TRIX: A Communications Oriented Operating System

by

Jonathan D. Sieber

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

TRIX is a general-purpose, multi-user, interactive operating system. It is designed to support investigations into communications systems and their semantics in both a research environment and a production situation. The communications primitives available to the user have been used to implement the following modules outside of the kernel:

- a file system
- terminal support
- an integrated network server
- a window system

TRIX is implemented as a small kernel that has a high degree of portability. This thesis discusses the evolution and current structure of the kernel and describes some of the ways in which it is being used.

Name and Title of Thesis Supervisor:

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

Key Words and Phrases:

operating systems
communications

ACKNOWLEDGMENTS

I would like to thank my family, my friends, and M.I.T (particularly its representative, Professor Stephen Ward). Together, they gave me encouragement and a place to work.

Any success this project has achieved is due in large measure to the numerous people who have taken part in the discussions that lead to the current implementation. Among those I would like to thank are: Steve Ward, Bert Halstead, Chris Terman, Dave Goddeau, Jack Test, and all of the members of RTS.

The existence of this document is due to the help of Earl Cohen and Oded Feingold. Both encouraged me to finish and had the fortitude to read and comment upon a number of drafts.

This work is dedicated to Ken Thompson. There is no way I can express my appreciation for what he has taught me over the years.

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

Early time-sharing systems were primarily motivated by a desire to partition a single physical machine into many virtual machines. Later work was driven by an interest in allowing sharing among the users of the virtual machines while controlling their access to "private" information; the resulting systems emphasized protection and security.

The current trend in operating system development has been towards replacing virtual processors of a time-sharing system with multiple real processors. These multi-processor systems initially used tight coupling of shared memory systems, but are evolving towards the loose coupling of a network. Though this move to geographically isolated processors makes protection more tractable, it also makes the cooperation that resulted from timesharing difficult to achieve.

These changes in software systems reflect the evolution of hardware in the last two decades. Timeshared operating systems were developed in response to the high cost of mainframes: the individual users of a mainframe each wanted his own machine but the cost was prohibitive. Timesharing a mixture of (high priority) interactive jobs and (lower priority) computationally bound jobs increased total throughput with little impact on each user's access to the machine. Device channels and devices using Direct Memory Access (DMA) or interrupts were developed gave each user an independent "processor" during I/O. Since these special purpose processors cost less than a mainframe, this was an inexpensive way to obtain substantial amounts of additional processing power. However, the current trend towards computationally intensive I/O (e.g. high-density bit-map displays) eliminates much of the asynchrony of I/O to the terminal and reduces the processing power of a system to that of its CPU elements. Fortunately, the cost of processing elements has decreased to the point that the use of multiple processors is now economically feasible.

The TRIX Operating System is designed to take advantage of recent hardware developments by supporting communications in a distributed environment. It gives user programs the same interprocess communications mechanism that it uses internally to implement various system functions. For example, the object naming semantics of a file system is not included in the underlying structure of TRIX. The user's ability to implement such structures (as opposed to making them system primitives) demonstrates the flexibility of the available mechanisms.

The hardware configurations on which TRIX can run include processes running on both real and virtual parallel processors, with varying degrees of coherently shared memory. Many existing operating systems rely upon the fact that it is easy to achieve coherent memory sharing among multiple virtual processors running on a single real processor. But, when the bandwidth limitations of a shared memory bus require more loosely coupled systems, a different communications structure is required. TRIX has been designed to unify tightly and loosely coupled communications.

The goal of the TRIX design effort [WARD1, WARD2] was to develop an operating system that is useful in investigating the organization of multi-processor systems and their supporting communications mechanisms. Much of TRIX's power results from a mechanism that does not force a distinction between local and remote communications. Its implementation is based upon an abstract machine that minimally defines the semantics of information sharing thereby permitting a variety of implementation strategies depending on the situation. This abstract machine lets TRIX use a single communications primitive for all interprocess communications, without degrading the performance of simple tasks that can be run on conventional operating systems. This idea, that the virtual machine (as specified by the kernel) need not be as rigidly defined as the underlying hardware, is a departure from most common operating systems.

The original TRIX communications facility (TRIX-0) was a nonblocking, message passing mechanism. As work progressed this implementation was found to have deficiencies often overlooked in the literature. These deficiencies led to the development of a revised system, based on a remote procedure call mechanism. Both the original implementation and the revised systems will be discussed with emphasis on the tradeoffs that were considered during their development.

2. COMMUNICATIONS SEMANTICS

This section outlines a framework for discussing communications structures in isolation from the surrounding operating systems. The resulting model allows us to compare structures along with respect to several semi-independent measures:

- the types of relationships supported between communicating processes,
- the semantics of the communications system (as defined by a list of "features" that it supports),
- the cost of communicating,
- assertions that can be made about the state of the communications system at any point in time.

2.1. Relationships Between Communicators

Depending on the situation, the relationship between communicating processes can range from master/slave (requester/handler) to cooperation among equals, and from mutual trust to mutual suspicion. Because each is appropriate in some situations, a communications mechanism should not preclude any of them. A related issue is whether the communicators can trust that everyone is adhering to the proper protocols, and whether this adherence is enforced or is simply a set of conventions. It is important that a malicious process be unable to impact the operation of other independent processes. In a distributed environment with no central authority such immunity can be crucial since there may be few situations in which the communicators trust each other enough to obviate enforced conventions.

The passing of control among processes is also an issue. Possibilities include: message based send/receive, procedure based call/return, and a data driven approach in which the data itself passes control via reads and writes.

Finally, the methods by which processes synchronize their actions is also considered a part of the communications system (though it has often been discussed independently in the literature).

2.2. Semantics

Most communications systems are developed from the standpoint of desired functionality. For example, if a desired application (such as a network server) can not be implemented with the existing communications mechanisms, during its implementation the required functionality is specified and the minimum sufficient mechanisms added. Some useful semantic features of the communications mechanism include the following.

2.2.1. Multiplexing Multiple Requesters Into a Single Handler

An ingredient missing in most communications schemes is a mechanism that lets a single centralized service to simultaneously do work for more than one user. Such simultaneous service is difficult to implement because the simplest communication schemes involve some form of blocking I/O. In such an environment, it is difficult for one process to interact with many other processes without blocking on any single one.

2.2.2. Asynchronous Communications

When using a sophisticated communications mechanism we can increase throughput by partitioning an applications program into multiple activities that can run in parallel on the multiple processors of a distributed system. One situation when an otherwise serial program often allows concurrent activity is during I/O. To take advantage of this concurrency a user should continue computing as soon as possible after initiating communications with an I/O handler. This is most simply done by making the I/O mechanism invisibly asynchronous. For example, doing disk I/O through a read-ahead/write-behind cache lets the user continue executing before the actual I/O completes.

A problem with hiding the asynchrony from the user in this way is that the status returned by such an I/O operation does not reflect completion of the operation, but rather its initiation. A consequence of errors not being reflected back to the user is that, when writing through a block cache it is difficult to guarantee that the information has been stored reliably to disk. One solution is to make I/O explicitly asynchronous, but this is more complicated for the user since it requires that he explicitly synchronize his I/O activities. Note that the fact that two "independent" processes are communicating does not necessarily imply concurrency; it may be that one process is always waiting for the other, and that a single point of activity is passed between them.

2.2.3. Flow Control and Artificial Serialization

Two communicating processes are not always matched in speed. In systems constrained to avoid communications loops, a common solution to the flow control problem involves blocking the faster process when it gets too far ahead of the other. This blocking bounds the amount of buffer space (or some other communications resource) needed to complete the computation. A cleaner solution is to limit the faster one in a semantically transparent way; in effect, scheduling becomes more and more biased against the faster process until a balance is achieved. This solution is not applicable to most distributed systems because it requires a centralized scheduler to coordinate the biasing. It is generally replaced by an approach that puts hard limits on the amount of traffic over each link of the communications system. But this approach, because it is no longer semantically transparent, can lead to deadlocks. There are two conflicting goals here, namely the need to implement flow control mechanisms limit the use of resources, and the need to avoid artificial serialization of running processes. Avoiding artificial serialization (situations in which one process is waiting unnecessarily for an otherwise independent process) is important in deadlock avoidance since these unnecessary data dependencies can not be anticipated easily.

2.2.4. Controlling and Restricting Communications

In systems that support communications among antagonistic processes it is desirable for a module to be able to restrict what other modules can communicate with it. This restriction can be based upon either the *identity* of the communicator or some verifiable *capability* to communicate.

2.2.5. Use of Tightly and Loosely Coupled Processors

Underlying the difficulty of implementing many of the features outlined here is the desire to allow the user to deal transparently with all types of distributed situations. For a single communications mechanism to be able to deal with both tightly and loosely coupled configurations, it must not depend upon the small latency of tightly coupled communications. Thus, the semantics of communications should not be based on shared memory, since it is difficult to map such a mechanism efficiently onto loosely coupled processors. However, the use of shared memory should not be precluded when it is available, since it can often significantly increase performance.

2.2.6. Transparent Interposition and Encapsulation

A useful tool in dealing with transparent communications is to allow the insertion (interposition) of processing in a communications path. This interposition makes it possible to isolate a process from the rest of the system, thereby "encapsulating" it. Semantically, this encapsulation lets a module's functional environment be modified at will. Encapsulation may be used to implement per-user policies for limiting resource use (which in turn can be used to support service requests from a network), or to monitor and control a module's activity with respect to the outside world. It can replace the concept of *identity* on which non-capability systems base their security, and has been valuable in debugging modules that have complex interactions with other parts of the system.

2.2.7. Universality of Communications

When a system cannot support encapsulation, that shortcoming usually reflects a lack of universality in the underlying mechanism. Most existing communications mechanisms do not let the user pass certain types of system-managed information. The result is that communications with some system-supported objects must use a mechanism unavailable to the user, and it may be extremely difficult for the user to build some fundamental constructs (such as naming systems).

2.2.8. Expressing Complicated Relationships Between Objects

Though communications typically consists of unary operations on some object (read, write, etc.), it is often desirable to express more complex relationships. In the simplest of these relationships the user must test whether two or more independent communications paths are talking to the same object. Other higher level operations (such as locking a group of communications channels as an atomic operation) can be built as special cases, but it is difficult to integrate them with the mechanisms that cope with certain types of failures (e.g. software bugs that cause failures of an application module).

2.3. Costs of Communication

Communications should allow processes of varying degrees of independence to transmit information among themselves (so that they can collectively perform a task), both to synchronize control and to move data. Measures of the cost of such mechanism must be expressed in relation to the total bandwidth available to the communicators.

The first measure of the cost of using a specific mechanism is a communication containing no data. Such messages are used to transfer control information, and in an ideal situation would be comparable in cost to a subroutine call with no arguments. In real systems, the existence of independent tasks running in independent address spaces significantly reduces the control bandwidth. The actual cost is closely related to the number of context switches that the mechanism requires and how well the required context switch semantics are supported by the hardware. It is also related to the scheduling overhead that it semantically requires; a mechanism that requires completely synchronous behavior will often require significantly more scheduling.

The effective control bandwidth is also affected by the power of the underlying primitives themselves. If many kernel calls are required for common operations, it is grossly inefficient to supply the user with a set of primitive kernel functions from which any desired function can be derived.

The most common use of a communications mechanism is for moving data. Thus, the data bandwidth between communicating processes is a significant and quantifiable cost of the mechanism. Anything that forces data to be handled unnecessarily should be avoided. Two specific pitfalls are: requiring that the data be copied from where it is to some other place for easy communications, and forcing layers of a protocol to touch the data even though they have no direct interest in it. We again distinguish between attempts to avoid unnecessary copying of data in contrast to semantically precluding such copying.

2.4. Assertions About the State of the System

The rich communications topology that can result from the use of a general communications mechanism raises organizational problems that are commonly avoided by the hierarchical organization of most operating systems. For example, in the case of resource allocation, reference counts are not sufficient to keep track of resource use, and a garbage collected object space of some type is required.

The best hope for reasonable resource management lies in imposing constraints on the use of communications mechanisms. But there is a tradeoff between overly constraining the mechanisms (thereby diminishing functionality) and leaving them unmanageable (thereby dimishing reliability).

3. CASE STUDIES OF COMMUNICATIONS SYSTEMS

Networks and timesharing systems have existed for well over a decade. As a result, one might expect widely used software systems to have evolved mechanisms that support robust communications. Although many systems have moved in this direction, each has placed limitations on the use of its underlying mechanisms that kept it from fulfilling the needs of many users. These limitiations are a consequence of the (relatively) low speeds of available networks and the traditional use of timesharing systems as multi-user batch systems, both of which have deemphasized the need for generalized communications.

3.1. UNIX Based Communications Mechanisms

The UNIX† Operating System [RICHIE] has become a standard for operating system elegance and functionality. However, it contains a number of historical deficiencies which can be traced to its development at a time when communication was not a driving force. While most of these deficiencies (notably the minimal support for terminal handling and job control) have been remedied over time, work to extend the communications mechanisms has had only partial success. In many ways this is simply a reflection of how well engineered UNIX is: people attemting to add a more generalized communications mechanism have found it difficult to extend substantially UNIX's basic machine model. More important is the fact that experimenting with solutions to these problems requires changing the UNIX kernel, an environment in which it is often difficult to isolate the functionality that needs to be modified.

3.1.1. Version 7 UNIX

Communications under UNIX uses three structures:

- *signals* allow processes to notify each other of unanticipated events,
- a *pipe* is a synchronized data transport mechanism between processes, and
- the *file system*'s directory structure names files and controls their access.

A signal is a user-level interrupt mechanism that permitting a process to designate an interrupt handler that will be called after a signaling event. Because signals were designed to allow programs to be notified of user generated interrupts or exceptions (which were expected to occur infrequently), a

† UNIX is a Trademark of Bell Laboratories.

signal that recurs before the last one is completely handled may be lost. This problem is aggravated by allowing one process to signal another using the *kill*() system call. The absence of flow control (notifying the sender that its signal has been received), makes it difficult to guarantee that signals are not sent faster than they can be handled.

Another problem is the limited control over the use of the *kill*() mechanism; only processes owned by a single user can use signals to communicate. Integrating a useful control mechanism with signals requires adding a new mechanism for distributing the capability to send them.

A pipe is a data transport mechanism: a FIFO buffer with two ends, one readable and the other writable. Communication with pipes is completely data driven; control is passed among the communicating processes solely in response to the availability of data. As a result, the only functions pipes support are reading and writing a data buffer.

Like most data driven mechanisms, pipes are simple to use. The implicit flow control and the limited amount of asynchrony that results from the pipelining are sufficient for many simple uses, but break down in more interesting situations. The most glaring deficiencies of pipes:

- There is no way for a single server to accept input from more than one user (because of the blocking nature of UNIX I/O).
- There is no way to pass control information (to support randomly accessed files or retrieve the status of a remote file), a typical problem with data driven mechanisms.

Even the use of the FIFO buffer is a mixed blessing. Though it effectively pipelines communications and reduces the latency between processes to allow greater concurrency, copying data to and from the buffer increases the cost of the mechanism. This copying is semantically necessary since there is no way for the writer to be informed when a delayed write completes (signifying that the writer's data buffer is available for reuse).

The file system's hierarchical directory structure supports a single name space for files. Users can access shared files to communicate. This is the only name space in UNIX that supports controlled sharing of information. Unfortunately, other communications mechanisms are not integrated into the name space. Only directories and files can be named and accessed through the file system; there is no way to name and distribute access to a pipe. Other communications functions can be built using file system mechanisms and agreed-upon conventions, but because these conventions are not enforced by the system, they are unacceptable for general use. (For example, the file creation primitive can be used to implement lock files and hence synchronization, but users must voluntarily test the lock.)

3.1.2. Rand Ports

Rand *ports* [BALZER] integrate an extension of UNIX pipes into the file system's naming structure. The appeal of integrating pipes into the normal file naming scheme is that it adds little new user level mechanism. Unfortunately, it is insufficient in even simple cases. For example, the open on a port simply returns a file descriptor that lets opener communicate with the server. Since the only access verification is at the point of contacting the name in the file system (rather than after the handling process is allowed to validate the existence of the requested file on the remote machine), there is no way for an open request to return a "remote file not found" error reply.

In addition, every port needs an active process waiting on it even when it is not in active use. Though this approach might be adequate to implement a limited number of special handlers supporting (well known) services, if it were used heavily it would require an unbounded number of idle processes waiting to be contacted.†

Ports also support a *capacity*() call that lets the user determine whether an I/O request on a port will block. An additional call *await*() lets the port's reader block while waiting for any of his ports to have data available. These extensions let a single handler read from more than one port, thereby supporting simple multiplexing.

3.1.3. MPX I/O System

The MPX I/O System was introduced into UNIX to allow the user to build communications multiplexors. It takes the approach of separating the communications medium from the multiplexing mechanism. A *channel* is the actual communications medium; it supports a superset of the semantics of pipes, files, and the terminal driver. A *group* does the multiplexing; it structures the data on both input and output streams to identify with which stream the data is associated. By giving the user warning before blocking, it also supports I/O on more than one stream.

Like a RAND port, a channel may be created with a name in the file system. When the name is accessed, the handling process is notified and may either accept or deny the access. This option on the part of the handling process is extremely useful in building a network-based remote file access system since it lets the remote file's existence be verified before the open completes.

----

†At one point, I developed an *active inode* mechanism for UNIX [SIEBER] which allows specially marked files to initiate active processing when they are accessed: the spawned process was connected to the opening process by a pipe.

3.1.4. Conclusions

Most of the work done to extend UNIX has centered around integrating a communications channel into the UNIX file name space and allowing a single server to accept service requests from more than one channel. Little work has been done to address the limitations resulting from use of a data driven communications channel. Although these extensions are acceptable for simple mechanisms (e.g. remote login) they still do not support random access of remote files. As a result, there is always an asymmetry between local and remote files.

Another problem is that UNIX's concept of what constitutes an asynchronous activity (a process) is too expensive to use in a number of situations. For example, using multiple processes to implement a full duplex terminal link program (*telnet*) involves a full address space context switch on each typed (or echoed) character; at high data rates this is prohibitive. In simple cases the polling mechanism that is added to blocking I/O can simulate real asynchrony, but it is insufficient in the more complex cases that arise when communications takes a more central role in system operation.

3.2. Message Passing Systems

The term "message passing" is used in the popular literature to describe a variety of different communications schemes. Usually, it denotes a general implementation. In some cases it refers to the modular structure that results from using such schemes. We will describe a specific style of programming and use "message passing" to refer to an underlying mechanism supporting it. It is important in this discussion to understand that the distinction between message passing and other styles of communications is mainly an issue of emphasis rather than strict functionality.

In a message passing system [LAUER], the universe is composed of independent processes. These processes share no portion of their address space, and are long lived compared to the time for a single communication. The only communications mechanism between these processes is the act of sending and receiving messages. The result is a master/slave relationship between two processes in which one is generating a stream of messages for processing by the other.

A message is a small block of memory in which the sender places all the information the receiver will need to perform the requested service. The receiver's action is to dispatch on the function to invoke the appropriate piece of code.

Message passing systems generally include a capability mechanism to control which processes can communicate with which others. A message passing capability is a protected object empowering its owner to send messages to another process. To support a communications topology that can change over time, messages are allowed to contain some number of capabilities.

The act of communicating involves putting a message in the receiving process's pending message queue; the fundamental operations are *send*() and *receive*(). Send() adds a new message to the pending queue of another process. Receive() removes a pending message from the calling process's queue to allow it to be processed. Often, receive() lets the user wait for some class of messages based on their source (*source specificity*).

A *reply*() operation may be included to let the receiver return status information. It can be modeled as the implicit inclusion of a use-once capability letting the receiver return a message. Source specificity is most commonly used to let a process wait for a reply without accepting any new messages.

Part of the appeal of message based systems is that they map well onto a loosely coupled system. There are two reasons for this: First, the message itself is a compact encapsulation of all information necessary to satisfy the request for service. Second, all synchronization is implicit in the queue of outstanding messages waiting to be serviced. Both the finite message and a queue of outstanding service requests can be implemented without any tight coupling.

Another advantage of message passing is that it is inherently a split transaction protocol (a protocol wherein there may be multiple outstanding transactions). Since the sender of a message can continue running without waiting for a reply, the outstanding message is viewed as an asynchronous task that has been spawned as part of the send. This asynchrony important when a communications medium has widely varying latencies, as does network communications. The ramifications of modeling a message as an asynchronous task will be discussed in greater detail later.

3.2.1. TRIX-0

TRIX-0 is a message-based implementation of the TRIX kernel that has been used to experiment with message passing as the underlying structure in a general communications system. It is a nonblocking message passing system in which messages are passed across interprocess links known as *ports.*

TRIX-0 supports four basic communications primitives: send(), receive(), forward(), and reply(). The forward() operation allows a process receiving a message to dispatch it to another process and have the reply go directly to the original sender. It is an optimization to support the message passing equivalent of *tail recursion* [STEELE].

Each message contains: an *operation-code* (opcode), some uninterpreted data (passed by value), an arbitrary size buffer of data (passed by reference), and an (optional) pointer to a port that the receiver can use. The opcode specifies what request is being made of the handling process and by convention is chosen from a single name space of operations. The data passed by reference (*data buffer*) is accessed using a pair of system calls, *mread*() and *mwrite*().

When a process starts it has a single port, known as the *environment* port, to which it can send messages. This is the process's initial contact with the rest of the world. It accepts a number of messages that let the process access I/O devices and the root of a file system name space.

Sending a message is a nonblocking operation. In order to avoid the problem of a single process swamping another with messages a simple flow control scheme is included: only one outstanding message is allowed on each port. Thus, communications is not a simple split transaction mechanism and the user of a port is required to manage a queue of messages waiting to be sent. Hence the parallel processing that the message "tasks" represent is artificially serialized. This serialization represents an unwanted data dependency in the communications system, and without special care can result in a deadlock. To guarantee that the TRIX-0 file system (the most complicated of the handlers we implemented) could not deadlock, the protocol for looking up a file in a directory was made considerably more complicated.

Conceptually a directory is simply a process that associates a port with a name. It accepts *enter* messages (containing a port and a name) that create the associations, and *lookup* messages (containing a name) that return the associated port. Looking up a compound name should be recursive: lookup(dir,"n1/n2") --> lookup(lookup(dir,"n1"),"n2")). Organized in this way, the first (outermost) lookup request is made by the user and the subsequent (inner) requests are made by the file system. This organization leads to a problem since file system resources will be tied up while it waiting for replys from the recursive messages. (These resources may be significant since the file system is performing the stack management necessary to simulate the per-message tasking.) Because any port can be entered in the directory, these inner requests may be to a process other than the file system. In that case, the use of resources by the file system (and thus its reliability) can depend upon the reliability of an unknown handler. Hence it is impossible to bound the resources a file system process might need to fulfill all legal requests. In the worst case, it can run out and deadlock. This problem stems from the impossibility of associating the file system's resource needs with the process sending it a message.

An alternate implementation makes lookup iterative. In this case the entered port can only affect the initiating process's reliability. An unfortunate side effect is that parsing a multilevel name generates a surprisingly large amount of traffic.

Using a library of UNIX system call write-arounds (library routines that emulate the UNIX environment under TRIX), we were able to compile and run UNIX commands. Our benchmark of the communications mechanisms and the file system (which ran as a user process) was running the UNIX "ls -l" command (to list the contents of a directory and the state of each of the contained files). Its speed was extremely disappointing. Later comparisons with the same command, run under the version of UNIX (NUNIX) brought up on the same hardware, showed that it was about 1/20 the latter's speed.

Quite a bit of investigation was needed to explain this speed difference. Initially it was assumed to result from a combination of the relatively slow speed of the disk and the fact that the TRIX-0 file system had no cache for disk blocks as does UNIX. However, implementing such a cache resulted in only a 30% speedup.

Further investigation showed that the most significant mistake consisted of choosing the wrong set of primitive messages upon which to build file access and other high level protocols. In the name of aesthetics (and to allow complete encapsulation of processes) we chose an extremely simple set of low-level primitives, and few of the operations we desired mapped even indirectly onto them. This inefficiency was further aggravated by the unduly complicated protocol for opening files.

Under UNIX, N+2 system calls are needed to list a directory with N entries. (One to open the directory, one to read its contents, and one to get the status of each of the N entries.) Under TRIX-0 the number of messages sent was over 10 times that. Each message represented four system calls (send(), receive(), reply(), and receive()) not including the mread() and mwrite() calls used to access the data buffer. The file status write-around (fstat()) had the highest ratio of TRIX-0 to UNIX traffic because its speed was dominated by the time to set up and break down connections, an operation that we did not anticipate being in the inner loop.

Though this initial implementation was demoralizing, it did demonstrate that a message passing system can support adequate amounts of traffic, if the traffic is used effectively.

3.2.2. DEMOS

The DEMOS Operating System [BASKETT] is a message based communications system written for the CRAY-1. The interprocess communications operations DEMOS supports are: *send*(), *receive*(), *reply*(), and *move*(). Send() and reply() transmit a message over a link to another process; receive() waits for and returns the next pending message in a specified class. Four types of links (request, resource, reply, and general) each allow a slightly different class of communications. For example, a reply link can only be used once; its owner loses the ability to use it after sending one reply message. The different link types are used to restrict the communications so that two unrelated processes can only initiate communications through the *switchboard*. In this way, the operating system is guaranteed to have a hierarchical organization and deadlocking is avoided.

A link may contain a pointer to a data area in the sending process; move() lets the owner of such a link access that data. If the link containing the data area pointer is a reply link this pionter is semantically equivalent to the data buffer included by TRIX-0 in the message itself. This kind of pointer can be used to emulate shared memory regions between processes by placing it in a non-reply link. These shared regions are not supported directly because of limitations in the memory management hardware on the CRAY-1.

DEMOS's communications protocols are general enough to support many system functions. However, to alleviate problems resulting from artificial serialization and to make garbage collection tractable, it has limited abilities to support operating system extensibility. DEMOS's use of message passing shows that a message passing system can be used both for interprocess communications and to structure the operating system itself. These constraints on how the mechanisms are used reflect the realities of message passing; a pure message passing system is difficult to use in a general way. Also, questions as to how the mechanism interacts with virtual memory never arose in the context of the DEMOS implementation (a CRAY-1).

3.2.3. ACCENT

ACCENT [RASHID] uses message passing more aggressively than either TRIX-0 or DEMOS. It is philosophically very similar to TRIX-0 in that it tries to provide a virtual memory system with transparent network communications. The most significant difference is that ACCENT is committed to solving organizational problems that led us to abandon the use of message passing. Because of this commitment, its message passing mechanism is considerably more complicated than those of TRIX-0 and DEMOS. In particular it includes the following:

- A software interrupt mechanism which notifies a compute-bound process when an asynchronous message is received. This is necessary to let such processes respond quickly to newly arriving messages without periodic polling.
- A limit on the number of messages that can be outstanding on any port. Under TRIX-0 overrunning this limit would result in the sender getting an error; this necessitated the more complicated file system name search. ACCENT gives the sender three options as to how to deal with this limit: the process can be suspended until there is space, it can return immediately with an error, or it can return immediately and get an interrupt when space becomes available. These options suffice for a number of common situations including: a user process communicating with a server, a datagram style message that the sender does not care about, and a server process communicating with a user process.
- A timeout on the receipt of a message. This wakes up the receiver if no new messages have arrived after a certain interval.

These extensions are designed to solve many problems that stem from the use of messages to represent asynchronous tasks. Though they deal adequately with a number of common situations, when message passing is used as the basis of all communications special cases arise, requiring additional mechanisms.

3.2.4. Conclusions

While investigating the performance of TRIX-0, certain issues arose that led us to reconsider message passing as an underlying communications mechanism. Many of these problems resulted from viewing messages as independent tasks and the program organizations that support this view.

One organization that significantly simplifies stack management is for a receiving process to view each incoming message as an interrupt. (In the case of ACCENT, a message's arrival may in fact generate an interrupt.) This organization lets a single stack support all the asynchronous activity but requires that messages be handled in controlled ways: a message should be replied to (as are interrupts) before the next one is dealt with.

Programming methodologies which use an explicit return PC for a continuation after subroutine calls can eliminate the use of a stack. Although this type of continuation lets handlers exhibit more complicated patterns of asynchrony, one pays the price of being unable to use recursive stack-oriented languages. The generalized interrupt model is quite sufficient to support (for example) a real time process control system, with a single running task and asynchronous events triggering state changes.

In designing and implementing some TRIX-0 subsystems (file system, network server, and window system) these conventions were painful to live with, and we felt that forcing users to do so would limit their interest in using the system. We preferred to view each incoming message as an independent task spawned by the sender. To support this implementation, each receiving process ran a simple multitasking system (including stack management). We felt this additional complexity was justified, since the use of multiple outstanding messages is what lets the user specify concurrent activity, hence letting multiple processors work together to increase throughput.

In effect, the model we tried to approximate is one passing full *continuations* as embodied in the message passing of ACTORS [HEWITT] and the MU Calculus [HALSTEAD]. In this model the act of sending a message spawns a new task and its continuation is actually a complete process state. This process state is used as the new environment in which processing continues after the reply. A simple return PC was a simple step in this direction, unfortunately too limited for our needs. Implementing a task stack in addition to the PC was a closer approximation to the desired mechanism; it sufficed to support the C execution environment but required that the send() mechanism block the sending subtask until the reply arrived. Hence a message send no longer represented a new task, and another mechanism was required to provide the desired asynchrony. This blocking send mechanism obviates copying the sender's stack into a continuation stack (which is necessary if the sender continues running).

A deeper problem with message passing is that it obscures the structure of the underlying activity. As a result, it is difficult for the kernel to make reasonable decisions about scheduling and resource allocation. The best-understood of these problems is scheduling in message based systems. The following example, based on issues that came up during implementation of the TRIX-0 file system, outlines the problems associated with using message passing.

We first used the interrupt style of message passing. In that case sending a message was combined with a receive that waited for the reply, and the file system handled only one request at a time. This approach was undesirable since it limited the amount of overlapped I/O, greatly reducing the system's overall performance. This performance degradation was especially significant when communicating with devices with long or varying latencies (as might be found in a network virtual disk): in a file system it precluded having more than one outstanding request to a disk. Thus it was impossible to reorder disk accesses to optimize head movement. What should have been viewed as totally independent tasks turned out to be artificially ordered by the communications mechanism.

An additional drawback to the interrupt-driven approach was that it reduced the overall reliability of some modules. In the case of a single file system process doing file mapping for more than one disk, one disk's reliability affected the reliability of files on the other. (For example, if due to software or hardware failures the first disk did not respond properly to a message, files on the second disk were also inaccessible.) In most operating systems this problem is resolved by trusting disks at least as much as the file system. Thus it is safe to bound the reliability of the aggregate file system with that of the least reliable disk. In TRIX-0 this solution is unacceptable since it precludes some of the disks (for example a remote network disk) from being user-level objects. In general, an individual object's reliability should reflect only the reliability of objects on which it directly depends.

In our TRIX-0 implementation we used a more robust programming model (our user-level multi-tasking) and decoupled send() and receive() to allow new requests to be handled while a prior request was outstanding. This model sufficed for our immediate needs but it became clear that the coming shift to a virtual memory system would reintroduce the serialization problem. Because the kernel scheduler can not "see" the multiprocessing proceeding inside a single process, a "task" taking a page fault implicitly sends a message and waits for its reply before any other processing can continue. The kernel cannot schedule a new subtask - this must be done by the internal scheduler, and that is driven by the arrival of new requests or replies. This serialization becomes yet more problematic when one considers that received messages can contain a reference to another address space: Accessing that foreign address space may result in paging to or from a device that the receiver process has no reason to trust. Under conditions of artificial serialization, *all* data objects managed by the receiver may have to wait on this (possibly unreliable) paging device. Note that this problem does not depend upon how the message is implemented. Instead it reflects the kernel scheduler's inability to manage message parallelism properly. Though message-based systems are isomorphic to procedure-based systems with respect to functionality [LAUER], they differ in regard to amounts of parallel activity and where such activity is serialized.

If tasking were explicit, artificial serialization could be completely avoided. Only those tasks dependent on a particular paging device would get hung up in case of that device's failure. The TRIX-1 kernel is predicated upon the idea that parallel running tasks must be visible and controllable by the kernel.

3.3. Other Systems

3.3.1. TOPS20 Shared Memory

The classic example of a shared memory operating system is the MULTICS Timesharing System [ORGANICK]. However, it introduces a number of complications, such as dynamic linking, which complicate the issue of using shared memory as a communications medium. As a result we will concentrate on shared memory communications under TOPS-20 [].

Interprocess communications between unrelated processes under TOPS-20 is built around two mechanisms: a message based interprocess communications mechanism, and regions of shared memory (initiated by mapping the same file into two process's address spaces). If this shared memory were associated with a synchronization mechanism, the result would be a functionally complete communications system. Programs would not have to loop while waiting for a memory value to change, a particularly useful feature. The use of mapped files to set up shared memory regions integrates it into the naming and protection of the TOPS-20 file system. Unfortunately, no such synchronization mechanism exists and the message based system must be used to pass control information.

Though the use of shared memory regions permits a high data bandwidth, the result of using both the message based communication mechanism and shared memory is unfortunately similar to extensions that have been added to UNIX: neither has a truly general communications structure that can support the types of investigations we intend to make. Obviously, these shared memory semantics are not easily extended to the loser coupling of a network. (In fact they have not even been implemented on a tightly coupled multi-processor system.)

3.3.2. Conclusions

In general, the hardware support for memory management is the primary constraint on communications data transfer bandwidth. Systems that use shared memory as their only communications mechanism require coherence within that shared memory. This type of coherency can be described by the following guarantee:

> If two processes share a block of memory, and one process changes two locations in the order X then Y, then the other process will not see evidence that Y changed before X.

Unfortunately, it is difficult to make this guarantee without requiring that there be only a single copy of any shared data. In a single-processor configuration this requirement is acceptable, but in a multiprocessor situation it requires extra traffic across the communications medium (bus, network, etc.) to arbitrate ownership of that copy.

This problem becomes apparent in the design of both hardware memory and software disk caches: if the underlying copy is shared, the caches need a mechanism for invalidating each other's copies. A simple solution in a memory cache is to make it *write-through*, but this results in bus traffic on every write to memory. An enhancement is to have the users "negotiate" for a single writable copy of the data which invalidates all readable copies, but such negotiation still requires that all processors synchronize at the point of negotiation. Furthermore, none of these mechanisms works acceptably in a network environment in which long latencies make such negotiations prohibitively slow.

3.3.3. Language Based Systems

The goal of the LISP Machine System [WEINREB] is to create a powerful single user software development base on which large, complex systems can be quickly written and debugged. In contrast to the model of communicating independent processes, message passing in the LISP Machine Flavor System is simply a canonical form of function call with a generic operation. The single shared address space of the processor is used to let independent tasks communicate and synchronize. Use of single single, large, uniform address space has sidestepped a number of important design issues. However, it is at odds with our desire to map single-machine semantics onto a group of machines, whether tightly or loosely coupled. In addition, it makes it difficult to isolate segments of a large system: a failure in one segment often degrades the reliability of unrelated parts.

Many languages (ADA and Concurrent Pascal) support intertask communications, but like the LISP Machine System are wedded to use of a single shared address space and a single language. Our experience has been that no single language environment can really satisfy all the needs of a diverse user community. An operating system should support a virtual machine that lets a variety of languages coexist. In this way common services (editors, loaders, text formatters, etc.) can be used across individual language environments.

4. THE TRIX KERNEL

4.1. Overview -- Organization and Semantics

4.1.1. Organization

The TRIX virtual machine includes a number of *kernel calls* extending the host machine's instruction set to support control of some strongly typed objects. The kernel also implements the lowest level device drivers, mapping their hardware implementation into the TRIX communications mechanism.

4.1.2. Execution Model

This section would normally be titled "Processes" but TRIX has no simple concept of a process. The "locus of control in an address space of resources" normally associated with a process is divided into two objects: a *domain* and a *thread.*

A domain resembles a conventional address space: it consists of a set of addressable, untyped, memory words (grouped into contiguous segments), and a set of strongly typed data objects, namely *handles* on *ports*, both of which will be discussed later. In our implementation the handles are among the objects maintained by the kernel for the user. The user accesses them with corresponding *handle-IDs.* The significant difference between a domain and the accepted idea of a process is that a domain is static; it does not have its own program counter, stack and stack pointer, registers, etc. In future implementations, a domain's static nature will simplify the use of paging and other methods of migrating data to stable storage. The states of important domains can be saved in this manner, and they will be able to survive crashes.

A thread is a single sequential path of execution under TRIX. As the unit of multiprocessing, it incorporates all the properties of a "process" that are not a part of the domain. In general, more than one thread may execute concurrently in a single domain, and a domain may exist without any thread executing in it. Each thread can access its own stack as well as the resources of the domain in which it is executing. Also associated with a thread are: saved register contents, scheduling information, and a *data window* that will be discussed later. Threads are created with a *spawn*() call, which initiates execution of a new thread (with an empty stack) at a designated point in the spawning domain.

Under UNIX, a user performing a single task uses a number of processes, all but one of which is waiting for the others to complete execution. Under TRIX, a user with only a single activity has many static domains but only one thread. In addition, a domain calling for asynchronous activity (for example asynchronous I/O) can achieve it by explicitly spawning threads that run in parallel. Because multiple threads are the sole means of initiating concurrent activity, the spawn() mechanism has been made as simple as possible, while preserving the functionality needed to support TRIX's communications mechanisms. This simplicity is reflected in a newly spawned thread's being given an empty stack, in contrast with the duplication of the stack necessary to support a UNIX fork().

4.1.3. Communications Mechanisms

A *port* is an entry point into a domain (its handler). It is designed to let other domains communicate with the domain creating it. Ownership of a port cannot be shared or given away. In contrast, some systems (among them ACCENT) let the port's owner give away ownership of a message link's receiving end, in order to make the overall system more fault tolerant. When a server is an active process, this feature may be useful, but the static nature of a domain makes it less important.

Each port is associated with one or more references to it (*handles*), that the user specifies using *handle-IDs*. A thread executing in one domain can use the handle to request service from another domain by doing a *request*() on a port into that domain. A request() is an interdomain procedure call with a stylized convention for passing system-protected information. The corresponding *reply*() mechanism lets the thread return to the calling domain. There is also a *relay*() mechanism that is an interdomain branch: a thread running in the new domain will reply to the original requester rather than the domain executing the relay.

The *newport*() call takes three arguments: one specifies the priority at which the entering threads will run (after a request) and two data words that will be passed when a requesting thread enters the handler domain. These words are used to specify the handling function (i.e. the initial program counter) and a pointer to the database containing information needed to process the request (called the *passport*). In this way, the handler can identify on which port a request() was made. (Although one word would have sufficed, we supported two because we generally used both.)

Currently, four types of information are passed in a request()/relay() or reply():

1) An optional handle is passed by including its handle-ID. The passed handle must be owned by the domain making the request. Since this domain loses access to the handle when it is passed, it must explicitly duplicate the handle (with a *dup*() call) if it wishes to retain a copy.

2) An *opcode* specifies what operation is being requested. If the operation is a reply this opcode becomes a *replycode*, and it specifies either that the request was successful or the reason for the failure.

3) Up to three words of uninterpreted data pass basic information, operands or reply status. (Based on our experience with TRIX-0 messages, three words are sufficient.)

4) In a request, a *data window* is passed with READ or WRITE access (or both). Conceptually, the requesting thread carries a pointer to a data buffer. This buffer is mapped into the handling domain's address space whenever the thread is executing in it. In actual implementation, limitations imposed by the available memory management hardware may require that accesses to this data be emulated with kernel calls.

In a request(), three opcode bits are interpreted by the kernel. One of these bits indicates that the optional handle is present; the other two indicate what combination of READ/WRITE access is being passed for the data window.

Only two bits are interpreted in a reply(): one indicates that a handle is being returned and the other, which will be discussed later, indicates a "fatal" error during the request.

A thread making a request may use any contiguous region of accessible memory as its data window. This region must be either the domain address space, the thread stack, or the current data window. If a data window is passed by a relay() the domain's address space is not included in the accessible regions of memory. (Otherwise, a reference would be created to the domain's address space and the domain could not determine when it was no longer in use.)

This decision, to make the data window inaccessible to other threads running in the handling domain, was based on the desire to have a clean semantic model of how the data window can be used. In particular, TRIX avoids expanding the data window concept to a real memory access capability. Our experience indicated that such extension results in a conflict between two requirements: one, that a capability's life should extend only extend over the life of the request, and two, a desire to avoid complications resulting from the capability's revocation. In particular, if access were passed out of the handling domain by another thread's request, it would be effectively revoked when the original thread did a reply(). This situation could then result in the second thread getting some type of error or fault when it attempted to access its data window.

An optimal implementation of data windows would avoid the need to copy the data as it passes through interposed handlers. With proper hardware support, such an implementation would map the window on some other address space into the handling domain while the thread is running. Due to limitations on our memory management hardware, access is currently achieved through the *fetch*() and *store*() *calls*, which let a thread move data between the data window and the address space of the domain in which it is running.

When a port is freed its owner must be notified so that any resources that have been allocated to deal with the active connection can be freed as well. The *close*() *request* allows the owner of a handle to give up its use. This is the only request that is specially interpreted by the kernel. For efficiency the handler is notified only when the port is actually free (the close() request is ignored if there are other active handles on the port). At the point of a close() there may be threads executing in requests on the closed handle; they are completely unaffected.

4.1.4. Kernel Supported Handlers

The kernel implements a few basic handlers that are needed for the initial execution of TRIX. The first group of kernel handlers are the ports into the kernel that are associated with low level device drivers. These handlers translate the hardware interface of the device (device registers and interrupts) into the TRIX request/reply semantics. The disk and serial port drivers are in this class and handles on them are identical in function to handles on user domains. Note that the teletype driver, the network driver, and the file system are not in this class of low level drivers but simply use the normal communications mechanisms to communicate with the serial port, ethernet, and disk, respectively.

Another kernel handler is the synchronizer. It accepts two requests: *sleep*() and *wakeup*() each of which takes a number as an argument. This is the only mechanism that allows a thread to be taken out of the pool of runnable threads. It is identical to the UNIX sleep/wakeup mechanism which, by convention, chooses the numbers to be addresses of locked data structures. A sleep() suspends the execution of the requesting thread until a corresponding wakeup() occurs. This sleep/wakeup correspondence only holds among calls on handles to the same port. Different synchronizers are completely independent, but a single synchronizer may be shared by all the domains with handles on it.

The most complex of the kernel handlers is the domain handler. When a domain is created, the user is returned a handle that accepts requests to access and modify the resources of that domain. Among these requests are ones that allow the initially empty address space to be read, written, or initialized from a handle. (The handle is to a file containing an executable image.) Another request to the domain handler allows a thread to begin executing in the actual domain at the initial entry point.

From here on we will be careful to differentiate between the domain and the kernel domain object. The former is something in which a thread executes, while the latter is a small entry in a kernel table containing the information about the domain. After a request to a handle on a port created by a domain, the requesting thread begins executing in the domain's address space. A request to a handle on the kernel domain object results in the thread continuing to execute in a special kernel domain (the one that also includes the low level device drivers).

All the kernel handlers can be simulated by user level domains but represent mechanisms (an interface to device interrupts or the scheduler) that must be supported somewhere in the kernel. The fact that they fit so well into the normal communications scheme justifies the choice of the TRIX mechanisms.

4.1.5. Object Naming

Different TRIX objects are named in different ways. The user references handles using a handle-ID that is passed (and protected) by the kernel. These handle-IDs are not constant across communications: in the process of passing a handle in a request() or reply(), its handle-ID may change at the discretion of the implementation. This avoids problems that might occur when the mechanism is distributed across loosely coupled processors.

Ports cannot be directly referenced: the handles on a port are indirect references to it. (Creating a port (using the *newport*() call) returns an initial handle for the port.) It is important in the development of garbage collection strategies that even the domain that created a port cannot create a new handle on it without doing a dup() on an already existing handle. Nor is it possible to destroy a port: the port is freed when the last handle to it is freed (using the close() request).

There is no way to explicitly refer to threads. Implicit to a running thread is its own identity, and a few calls effect its state with respect to its current domain. The only other relationship among threads is the hierarchy imposed by the parent/child/sibling relationship when they are spawned. This relationship allows the TRIX signaling mechanism, *recall*(), to reference a tree of child threads. The ability to deal with subtrees of activity is essential in a system which has a great deal of parallel activity. It is often difficult to manage in systems where processes only have unique names. If desired, a new thread can be detached from this hierarchy at the time it is spawned to form an independent tree of activity. This is useful for spawning background tasks that run independently.

The possibility of associating a thread with a handle that could be used to control it or kill it was investigated and discarded because it does not seem to be required to solve any interesting problems. The main use of UNIX process IDs that our mechanism does not already support is allowing a process spawned by another user to be killed. Since the handles associated with such threads would not be available to the process attempting to do the kill(), they do not seem to be useful. (This mechanism can be built using the existing TRIX mechanism if it is found to be useful.)

Eventually some ad-hoc mechanism will be added to let the equivalent of the super-user on UNIX stop any running thread. It will most likely take the form of a kill handle into the kernel that will allow a user to ask the kernel to recall any thread and will require some new name space (other than handles) for referring to the threads. (This type of blemish is generally needed in a capability based system to allow it to be used in a production environment.) More significantly, since it requires that a user have kill handles for all machines on which a thread might run as well as a way to determine the machine on which the thread is running, this kill handle does not extend cleanly over a network.

Domains are referred to with a handle that allows the user to communicate with the actual kernel object containing the information about the domain. A number of requests exist to allow the user to manipulate the domain (particularly its address space) if he owns a handle on it.

As this discussion indicates, handle-IDs form the single name space available to the user. At present the name space is global but this is not of particular interest to the user since (unlike most systems with a shared name space) a given name is only accessible to threads executing in a single domain. The resulting protection mechanism is capability based with handle-IDs playing the role of capabilities.

Any more sophisticated naming/protection functions (for example those normally performed by a file system) must be built on these underlying mechanisms. A consequence of this is that there is no limit placed upon the number of handles a domain may own. Such a limit would make it difficult to build directory objects that can contain arbitrary numbers of handles. In more hostile user environments, it might be necessary to limit resource use in some way, but this would not be a semantic limit and it might be very loose in the case of a "trusted" domain.

The use of a single uniform mechanism for all communications and naming allows a domain to masquerade as a well known service. For example, a network server can transparently forward requests for remote objects to a corresponding network server on another machine where the work is actually performed. Balancing the power derived from such a system is the difficulty with which it can be constrained at all. The powerful mechanisms that allow the distinction between user and system software to be blurred also make it difficult to build hierarchies of dependability (organizations in which services only rely upon other services that are *at least as* dependable (trusted) as themselves).

4.1.6. Preemption and Scheduling

The TRIX thread abstraction allows there to be more than one active thread in a single domain. In general, it is desirable for the semantics of such a situation to allow all such threads to preempt each other. When tightly coupled processors are available, two or more threads can be run at the same time in the same domain. Unfortunately, programming in an environment with no control over preemption requires that programs control the use of shared resources by using explicit locking in critical regions. UNIX takes advantage of the fact that a non-preemptive execution model results in code that is clearer than the corresponding code with explicit locking. An undesirable side effect of this non-preemptive model is that it is difficult to transport UNIX to a symmetric, tightly coupled, multiple processor environment. (UNIX runs quite successfully on a pair of processors when only one of them executes the kernel code [GOBLE].)

The TRIX scheduler supports a simple mechanism that provides priority scheduling of both parallel and serialized computations among the threads in a single domain. The *spl*() (Set Priority Level) call changes the priority of the calling thread relative to the domain in which it is currently executing. (Entry to a domain through a port sets the priority of the running thread to the priority specified when the port was created.) This priority is not a mechanism for guaranteeing real-time response: it is simply a way to control concurrency. In particular, it is not meant to make it more likely that some thread will be run. Instead it allows the user to describe many interlocking situations very succinctly. (The priority should probably be referred to as the *preemptability* of the thread, but history prevails.)

The semantics of the priority scheduling guarantee that a thread running at one priority is never preempted by a lower priority thread. There is no guarantee that the arrival of a higher priority thread will immediately preempt one of lower priority. This indeterminate transit time for a request() is not explicitly visible to the user since threads cannot determine the order of operations that are not explicitly synchronized (by executing in a single domain). This property is important since there may in fact be intervening domains (including network servers) that actually result in such delays. Note that two synchronized requests on the *same* handle must be received in the order they were made. A program doing sleep and wakeup requests on a synchronizer handle may conditionally do a sleep request based on the outcome of some test. It uses the priority to guarantee that threads that might change the condition (and do a wakeup) are excluded during the period between the test and the request. If the requests are not received in the order they are sent the sleeping thread will never wakeup (due to the non-counting nature of sleep/wakeup).

A similarly loosely defined concept of unsynchronized time is found when the spl() call is used to increase the priority. If multiple threads are running concurrently at the same priority and one of them increases its priority, the scheduler is allowed to delay the effect of the spl() an arbitrary length of time (by temporarily suspending that thread's execution). This can be used to to maximize the use of the available tightly coupled processors by allowing the concurrent (low priority) threads to run as long as possible. When the priority of a thread is lowered it may have to be suspended immediately if there are other runnable threads of higher priority.

The scheduling mechanism is further-enhanced by using the two least significant bits of the priority to specify that:

- the thread cannot be preempted by threads of equal priority while it continues to run in the current domain (it stops running in the domain when it does a request() or reply()).
- the priority of the thread will remain associated with the domain during a request() into another domain. In effect, the requesting thread can be considered to continue running in the domain during the request.

Though domains are currently scheduled using a simple round robin scheme, it is easy to envision more complicated approaches that schedule domains based upon the real time latency requirements of their currently active threads. In addition, each domain could have parameters that adjust its responsiveness to both the arrival of high priority threads and priority changes. This part of the scheduling mechanism is semantically transparent to the user and has not been fully investigated.

4.1.7. Exception Handling

An important aspect of an operating system is exception handling. Of particular interest in a highly asynchronous system like TRIX is the ability of a "parent" to abort any current activity of its "children": This is often used to recover from some unforeseen situation. An example of this is the UNIX *signal*() mechanism which allows a user at a terminal to interrupt any of his running processes. In many systems this interrupt mechanism is weakly tied into the mechanisms used to handle hardware and software exceptions. In TRIX there is a single mechanism that uniformly handles both: the abort mechanism, *recall()*, simply causes a special exception in a child thread.

In TRIX it would be simple for a thread do a forced reply() in response to any exception. This is undesirable when the return from a called domain requires that resources be reset to a consistent state. Data structures, for example, may be partially allocated or locked. Our chosen alternative is to allow every domain to designate a handle that will receive exception requests. When an exception occurs during a thread's execution in a domain, a forced request() occurs to this handle. This handle may be on another domain (which might be a debugger), or on the domain itself (allowing the domain to leave its own state consistent).

After either the recall() or the exception causes a forced request(), the resources of the domain must be made accessible to the exception handler. Since this has to be accomplished using the regular request() mechanism, there are two possible ways for this access to be passed to the handler -- through the data window or through a special handle. The data window could be a pseudo data window; it might point to a pseudo address space which gives read and write access to all the resources of the excepting domain (even though they are not a single contiguous region of accessible memory). With this access the called domain could do whatever is necessary (to attempt) to solve the problem or clean up before control is passed back. Unfortunately, this maps very poorly across networks since it would require that the entire address space be copied across the network to support the desired semantics (the transmission of the data window can not be demand-driven since the semantics of the window preclude any intervention by the windowed domain). The alternate solution of transferring a handle with the desired semantics is also less than perfect; the lifetime of the exception handler's access should be limited to the duration of the exception (and its resolution).

The solution we have adopted is to use the data window to give access to the state of the thread (stack, registers at the time of the exception, etc.) and to require that the exception handler have a handle on the kernel domain handler if it needs to access the domain's resources. This is consistent with the use of the exception handler as a debugger, since, in general, it has created the running domain and can have a copy of the domain's handle.

Normally the exception handler simply does a reply() to continue running in the excepted domain. On occasion, the handler must force the excepted domain to do a reply. This double reply from the exception handler is facilitated by using the bit in the *replycode* that indicates a fatal error. It is set to force the double reply. If there is no exception handler, the default is equivalent to a domain that does a reply() with the fatal error bit set (thus forcing a reply() from the excepted domain).

4.2. Resource Reclamation

The ability to reclaim and reuse resources that are not currently in use is a central part of any system. The use of algorithms and structures that "get tired" over time so that the system must be brought down periodically to invoke some type of restorative mechanism is unacceptable and, as a rule, we have rejected any mechanism that loses resources over time. For example, the need to occasionally compact the disk when using the MERT [BAYER] contiguous-extent based file system is a serious inconvenience.

Though domains, ports, and handles must all be reclaimed, the garbage collection problem is not simply one of freeing kernel resources. User domains need to know when one of their entry ports is no longer in use. Using the close() request to terminate access to a handle allows the creating domain to receive a final request notifying it that the port is now free. This request may perform one or more of the following functions:

> Often a handling domain (for example a file system) has internal resources allocated to a port that it can free when the last handle on the port has been freed. These internal resources normally describe the state of the object (in the file system example they include the internally cached inodes that the passport points to).
>
> Another use of the mechanism is to change the state of some other handler. An example of this is the implementation of an *exclusive use lock* on a port. One such implementation is to have an exclusive use request return a new port that can be used to access the original object while accesses to the old port wait for the new one to be freed. (Note that we do not try to lock a handle since there is no way to differentiate among the multiple handles on a given port.)
>
> The most complex cases use the close request to allow the handler to perform a termination action. This is similar to the close action in a UNIX device driver that is used to put the device (real or simulated) into a known state. Examples of this use are: rewinding a tape unit, formfeeding a printer, and terminating a network connection that is no longer in use. Notice that these different situations require a range of processing. Rewinding the tape simply involves putting a command in a device register, while closing a network connection can require a fairly complicated protocol exchange (particularly when there is data that has been buffered in the network handler that must be transmitted first).

When the communications topologies are constrained to avoid loops, the use of an explicit close() (and associated reference counts) allows all unused resources to be reclaimed. A common way of imposing this constraint on the topology is to force it to be built using constructive operations (i.e. the set of objects with which a domain can communicate is frozen when a reference is created to that domain). Operating systems often impose this constraint on interprocess communications, and in UNIX it is derived from the lack of a way to pass file descriptors for pipes to unrelated processes. This is similar to the constraints that pure applicative LISP places on the data topologies, and in either case the need for garbage collection is avoided.

An alternative (and stronger) constraint is to only allow transformations on the communications topology that cannot result in loops; this is often used keep a file system tree structured. The UNIX file system imposes this constraint by limiting the use of the link() system call to create links only to regular files (which are leaf nodes in the file system tree structure).

TRIX does not constrain the use or transmission of handles between domains. As a result, loops can and do form in the communications topologies. An abstract view of the running system is as a collection of threads that move from domain to domain across interdomain links (handles). Because these structures are all implemented by the kernel, the topology is visible even when the user language is not strongly typed. Therefore the kernel can perform garbage collection based upon accessibility.

The concept of accessibility corresponds to the more standard concept of a process being active in a process based system. A domain is *accessible* if some series of requests and replies on existing ports can result in a thread running in it. There is also a stronger concept of a domain being *interesting* if it is both accessible and some activity in it can influence the "outside world" (most often taken to be the I/O devices). After investigating the feasibility of garbage collecting uninteresting domains, it was decided to use the computationally simpler notion of accessibility.

The TRIX garbage collector finds all the inaccessible domains (including those that are in inaccessible communications loops) and frees them. Freeing a domain actually frees two resources: its (untyped) address space, and the (strongly typed) handles it may have. Dealing with the address space is simple since it is never shared with another domain and there cannot be outstanding data windows on it (since they must be associated with a thread that could eventually return to that domain and thus would prevent it from being garbage collected).

Freeing the handles that are currently owned by the domain is more complicated. The first issue to be dealt with is what thread(s) will be used to do the close requests on the handles owned by a garbage collected domain. There is no thread whose task is specifically garbage collection; instead, it is performed by the thread needing the (unavailable) resources. This thread should not perform the close() requests since that would serialize the domain requesting the resource on the replies to close requests to domains with which it has no connection. A close request can, either for legitimate reasons or because of the perversity of the handler, take arbitrarily long to execute. (For similar reasons all the close requests should be done in parallel rather than in series.) Our solution is to spawn the required threads at the time of the collection. We have bound the number of threads any garbage collection pass may require by allowing only one thread to be spawned for each domain receiving a close request. This serialization is acceptable since it can only effect the internal workings of a single domain. That domain can alleviate any problems by immediately spawning an independent thread to perform the close activity (allowing the garbage collector's thread to return immediately).

A more complicated issue is determining the semantics of close requests sent by the garbage collector to a domain that was being garbage collected. Such a request reactivates the domain and reconnects it to the outside world. These requests can be used to give the domain a last chance to perform any pending I/O. Among the places this is useful are: a network driver that must shut down its connections to the outside world before it is collected, and a file system that must flush any internally cached information out to a disk. The concept of encapsulating a resource with state (like the network or a disk) so that that the state the user sees is not identical to the real state is an important approach to increasing performance (it supports both caching and pipelining) and had to be preserved.

The problem is to allow all domains to be garbage collected when they are no longer used, while still allowing domains that perform some state caching to be collected properly. We initially designed a multipass collection scheme in which the domain would get the close requests during the first pass and, if it had not been reconnected, it would be collected on the second pass. This seemed adequate but requires the topology of the communications to be frozen between the two collections to avoid the chance of missing the fact that the domain had been (at least temporarily) reconnected and was thus not immediately collectible. Freezing the state of the communications topologies during the garbage collection makes incremental garbage collection impossible.

Our eventual insight was that our model of these caching domains was incorrectly formulated. The flushing of the cache can be viewed as an activity that is driven by the event of an incoming request. It thus requires a final event to guarantee that the state is completely updated. An alternate view is to see that the dirty cache can have an associated cache flushing task, which corresponds to the asynchronous flush of a *write-behind* cache. Though this asynchronous task can be driven by the request events, a cleaner solution is to have an independent thread that exists whenever the cache is dirty. This thread can run at a low priority so that, when there are no other demands being made of the domain, it can update the out of date state. When the state is up to date, the thread terminates. Thus, the domain is only collectible when the user visible state is synchronized with the actual state.

The described scheme allows any inaccessible domains to be collected while preserving the possibility of a close request notifying an active domain when a port has been freed. The remaining problem results from the granularity of the garbage collection algorithm. Our model of a domain is that a thread entering on a port has access to *all* the resources of the domain. Though this is true, it ignores the structure of the domain itself.

Take the example of a domain that is providing the functionality of an environment handler. (An environment is a detached directory that associates names with the initial handles available to threads executing in a domain.) Though an environment may contain an explicit handle on another environment, it is not possible to access the handles an environment contains without a handle on the environment itself. This should lead to an implementation in which each of the environments is a separate domain that is treated independently from the standpoint of garbage collection. Unfortunately, the simplest implementation of an environment combines all the these conceptually independent handlers into a single domain, an organization that leads to some difficulties in the design of the garbage collector.

Although the environment's implementation only gives an entering thread access to those handles contained in the associated environment, the garbage collector sees only the external structure that gives any thread entering the domain access to all its handles. As a result, the garbage collection process is overly conservative.

If the domains had been left independent this problem could have been avoided at some cost in organization. Among the reasons for combining independent handlers into a single domain are:

- to increase resource sharing between very similar domains.
- to simplify sophisticated handlers that could not otherwise use the simple synchronization and exclusion mechanisms available to threads running in a single domain.

Since the garbage collection problems result from the domains being aggregated together, their solution requires that the internal structure be visible to the kernel while keeping all the handlers in the same domain. One possibility is to create ports with a *subdomain* number that identifies how entering threads fit into the domain's fine structure. Threads entering a domain are associated with the subdomain of that port, and can only access handles that belong to that subdomain. This allows sharing among otherwise separate domains. A newly created port can actually have one of three numbers: zero (which is the number for globally shared resources), the number of the current subdomain, or a new unique number. Note that though subdomains would not themselves be collected, ports entering subdomains would not be assumed to have access to all the domains resources. If the subdomains are guaranteed not to have references to each other, unused resources will always be collected.

One drawback of this solution is that the enforcement of the subdomain/port association (at the point the domain is created) cannot be reconciled with the use of a handle for creating ports. There is no way the kernel handler can determine the number of the current subdomain without using some mechanism that would make interposition impossible. This is the reason newport() is a call instead of a request to the domain handler.

The problem of unused resources not being collected in TRIX is similar to problems that occur in LISP systems that depend upon a combination of a large virtual address space (with a significant amount of backing store) and an occasional reboot to start from scratch. For example, in the LISP Machine System, it is the responsibility of the user to explicitly free open files (with a close). If the last of the references to an open file stream is garbage collected, the external resources (network connection, open file, etc.) can not be freed.

4.3. Implementation

4.3.1. Memory Management

The TRIX memory management subsystem is built on segments (contiguous allocations of memory) that may be paged and/or swapped onto some secondary storage device. Because the secondary storage device is specified with a handle, the system can page or swap to any handler supporting random access I/O (e.g. a local disk or a network disk server).

Segments are created either as a side effect of manipulating the address space of domains or to support the stacks that are created in the spawning of threads. Though there is no way for the user to directly refer to a segment, they are discussed because of the semantics they impose upon the domain's address space. Depending upon the operation that creates a segment, it may be initialized in a variety of ways:

- zero filled,
- with data explicitly written to it, or
- demand loaded from a handle.

The semantics of the memory management leaves undefined the time at which demand paged segments are initialized from the specified handle (by performing a read request()); the only guarantee is that it will occur before the segment is accessed (either implicitly, with memory references if it is mapped into a domain, or explicitly, by doing reads and writes on the domain handle).

Segments also have a number of protection modes:

- read-only,
- copy-on-write, or
- write-back on write.

Read-only and copy-on-write pages are not a semantically visible sharing mechanism.

TRIX avoids the temptation to implement operations in which a large piece of existing address space must be duplicated (as is the case with the UNIX fork()). The problem with fork() is that the address space that is being duplicated may represent an arbitrary pattern of sharing among the address spaces that were "duplicated" to form it. Such situations make it difficult to implement the copy-on-write that is needed to avoid thrashing at the point of duplication. (Further problems result from the fact that most memory management hardware [VAX, NS16082] does not properly support arbitrary sharing on a page level.) Simpler uses of copy-on-write allow an address space to be initialized from a *read-only* file; the changes made to the local copy are never the basis for a new version.

The write-back on write mode allows a handle to be mapped into a segment and have changes made to it be reflected back to the handle with write requests. TRIX minimally defines when this write-back takes place:

> a handle that is only accessed by being mapped into a single segment will be consistent with respect to changes made by accessing that segment (both implicit memory accesses and explicit read/writes to the domain handle).

Beyond this there is no guarantee as to when the handle will be read or written. In particular, if the same handle is mapped into two different segments there are no guarantees of consistency between them. This ambiguity is important because it allows multiple copies of the data to exist when appropriate for increased efficiency in a distributed environment without precluding sharing when it is feasible. Whether sharing or copying is used in any situation will depend upon hardware considerations such as the underlying memory management architecture.

Because these modes of memory mapping are not defined to require consistency, the mechanism is not a communications mechanism but rather a way to make programming simpler. Applications that need stronger guarantees for synchronization or atomicity should explicitly implement them with other communications. The one support given the user is that the kernel domain object will react to an *update* request by writing all the dirty pages from the segments mapped into domain. The request returns when all the dirty pages have been cleaned once (by then some of them may already be dirty again).

The data window, as passed in a request(), is simply a reference to a segment with a base, bounds, and access mode (read and/or write). The distinction between the use of the data window and typical shared memory semantics is that we have not defined the semantics of the data window to specify whether it is a copy of the data in the original domain or a real pointer to the original copy. This ambiguity is again completely intentional. Though we take no issue with the use of shared memory to increase efficiency, we object to guaranteeing the semantics to be those of a consistent single copy of the data. By leaving the semantics minimally defined it becomes a detail of the implementation. Like the lack of definition of the order of operations in many compiled languages (to leave code generation minimally constrained) we take the same liberty with respect to the implementation of the data window. Even in a single implementation it may at different times be defined to be a single consistent copy of the data or multiple copies of the data. The guarantees of consistency we give the user are that:

- any readable data will be read out of the calling domain's address space sometime after the request() initiates and before the fetch(), and
- any data modified with a store() will be reflected back into the calling domain before the the request() returns.

Although the data remains in the address space of the requesting domain, there is no guarantee that the requester will be able to coherently access the data window during the request().

Current uses of shared memory fall into four categories:

- to save physical memory space by reducing the number of copies of shared read-only data (particularly the text of pure procedures),
- to extend the address space of a machine by allowing multiple processes to act as one large process,
- to support missing features that can be built with shared memory (particularly synchronization mechanisms), and
- to make communications more efficient, even in the case of independent processes.

The sharing of read-only data is simply a method of saving physical memory and I/O bandwidth. Since it is not semantically visible to the user it can be ignored. The use of large address space machines make sharing unnecessary for extending address spaces. TRIX eliminates the other uses of sharing by supporting the necessary primitives in the system and using semantically transparent sharing for efficiency.

4.3.2. The Request/Reply Mechanism

The request()/reply() calls are the basis of the TRIX communications mechanism and, because of this, they have a significant effect upon the overall performance of the system. This section will discuss in greater detail the implementation of these mechanisms.

There are two stacks used in processing most TRIX kernel calls: the thread (user) stack and the kernel/interrupt stack which is used to support the kernel machine extension by guaranteeing the availability of a stack to handle interrupts, traps, and exceptions. Unlike UNIX, which uses an independent kernel stack for each user process, TRIX needs only one kernel stack per physical processor.

The thread stack is used to run user level programs and is mapped out when a new thread is run. Conceptually, a new thread stack is created when a thread does a request into a domain. In the current implementation, the memory segment containing the stack of the requesting thread is extended (if necessary) and remapped to protect the portion in use before the request. This reduces the bookkeeping required to deal with threads at a request while preserving the semantics that would result from a new segment being allocated. In addition to the protection issues, isolating the execution of a thread in the handling domain from the state of the thread before the request is important because it allows a pair of independent threads to be used to simulate the mechanism in a loosely coupled environment. To support requests across a network, each machine has a network handling domain that, upon the receipt of a request, sends a message to a counterpart on the remote machine. The remote domain then spawns a thread to actually process the request. This would not be transparent if the state of the requesting thread was accessible after the request.

The only time the kernel deals with the thread stack is to remap it (for protection at a request/reply) and to automatically extend the memory segment if a memory fault occurs with the stack pointer out of range. A third, independent, stack is used to save the protected portion of the thread state across requests (the *request stack*). Included in this state are: the current domain, the thread priority, the machine registers (during a call), the thread stack mapping, and the currently accessible data window. This stack was not combined with the already existing thread stack because it would have resulted in some unacceptable interactions with the use of virtual memory:

- it could have required paging to push the request stack entry onto the stack during a request call. We avoid the need to allow kernel calls to restart on a page fault by eliminating the possibility of a page fault from anything other than a user level fault and the fetch and store calls.

- garbage collection would require paging the user stack to access the request stack, and each of the required pages would only contain a single request stack element. In general the resources used by the request stacks are minimal when they are packed into pages.

To understand the implementation of the request and reply we will outline the sequence of actions that they involve. Initially the user is running on the thread stack. (This is the case even when we are running in a "system" domain.) To call into the kernel, the thread must execute an appropriate trap instruction, which will leave it running in supervisor mode on the kernel stack. Since some traps resulting from hardware interrupts will cause rescheduling similar in nature to that done explicitly by user invoked traps, it is the responsibility of the kernel to save the user state (registers) on the request stack.

When it has been determined that the call is a request the following occurs:

- The current top of the request stack is pushed and a new element is initialized with the new domain and the initial register values (in particular the program counter and the passport).
- Any passed handle is validated and the handling domain is made its owner. The passed data window is then validated to determine that it represents a valid access to a legal segment and this information is also saved on the request stack.
- The user stack segment is remapped to protect the used stack.
- The thread is taken out of the scheduling list of the requesting domain (affecting its priority) and scheduled to run the handling domain. This may result in the current thread being runnable or may require that the thread be suspended in favor of another thread (if for example the handling domain already has a higher priority thread running in it).

Finally, the return from any trap results in a test to map in the current domain and the current thread stack (either or both of which are ignored if they are the same as the currently mapped domain or thread). Then the user registers are restored and the thread continues running.

The relay (and reply) are similar but they avoid pushing (or popping) an element. In all cases there is some scheduling activity and a remapping of the address space.

4.4. Open Problems

4.4.1. Scheduling

The spl() mechanism is not quite sufficient in that it does not allow groups of threads to be viewed as being totally independent. This is often desirable when two independent handler domains are combined into a single domain. A thread that wants to lock out other threads with which it is interacting must lock out even the independent threads of the same priority. The priorities cannot be allocated among the independent handlers since then only the highest priority thread will actually run and there will be some artificial serialization between the handlers. A more sophisticated scheme could couple the priority to the subdomain number described in the garbage collection section. We have postponed solving this problem until we have more experience with the use of the priority mechanism (particularly in a multiprocessor environment).

4.4.2. Instantiating New Objects

Because we have not been able to uniformly resolve the question of how to create a new instantiation of an existing object type, we currently use an ad-hoc method. In a strongly typed language the type of a created object must be known at compile time by giving a proper and precise declaration. This declaration is precise enough that any two objects created with it are considered to be the same type.

TRIX is, unfortunately, a weakly typed environment in which the type of a handler is best defined by the protocol that it obeys. (The protocol is the set of OPCODES to which it responds.) This makes it difficult to describe the type since even the protocol is not complete information: the protocol of a "file with incore caching of data" and an "uncached file" are the same, yet they are not interchangeable. Nor is it possible to ignore the stable storage device with which the file is associated since it may be that the user will prefer one disk over another (for example a local disk may trade the cost of hardware against the longer latency and lack of availability of a remote disk). The LISP Machine flavor system allows types to be dynamically created by specifying the protocol they must obey (including protocols specifying user transparent aspects of the functionality like buffering). This is an extremely powerful mechanism that requires a great deal of runtime support and a single program address space: in effect the creator must be able to ascertain the internal structure of the created object. This does not extend to the C/TRIX environment.

A further issue is deciding with whom a program should communicate to create a new file or directory. One possibility is a server that accepts a protocol list and returns an instantiation of the simplest known object that obeys it. This protocol list would have to capture semantic features (e.g. buffering). Alternatively it could associate objects and object type names, which would require an additional global naming scheme (in addition to the OPCODE names). Though this server could be global it would be more powerful to have it be part of the local environment of the creating domain. This would allow new objects to be made known to the server without impacting the entire distributed system.

The alternative to this approach is to create by example. If you want a new object of a given type you ask an existing object of that type for a new one. This has the appeal of avoiding any explicit notion of object type and is the approach we are intending to pursue.

5. THE TRIX OPERATING SYSTEM

The TRIX "system", as distinguished from the kernel, is the set of user domains that provide the services normally associated with an operating system. These services include: the file system, the terminal/window manager, and the network protocol driver. The current implementation combines some of these services into a single domain known as the *system domain*. The only difference between this system domain and a normal user domain is that the system domain is always mapped in. This reduces the cost of context switching into the system following a request to one of these services.

A file system generally performs two major functions: making a convenient form of stable storage available to the user (the function of a file), and allowing objects to be named (the function of a directory). TRIX creates an object (file or directory) without a name (a handle is returned to the user). This object (or in fact any handle on any object) can then be associated with a name by doing an *enter*(name,handle) request to a directory. For a variety of reasons most operating systems do not separate these functions and it is impossible to talk about an object without its pathname. In UNIX, the *creat*() call requires a pathname, and the file is associated with the file system of the directory that contains it. Linking to existing files is also done by specifying two pathnames (the current name and the new name). Referencing file system objects only by name solves the problems of:

- keeping the file system strictly tree structured, and
- avoiding links between independent (*mounted*) file systems.

In contrast, TRIX must deal with a number of the complications that UNIX ignores. When a handle is entered in a directory it must be transformed into a pointer that can be stored on stable storage. For now, the TRIX kernel does not support storing a stable representation of a handle. Adding kernel support for such a representation would require additional mechanism to make it unforgeable. (Note that the unstable handles are kernel implemented communications capabilities and as such are protected by the kernel.)

An interim solution lets a file system transform handles on its own domain into stable representations. These can be regenerated by creating a new port on the file system (a function that exists) and the problem of forged stable representations can be ignored. A new mechanism (the *fold*() call) was added to let the file system determine whether an entered handle references one of its ports. The fold() call determines which local port the handle is associated with and returns the PC and passport that would be passed into the domain on a request on the handle. (If the handle references another domain's port, fold() returns an error.) The number of the local port is stored on stable storage (in the standard UNIX directory representation).

A remaining file system problem results from the dichotomy between stable storage structures and the communications structures that TRIX supports. This is an extension of the problem of garbage collecting the environment handlers that were combined into a single domain. Unlike the environment example, a directory may contain references to another directory that the kernel cannot see. (These are the internal links that are folded for stable storage.) Because the internal links in the directory structure use a reference count scheme to free resources, it is still important to keep the file system in a hierarchical tree structure. The solution, garbage collecting even stable storage, will have to be closely examined to evaluate its cost.

These problems do not stem from the use of the TRIX communications primitives to implement a file system; they simply reflect using the power of the TRIX mechanism to build a more "interesting" file system which was, for reasons of convenience, based on the UNIX disk format. They could have been easily solved by a using a file system structure that is just like that of UNIX, (one that only allows objects to be entered in directories by name).

6. EVALUATION

6.1. Functionality

Our experience using TRIX to implement interesting communications structures indicates that it is functionally quite powerful. For example, the file system implementation is straight forward and demonstrates functionality that few if any, systems in general use today could match.

Another functional test is how completely a UNIX environment can be emulated Our system call write-arounds successfully support UNIX code that does not make use of *fork*(). Although individual uses of fork() can generally be replaced with a spawn() call, there is no clean solution that can be used in all cases. This results from two aspects of the UNIX machine model:

- the semantics of the UNIX process fork/exec/wait system calls, and
- the use of shared file descriptors (which is also associated with fork() and exec()).

The fork() mechanism does not map onto the TRIX thread spawn mechanism. A version of spawn() extended to duplicate the spawning thread's stack would be functionally close to the 4.1BSD UNIX *vfork*() call. (Kernel support of the stack copying is necessary since the parent thread's stack is not available to the spawned thread.)

Fortunately most programs do not make explicit use of fork(). Instead they use the system() routine which includes the entire fork/exec/wait sequence; it can be emulated successfully under TRIX. The limited use of fork() reflects that, even under UNIX, fork() and exec() are difficult to use. For example, they do not interact properly with buffered I/O. (Output buffered before a fork() may be written twice if the accompanying exec() is delayed.) The shell is extremely careful to deal with these issues.

The final incompatibility involves sharing the file offset among the users of an open file descriptor. Although this mechanism is not widely used by UNIX programs, it is crucial to the implementation of the shell construct "(a ; b) > output" (which concatenates the result of running the command "a" followed by the command "b"). By sharing a common pointer into the output file, the execution of "b" continues writing at the point that "a" stops. Our view of this problem is that it is incorrectly formulated. A correct model of the situation is that both command "a" and "b" want to write to a non-random access (serial) file. We prefer to make this explicit by inventing a *serializer* object which can be interposed between the user and a random access file. It will encompass the shared pointer and allow intermingled writes to the single file to work properly.

Another way of achieving UNIX compatibility is used by MERT, which ran a UNIX environment as a system process. The ability to encapsulate a user domain is sufficient to build an independent UNIX environment. This environment would support the fork/exec primitives as well as the shared I/O offset. The problem would be allowing the UNIX subsystem to interact with the more sophisticated TRIX environment. This approach was avoided because we wanted to do more than create a new UNIX implementation with dynamically loadable device drivers.

6.2. Current Performance

It is difficult to measure TRIX's current performance on the Motorola 68000. Running single user, programs have as good or better performance under TRIX as under the UNIX implementation on the same hardware. As this UNIX implementation is the basis for a number of commercial ventures, this is a reasonable comparison.

Programs have two modes of operation: user mode computation (which will of course be unchanged since the same instructions are executed) and system supported computation (which is mostly I/O). Our performance measurements have indicated the following:

- TRIX gives better real time (wall clock) performance than UNIX because it has a shorter path to initiating the (asynchronous) I/O transactions.
- TRIX does more computation in performing I/O. Running single user this computation took the place of what would normally be running in the idle loop.

TRIX's greater computational needs are alleviated by the fact that the its structure is much more amenable to multiprocessor configurations [GODDEAU]. The most successful multi-processor UNIX implementation is the PURDUE Dual-Vax UNIX system. It runs user processes on two processors, but forces all supervisor code (system calls and interrupt drivers) to run on the master CPU. Though this is extremely effective when the job mix is compute bound, an I/O intensive job mix may run slower bacause of the additional context switching require to move a process to the master processor and back. TRIX exacts no penalty for compute bound job mixes and allows greater concurrency in processing system calls.

Furthermore, an examination of where TRIX is doing its extra computation indicates two bottlenecks that could be considerably better tuned. Much of the penalty results from the abstraction used to implement the file system and terminal handlers. They use the TRIX request()/reply() mechanisms to communicate with the disk and serial port respectively and they do not take advantage of the fact that they run in the privileged system domain. One possible optimization would be to let the system domain access a number of the kernel calls directly (with a subroutine call) rather than the trap that common users are required to use. This would not affect the semantics of the programs but would allow a number of the simpler calls to run considerably faster.

An unforeseen result of the unoptimized implementation was to demonstrate how far the mechanisms could be pushed. Often a communications mechanism is developed to support some small portion of the communications that a system uses (for example, remote login). Its success in that limited domain is then used to argue that it could support all the communications that occur. Our extensive use of the TRIX mechanisms demonstrates that generalized communications can achieve acceptable performance.

6.3. Future Performance

To allow this project to be exported to heterogeneous distributed systems, TRIX was designed to avoid depending upon the special purpose architectural features of any individual machine. This section will outline a few modest changes in the machine architecture that would increase TRIX's performance.

Some kernel calls (like spl()) often simply change a value in a table, and actually require kernel intervention a small fraction of the time. In the case of the system domain, the traps used for these calls can be replaced by subroutine calls that run considerably faster. The use of a microcodable machine would allow a number of the simpler kernel calls to execute in microcode, extending this performance improvement to common user domains.

Since TRIX does more context switching than UNIX, a fast context switch is very important. Under UNIX less than 1 of 4 system calls result in a context switch between processes. (On the PURDUE UNIX, any system call from the slave processor results in two context switches.) Under TRIX, the ratio will be far closer to 1 to 1. (Keeping the fsystem domain permanently mapped reduces the effective number of context switches to that of UNIX.)

Most architectural changes necessary to increase the performance of a context switch can be isolated to the memory management hardware. The following would be useful in a TRIX machine:

- a very fast switch requires that we do not have to load a series of segment or page registers (as is the case in the NU).
- an executing environment is composed of two orthogonal segments (the domain address space and the thread stack) that are changed independently. A single process description register (as is found in the National Semiconductor 16082) is

not sufficient. The P0 and P1 concept of the VAX is far better.

- to protect the stack during a request some protection window (perhaps a base and bounds register) is also desirable. Otherwise, the stack segment must be remapped even if it is independent.
- if the hardware will allow a window to exist on a segment, the data window mechanism could map the data into the handling domain. Then, fetch() and store() degenerate to block move instructions. This is more important than might be immediately apparent, since doing the data window access validation in software is a fair amount of work in an inner loop.

One implementation that would support all these needs is to divide the user address space into three segments which, like the VAX P0 and P1, would be designated by the high order address bits of an access. In addition to independent page maps for each of these segments, they should include base and bounds registers that can be set to a byte boundary.

The commercially available microprocessor that comes closest to this model of a TRIX machine is the Intel 80286. Its segment based memory management system is sufficient for TRIX (though it does not support paging). The drawback of this machine is that segment descriptors are not cached on the processor and loading one (in a 28 bit pointer reference) is rather slow. This is less a problem associated with TRIX's needs, than one of bringing up a weakly segmented language like C (that includes pointers).

7. Summary

The TRIX projects's stated goal was to built a system supporting investigations into the communications needed to support distributed computation. The result is a small, portable implementation of a communications system based on *remote procedure calls.* It lets the user experiment with ideas in communications and distributed systems that would otherwise require extensive "kernel hacking" if they had been attempted under most existing systems. Despite this additional functionality, TRIX's performance is comparable to that of UNIX.

More significant than the resulting software system were the lessons learned during the project. The current TRIX implementation is the third in a series. The first built a system around a data driven communications mechanism. This is an appealing organization since it stresses what seems to be the main problem in communications, moving data. Based upon this experience, we learned that the non-data portion of a communications mechanism is far more important than moving data. Issues of synchronization and out-of-band signaling quickly dominate both the implementation and the structure of the programs that are communicating. Eventually, the implicit synchronization that seemed so appealing at the start got in the way of more complicated program organizations. This contradiction is what led us to abandon the idea of simply extending the UNIX communications mechanisms.

The second formulation, TRIX-0, was a message passing system similar to several others described in the literature. Our choice of message passing was an evolutionary outgrowth of investigations into data driven communications. First, the data-driven stream was reorganized into a stream of data packets (with headers that give information about the sender). This change facilitated mechanisms to support multiplexing. Eventually, this drift towards message passing was replaced by a formal system.

Our initial enthusiasm for message passing was dampened by experience. Although message passing is a reasonable choice for simple program organizations (those that view an incoming message as an interrupt to be handled), in more complex organizations its use obscured the program's underlying structure. These more complicated organizations view the incoming message as independent task to be performed. They include window managers, file systems, and networks. In a message passing system the available concurrency is invisible to the scheduler and there can be artificial serialization among otherwise independent tasks.

The most recent step has been to make concurrent activity explicitly visible to the kernel. To do so, the normal concept of a process is divided into two independent components: the domain (a process's address space and resources) and the thread (the process's active portion including the registers and stack). A simple synchronization mechanism allows multiple treads to cooperate in a single domain to perform a task. The actual communications mechanism is an interdomain remote procedure call (request/reply) which lets a thread execute in a new domain.

Perhaps the most important aspect of this organization is that threads running in different domains may always run in parallel. Unlike threads in a single domain, this parallel activity does not depend upon the processors being tightly coupled. Thus, the TRIX communications structure gives the user more than just a way to implement a window system; it allows a software system to be described so that it can be distributed over a loosely coupled network of machines.

A key feature of TRIX is the minimal way it defines the implementation of data transmission. For a particular situation, it does not specify whether the transmitted data is passed by copy or reference. Often systems are over specified, to the extent that the definition constrains the implementation. Just as many computer languages often leave the order that operations are performed undefined, we see a need to do the same for the semantics of the virtual machine provided by the operating system. By leaving aspects of the mechanisms underdefined and potentially inconsistent, TRIX's communications semantics can be mapped onto a number of implementations. Thus a single mechanism supports communications among domains on a single processor, a number of tightly coupled processors, and a network of loosely coupled processors.

At this point we feel comfortable in claiming that the stated goal has been achieved. Among the projects built upon the basic kernel described above are a simple window-style screen manager and a multiprocessor implementation of the kernel. Unfortunately, although TRIX has been used extensively over the past year, hardware limitations (particularly the lack of a network interface) have prevented us from pursuing some of the projects that originally motivated this work. Among the areas requiring further investigation are integrating access to remote files in a distributed file system. Our next implementation of TRIX will run on hardware that can support these investigations.

REFERENCES

[BALZER] R. Balzer, *PORTS - a Method for Dynamic Interprogram Communication and Job Control*, AFIPS SCJJ Vol 38, 485-489, 1971.

[BASKETT] F. Baskett, J. Howard, J. Montague, *Task Communication in DEMOS*, Proceedings of Sixth ACM Symposium on Operating Systems Principles 23-31, 1977.

[BAYER] D. Bayer, H. Lycklama, *MERT: A Multi-Environment Real Time Operating System*, Proceedings of Fifth ACM Symposium on Operating Systems Principles 33-42, 1975.

[GODEAU] D. Godeau, *A Multiple Processor Implementation of the TRIX Operating System*, M.S. Thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

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

[HALSTEAD] S. Ward, R. Halstead, *A Syntactic Theory of Message Passing*, JACM, Vol 27, No. 2, 1980.

[HEWITT] C. Hewitt, *Viewing Control Structures as Patterns of Passng Messages*, AI Working Paper No 83, Massachusetts Institute of Technology, 1976.

[LAUER] H. Lauer, R. Needham, *On the Duality of Operating System Structures*, Operating Systems Review, 4/2/79. [NS16082] *NS16082 Memory Management Unit*, National Semiconductor, 1982.

[ORGANICK] E. Organick, *The Multics System: An Examination of Its Structure*, M.I.T. Press, 1972.

[RASHID] R. Rashid, G. Robertson, *Accent: A Communication Oriented Network Operating System Kernel*, Carnegie-Mellon University, 4/30/81.

[RICHIE] D. Ritchie, K. Thompson, *The UNIX Timesharing System*, Bell System Technical Journal, 1978.

[SIEBER] J. Sieber, *Active Inodes*, Massachusetts Institute of Technology, 1977.

[STEELE] G. Steele, G. Sussman, *LAMBDA: the Ultimate Imperative*, AI Memo No 353, Massachusetts Institute of Technology, 1978.

[TOPS20] *Monitor Calls User's Guide*, Digital Equipment Corporation, 1976.

[VAX] *VAX Architecture Handbook*, Digital Equipment Corporation, 1981.

[WARD1] S. Ward, "TRIX: A Network-oriented Operating System", *Proceedings of COMPCON '80*, San Francisco, 1980.

[WARD2] S. Ward, "An Approach to Personal Computing", *Proceedings of COMPCON '80*, San Francisco, 1980.

[WEINREB] Daniel Weinreb, David Moon, *Flavors: Message Passing in the Lisp Machine*, AI Memo No 602, Massachusetts Institute of Technology, 1980.