block cache) can help.

6.4. Results

A series of experiments with multiple thread tasks were run to measure the performance of TRIX. One conclusion from these was that TRIX running on the current dual processor Nu hardware is dominated by bus contention and disk latency. One experiment ran two copies of the *PUZZLE* program simultaneously. This program was chosen because it is small and computation intensive. The effects of disk latency are therefore at a minimum. This task was run on three configurations, a single processor system, a two processor system with one processor immobilized in a tight loop (though still requesting bus accesses), and a two processor system with both processors active. The execution times are shown in the table.

Experiment	Time (s.)
1 proc.	59
2 proc. 1 active	80
2 proc., 2 active	41

The difference between the first and second entries is due to bus contention. The difference between the second and third entries shows the effect of utilizing a second processor. Both processors are completely utilized in this task, and the speed at which the task is completed is actually doubled. The results of experiments with I/O bound programs are of course less impressive. Utilities such as ls are dominated by disk and system overhead and run serially on the dual processor systems. The majority of tasks, mixs of compilation, compute bound programs, string parsing, run faster on the dual processor system, but not nearly twice as fast. One important factor affecting system performance on the current hardware is that many programs are dominated by disk accesses, which must be performed serially.

6.5. Summary

The maximum throughput of each processor in a multiprocessor system is limited by two factors: bus contention and kernel lock contention. The effect of each of these can be analyzed separately and modeled to predict the maximum performance of TRIX on different hardware. The effect of bus contention depends on the number of processors in the system, the average access rate of each processor, and the cycle time of the bus. The effect of kernel contention is determined primarily by the fraction of time a given job mix (when run serially) must spend in the locked sections. For a system with two cached processors running tasks spending 15%, an atypically large fraction, of their time in locked sections, the performance of each processor is reduced by 3% by kernel contention and 2% by bus contention. The maximum throughput of the system would be 190% of that of a single processor system. Adding a third processor should bring the throughput to 2.7 times that of a single processor system.

7. Summary

A multiprocessor TRIX system has been built which meets the stated goals. The system treats all processors symmetrically, assigning any task to any free processor. The kernel supports execution by multiple processors simultaneously. Operations on critical sections of the kernel data base are locked to insure the safety of the system. The operation of the system is free from deadlocks between processors.

The structure and semantics of TRIX allow it to be adapted to a multiple processor implementation fairly elegantly. Kernel calls provide only the minimal functionality needed to implement the virtual machine model. Most "system" features are supported at user level. This reduces the problems of interlocking the kernel. There are only a small number of kernel entries which need to be monitored, and processors pass though the kernel quickly. This means that the latency for processor access to the kernel is small, so mutual exclusion on the critical sections in the kernel does not greatly degrade performance. The interaction of the TRIX kernel with the input-output systems is very small. Only at the lowest level of device handling, the interface of the user thread and the interrupt level processing, is the kernel involved at all. Even at this level, the amount of interlocking needed is small.

In contrast to TRIX, the UNIX I/O systems are all part of the kernel. This makes the UNIX kernel much larger and more complex than the TRIX kernel, with much more interaction between the various subsystems, such as I/O and processes control. Because of this, the UNIX kernel is difficult to properly interlock.

The performance of multiprocessor TRIX running on the current incarnation of the Nu computer hardware is dominated by bus contention and disk latency. Bus contention on this hardware limits the maximum throughput of the system to about 140% of that of a single processor system. Discounting the effect of bus contention, the performance of the system when running compute bound tasks approaches twice that of a single processor. The increase for I/O bound tasks is naturally less.

The major factors affecting the maximum throughput of the system are bus contention and contention for locked sections of the kernel. The effect of these has been modeled for a system in which several processors with caches share a common memory bus. The throughput predicted by the model depends on the parameters of the hardware and the characteristics of the job mix. A typical choice of parameters yields predictions for relative throughput of 190%

and 270% for a two and three processor system, respectively.

At some point, it becomes an economic issue whether or not to use additional processor in a system. There is a point of view which holds that a multiprocessor system is not worthwhile unless all of the processing elements are fully utilized. This position becomes increasing obsolete as processing elements become cheaper. A more appropriate perspective is concerned with whether the set of tasks running on a system are completed as quickly as possible. Additional processors will increase the effective computing power of a system, but this increase is subject to diminishing returns. The increment added by each processor diminishes as contention effects begin to dominate the system. There is, however, an optimum point on the cost-performance curve

REFERENCES

[Coffman73] E. Coffman, and P. Denning, *Operating Systems Theory*, Prentice-Hall Inc., Englewood Cliffs, New Jersey,1973.

[Goble81] G. Goble, and M.Marsh, *A Dual Processor VAX 11/780*, Purdue University, 1981.

[Kleinrock75] L. Kleinrock, *Queueing Systems*, John Wiley & Sons,Inc., New York, 1975.

[Ritchie78] D. Ritchie, and B. Kernighan, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Thompson78] K. Thompson. "UNIX Implementation", *Bell System Technical Journal*, Bell Laboratories, Murrey Hill, New Jersey, 1978.

[Sieber83] J. Sieber, *TRIX: A Communication Oriented Operating System*, M.S. Thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Ward80a] S. Ward, "TRIX: A Network-oriented Operating System", *Proceedings of COMPCON '80*, San Francisco, 1980.

[Ward80b] S. Ward, "An Approach to Personal Computing", *Proceedings of COMPCON '80*, San Francisco, 1980.

[Wulf73] W. Wulf, et.al., *HYDRA: The Kernel of a Multiprocessor Operating System*, Carnegie-Mellon University, Department of Computer Science, 1973.

[Wulf81] W. Wulf, R. Levin, and S. Harbison, *HYDRA/C.mmp An Experimental Computer System*, McGraw-Hill Inc., New York, 1981.