THE RIDGE OPERATING SYSTEM:

HIGH PERFORMANCE THROUGH MESSAGE-PASSING AND VIRTUAL MEMORY

Ed Basart
Ridge Computers
2451 Mission College Blvd.
Santa Clara, California

ABSTRACT

The Ridge operating system is decomposed into processes and relies on message passing for its interprocess communication. Messages and processes are used to improve reliability and extensibility and to facilitate networking. The challenge was to provide a high performance UNIX† implementation in this environment. The technique used was to blend in other operating facilities, such as virtual memory, with the message system. Key aspects of the design were to minimize the number of primitives and to provide support from the Ridge instruction set architecture.

1. INTRODUCTION

As computer technology has evolved over the last twenty years, new market classes of computers have evolved. The 1960's brought the creation of the minicomputer and desktop calculator. The seventies saw the emergence of the superminicomputer and microcomputer chip technology. Now the 1980's have brought personal computers and workstations. The Ridge 32 computer was created to fill a niche in the performance curve between microprocessor-based workstations on the low end and the coming generation of superminicomputers on the high end. The Ridge was designed to explore the high end of the personal computer and workstation spectrum of performance, with a particular emphasis on delivering the most performance for the least cost.

Ridge has concentrated its efforts on the computational intensive portion of the scientific and engineering market, which means large programs, lots of floating point operations, high-speed graphics, fast disks, and high-bandwidth networks. The Ridge is particularly well suited for computer-aided-design and computer-aided-engineering applications.

The Ridge 32 machine was designed as a complete computer systems effort, including a new instruction architecture, I/O bus, and operating system. Later, its style of instruction architecture became known as Reduced Instruction Set Computer (RISC).

What Came Before

The designers of the Ridge system had previously been involved in the development of a 1970 generation superminicomputer, a system that in many ways is in complete contrast to the new system:

a. The system required special air conditioning and power. It usually lived in a computer room.

b. The instruction architecture was extremely complicated, with a very high instruction count and many addressing modes.

c. The various levels of the hardware and software were "closed". Hardware and software were intertwined and the details were hidden from the user, both to protect the naive user and to protect the corporation's technical knowhow.

The problems with this type of system included:

a. The high cost and special environment made the purchase and acquisition of such systems difficult. The six-figure system cost required upper management of a corporation or laboratory to approve the purchase. Then, since the computer was so expensive, major attention was paid to squeezing the most out of it by attaching as many users as possible. Control of the machine was removed from the individual user, and elaborate protection and accounting schemes were put into place to prevent any user from getting too much of the computing resource.

b. The complicated (some would say baroque) instruction set architecture made it difficult to make higher-performance systems, or to lower the cost on similar-performance systems.

c. The closed nature of the system required that all software be created by the manufacturer. This type of architecture is incompatible with today's world, in which a large number of independent software vendors sell programs for a large number of machines.

† UNIX is a trademark of AT&T Bell Laboratories

Design Requirements

The Ridge design had to meet several stringent constraints. First, the machine's performance had to be from one to four MIPS, with prices ranging from $40,000 to $100,000. The machine was to run in an office environment with no special power or air conditioning. It was anticipated that the machine would be used by a small number of users (one to four) who would be part of a project that frequently used one application that consumed hours and processor time and many megabytes of disk storage.

The operating system needed to fill several needs. For the user running one application for hours, the operating system must generally stay out of the way and provide the complete computer resource to the application. At other times, the operating system would be expected to pass large volumes of data to and from the disk, so good file system performance would be required.

A third major need was extensibility. Operating systems are never finished; they are always under construction. Constant changes to the system provide a large and steady supply of bugs. Thus, the design should make additions simple and allow bugs to be quickly eliminated.

TO THE RESCUE: A RADICAL APPROACH

The Ridge system design began as a retreat from the complex, monolithic system to one of severe simplicity. The idea was to base the operating system upon a very small set of principles. This brings to mind the now traditional layering concept. However, a system that is more "flat" rather than "deep" was desired. If the operating system itself was made up of underlying layers, high performance hardly seemed possible. Also, complex applications, such as databases, access the "levers of power" and re-implement many parts of the operating system because the cost associated with penetrating all the operating system layers is too high. The Ridge Operating System was designed to make this unnecessary.

There were two operating system principles which seemed to fit the new mold: messages and virtual memory. Previous systems drawn upon include two message-based operating systems, Guardian[4] and DEMOS[3], and the IBM system 38[6], which implements single-level store in a large virtual address space.

Sections 2 and 3 below give a basic overview of the Ridge hardware and operating system. Section 4 attempts to justify the design strategy in decomposing the system. The "meat" of the article is Section 5, the message system design, hardware support, and message system performance. Finally, system performance, related works, and a critical review of the system is presented in Sections 6 through 8.

2. HARDWARE OVERVIEW

The Ridge 32 is the first commercial example of a new generation of computer architectures that use simplified hardware to provide high performance at reduced cost[2]. The Ridge is a RISC-style machine as described by Patterson[8].

The Ridge 32 utilizes 32-bit architecture to provide both 32-bit arithmetic and 32-bit addressability. Both single and double precision floating point are supported. The processor is implemented in standard Shottky logic, and fits on three fifteen by sixteen inch printed circuit boards. The processor uses sixteen general purpose registers, cycles in 125 nanoseconds, uses a 256-byte code cache with no data cache, and can load or store a 32-bit word to memory every three cycles. Simple instructions, such as integer add or shift, execute in one cycle, while more complex instructions, such as single-precision floating point add, execute in five cycles.

The system contains one or more I/O boards, each of which has its own DMA. The I/O boards communicate with memory over a common bus that is separate from the CPU bus. Minimum system configuration is an 80-Mbyte disk and four Mbytes of main memory. Terminals may be standard asynchronous devices or high resolution bit-mapped displays with keyboard and mouse. Ridge systems are frequently connected to Ethernet† to provide communication with other Ridge computers, as well as with computers manufactured by other vendors.

3. SOFTWARE OVERVIEW

As illustrated in Figure 1, the Ridge operating system (ROS) consists of a kernel and a set of server processes. The basic server processes are:

- User Monitor. Provides the UNIX interface to user processes.
- Directory Manager(s). Manage the directory structure for all named objects, including files and devices.
- Volume Manager(s). Manages disk space.
- Virtual Memory Manager. Implements memory mangement and high level scheduling policies.
- Network Server. Provides network access.
- File Manager(s). Provides file system services to individual files.

With this set of server processes, the ROS kernel needs to provide only the most basic support for memory management, interprocess communication, and interrupt handling. This greatly reduces the kernel size to only eight kilobytes of assembly language.

† Ethernet is a trademark of XEROX Corporation.

In traditional operating system implementations, a system call traps to the kernel. In ROS, a system call is an entry in a runtime library, which packages the request as a message and sends it to the user monitor. The user monitor performs the system service, either directly or indirectly through the use of other ROS server processes, and sends a reply message back to the user process. One user monitor is created for each terminal on the system, and provides most of the security checking for user requests. This allows the remainder of the operating system to be simpler, open, and free of protection requirements.

ROS emulates the UNIX hierarchical file system. Each UNIX file system is managed by a directory manager and a volume manager. The directory manager maintains the namespace for objects on that file system. The volume manager controls space allocation and maintains file control information, such as file owner, group, and reference count. These processes are both virtual and pageable. File name lookups are much faster than traditional UNIX systems because the opening of a file for each directory is avoided and a cache of recent directory entries is maintained by the virtual memory system. Applications that read the directory as a file are connected to the directory manager, which then behaves as a file manager by providing the relevant UNIX file system calls.

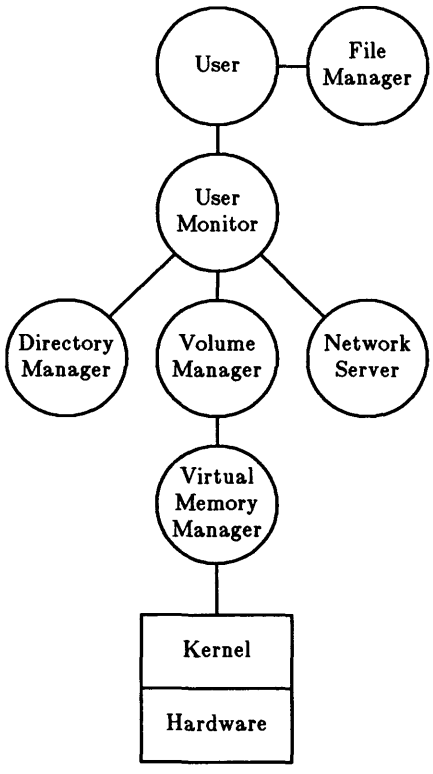

Figure 1. ROS Overview

Devices may also be entered in the directory structure, as in traditional UNIX. A directory entry for a device points to an associated device manager. If a device entry is encountered when parsing a pathname, the remainder of the pathname is passed to the device manager. This permits a manager to implement its own arbitrary naming scheme, which in turn permits extension to a distributed, heterogeneous environment.

A device manager operates together with the kernel to manage its device. The kernel guarantees a process exclusive access to the device by making the device manager the recipient of interrupts from the device. The kernel forwards interrupts in the form of messages to the device manager. Since device managers are implemented as processes, they may be initiated and dynamically bound to the device. This allows multiple device managers to exist for the same physical device, although only one device manager may have access to a device at a time. Thus, device managers are easy to develop, and do not require re-compilation or re-binding of the kernel.

When a data object is opened, the user monitor connects the user process to an appropriate I/O manager. This manager is either dynamically created or it is a static device manager. Once the connection is made, the user process performs I/O directly with the manager, bypassing the rest of the operating system.

Examples of static I/O managers are the device managers for the RS232 terminals, tape, and line printer. I/O managers are created dynamically for files and pipes. Each open file is managed by a separate file manager, whose data space is the contents of the file. Similarly, pipes are associated with pipe managers, whose data space is the contents of the pipe buffer.

Creating one process per open file may seem to be rather startling at first, but it does have its advantages. First, file manager processes place files into the process model in which all system services are provided by server processes. Second, by making a file the data space of a process, it was easy to create a combined virtual memory/file system.

The virtual memory manager is responsible for bringing pages from disk into main memory. Because virtual memory is combined with the file system, the virtual memory manager pages only files in from disk. Program code and data are files created and managed by the volume manager. For efficiency reasons, the disk driver is part of the virtual memory manager. Consequently, only the virtual memory manager has direct access to the disk and can allocate main memory. This allows the entire main memory to be treated as a disk cache and provides an opportunity for high file system throughput by utilizing the virtual memory manager to manage the file buffer cache.

UNIX applications operate in the ROS environment exactly as they operate in the UNIX environment. Porting them primarily consists of eliminating machine dependencies and recompiling them.

4. JUSTIFICATION

As stated previously, the idea was to provide two basic facilities -- messages and virtual memory -- make them as simple as possible, and build the operating system and user environment on top in a very "flat" manner. The motivation for this environment was to reduce the amount of "special" code to an absolute minimum, and to keep it at a minimum by discouraging the operating systems programmer from adding high level functionality into the kernel. Every effort was made to make sharing difficult or impossible, in the belief that pathological and time-dependent bugs come through unknown or poorly understood functions due to sharing. Although semaphores and monitors fix these problems, as long as shared memory is available, programmers may "forget" to use a semaphore and cause subtle bugs.

Ridge hardware was designed to complement this model. The processor has only two modes: user and kernel. User mode uses virtual addresses and is interruptable. Each user process has a separate virtual address space, which protects processes from each other. Shared memory is not permitted. (This did not work out too well in practice. Section 8 discusses this in detail.)

The kernel is intended for only the most basic of operating system functions. To simplify both the instruction set and the operating system model, interrupts are disabled in kernel mode. Therefore, only functions that are guaranteed to complete in a short interval may be placed in the kernel. The kernel code is non-interruptible in order to keep functionality in the kernel limited and to rule out asynchronous bugs. Usually the most difficult and seldom occurring bugs in an operating system lie deep within it, and frequently are created when some new feature violates timing or priority rules that have long since been forgotten. By making the kernel "synchronous", bugs should pop up early and be reproducible without having to wait for circumstances such as "fails under heavy load". Also, kernel mode uses only real addresses, making it difficult to peer into the user data space in performing operating system functions. Given these restrictions in the kernel, operating system functionality must nearly all be placed in user processes.

5. THE MESSAGE SYSTEM

The ROS message system was created from three guidelines:

Performance	Round trip message time should be below one millisecond. In order to decompose the operating system into multiple processes and to use multiple messages to perform a function, overhead due to messages and context switching must be kept at an acceptable level.
Protection	Since messages are the key component in the ROS architecture, it was decided to handle them within a protected address space, with access controlled by the message primitives.
Simplicity	The functions were to be few and simple in order to keep the size of code small and the number of bugs at a minimum.

The message system is described in detail according to the outline below:

a. Message primitives.
b. Data movement.
c. Context switching.
d. Trace execution of a round-trip message.
e. Hardware support and special instructions.
f. Performance.

The message system is connection-oriented. Processes send messages over a communication channel called a link, and messages are placed in a queue. Links, queues, and messages reside in a data space called a "queue segment", which can only by accessed by the kernel. Each process has its own queue segment, which is virtual, paged, and has a maximum size of four gigabytes. Making the queue segment virtual simplifies the message system, so that buffering and dynamic allocation of data structures are handled by virtual memory, rather than by the message system.

Message Primitives

When a process is created, its queue segment must be initialized by making the following kernel call:

```
ErrReturn := InitQSeg ( NumLinks, NumQueues,
        NumMessages, NumPages);
```

This establishes the maximum number of links and queues in the queue segment. Also set are the maximum number of messages for all links, *NumMessages*, and, for large amounts of data, the maximum number of pages, *NumPages*.

In order to receive messages, a queue must be opened:

```
ErrReturn := OpenQueue (NumMessages, MyQueue);
```

NumMessages specifies the maximum message count in the queue, and *MyQueue* is a descriptor used for receiving messages. The next step is to open a link to another process queue:

```
ErrReturn := OpenLink (ReceiverProcessId,
        ReceiverQueue, MyLink);
```

The process opening the link specifies the process identifier of the receiver, *ReceiverProcessId*, and the number of the queue, *ReceiverQueue* that will receive the messages. The *OpenLink* call returns *MyLink*, which is used as a descriptor when sending messages. The link number, *MyLink*, and the queue number (*MyQueue*, above) are both small integers, similar to UNIX file descriptors.

The processes are now ready to communicate. Data can be passed in both short and long messages. A message is sent by:

```
ErrReturn := Send (MyLink, Message);
```

This places a 32-byte message in the queue pointed to by *MyLink*. In addition to a short message, up to 4096 bytes (one page) can be sent with the following function:

```
ErrReturn := SendData ( MyLink, Message,
                Address, NumBytes);
```

Address is the location specifying *NumBytes* as a byte count of data to be sent. Both *Send* and *SendData* are asynchronous, which allow the sender to continue processing after the message has been queued, although most clients immediately wait for a reply from the server before continuing. Messages are received by:

```
ErrReturn := ReceiveData( MyQueue, Message,
                Address, NumBytes, SenderId);
```

The next *Message* is read from the queue *MyQueue* along with *NumBytes* of data. If there is no message, *ReceiveData* blocks and waits until a message is placed into the queue. *Address* is the location for received data. *SenderId* is the process identifier of the sender. If the message is a short message, this is indicated by an *ErrReturn* value.

The message system provides other primitives, which are only briefly mentioned here:

Test — Tests a queue for the presence of a message. *Test* is used in place of *ReceiveData* when blocking on an empty queue is not desired.

Arm and *Wait* — Processes may wait for multiple asynchronous events. Each asynchronous queue is *Arm*ed. The process then calls *Wait*. When a message is placed in an *Arm*ed queue, the process is awakened from the *Wait*. *Test* is then used to determine which queues have messages, and *ReceiveData* is used to receive messages from the queues. Prioritization of multiple asynchronous events is handled by making *Test*s in priority order.

Data Movement

Study of other message systems showed that short, fast messages were desirable and would be the most frequent. Passing messages in registers (as the message system does) helps to increase performance. Eight of the sixteen general registers are used to hold a 32-byte message, leaving eight registers for scratchpads, environment, and stackframe pointers.

Large amounts of data are passed through the message system in 4096-byte pages. Transfers longer than a page use multiple messages. Limiting the maximum amount of data to one page simplifies buffering by the kernel in the queue segment and in the receiving process. Receiving an arbitrary amount of data requires dynamic heap allocation, which reduces performance. DEMOS and the Stanford V system[5] solved this problem by introducing separate calls to handle data transfer. The dialogue proceeds something like:

```
Client: "I'm sending you 10,000 bytes of data".
Server: "I'll accept the data".
Message system moves data from client to server.
```

Although this allows an arbitrary amount of data to be transferred, the receiver needs to know how big a buffer is required. As a consequence, a synchronous send is used. Since it was desired to keep the ROS message system asynchronous, the data is transferred along with the message in the *Send* primitive. (However, measurements of the system have shown most messages to be synchronous. This decision is revisited in Section 8.)

The virtual memory system and message system work together to provide file system services. Each open file is accessed with a file manager process. The file data is passed one page at a time, using *SendData*. For example, on a file-read, the file manager calls *SendData* to pass the page of data at the current file position, through the message system, to the user process. The kernel checks to see if the page at that address is present in main memory and, finding it absent, locates it out on the disk. The kernel then generates a page fault for the file manager and sends a page fault message to the virtual memory manager. The disk page is then read-in by the virtual memory manager.

Context Switching

In order to improve message system performance, context switching and low-level scheduling are included in the message system. The kernel maintains the context and priority for each process in a process control block. The processes waiting to be executed are placed in descending order in a ready list. The kernel then dispatches the highest priority process on the ready list.

Processes with empty message queues are removed from the ready list. (Processes are also removed from the ready list on page faults.) Executing a *ReceiveData* on an empty queue suspends the process and removes it from the ready list. Sending the process a message places it again on the ready list; the process will become active when it becomes the highest priority process on the list.

Processes are generally scheduled as a result of sending and receiving messages, but there are two other scheduling circumstances. When an interrupt occurs, the kernel places an interrupt message in the interrupt queue of the driver process, which is then placed on ready list. If the driver has a higher priority, then the currently executing process is suspended and the driver is scheduled and dispatched. A timer interrupt causes time-slicing scheduling to occur every twenty milliseconds. If the currently executing process has had its time-slice, and another process of equal priority is on the ready list, the waiting process is dispatched.

A Round Trip Message

Let's follow a message from a user client process to a server process. The client loads the message into registers, then issues a *Send*. The kernel verifies that the specified link is valid, then checks the state of the server process queue. Note that both these checks are made on virtual data structures in the queue segment. Should any data be absent, the kernel sends a page fault message at the appropriate address in the queue segment to the virtual memory manager. After the page fault completes, the client continues executing *Send*.

Much of the traffic in the message system consists of higher priority servers waiting for messages. In order to improve message system performance, high-priority servers are given special treatment. If the server process is blocked when a message is put on its message queue, the kernel checks the server's priority. If the server has higher priority, the kernel suspends the client and starts the server running as if it had just executed a *ReceiveData*. The message is transferred directly in the registers without making any copies. In this way, a single entry into the kernel performs a *Send*, context switch, and a *ReceiveData*. If the server isn't waiting for a message, or does not have a higher priority, then the message is placed in the server's message queue and the client continues executing.

The server performs its function, then *Sends* a reply message to the waiting client. If the server has a higher priority than the client, the message is placed in the client's message queue. The client is then put onto the ready list and executed when it becomes the highest priority process.

After sending a reply message to the client, the server then does a *ReceiveData*. Since there is no message, the kernel suspends the server and removes it from the ready list. The kernel then starts the process at the head of the ready list, which in this case is the client. The client begins executing just after the *Send*, issues a *ReceiveData*, and locates the message. The round trip is completed.

Hardware Support

The Ridge operating system and processor were designed at the same time, allowing close cooperation at the software/hardware interface. After the main instruction set was defined, sixteen opcodes (out of 256 total) were set-aside for "privileged" instructions. These instructions were to be used for message system support, virtual-to-real translation and I/O instructions.

There are four instructions used by the message system:

	Approximate Execution Time in Microseconds (cycles)
Kernel Call	1.625 (13)
Resume User Mode	0.750 (6)
Load User State	12.875 (103)
Save User State	6.750 (54)

Creating the above special instructions does contradict the design principle of a reduced instruction set computer, but without them the performance objectives could not be met.

Kernel Call and Resume User Mode

All message system functions are placed in the kernel and accessed via the Kernel Call instruction Kernel Call enters the kernel at one of 256 possible entry points (so far there are about 80 kernel functions), switches from user to kernel mode, and saves the user program counter in a special register. When the kernel is finished executing, it uses the Resume User Mode to return to the user. This instruction reloads the program counter from the saved register, sets virtual (user) mode, and starts the user executing again at the instruction following the Kernel Call.

Save User State and Load User State

The Save User State and Load User State instructions are used to save and load the general purpose registers and environment registers. The Save User State is generally executed by the kernel upon entry to a function, and the Load User State is executed by the kernel at the exit of a function, which is immediately followed by a Resume User Mode instruction. Every effort was made to minimize the amount of environment information needed on context switch. The complete process state used by the hardware was reduced to nineteen words: sixteen words containing the registers, plus three control words.

The Load User State instruction reads three 32-bit words from main memory in addition to the registers. These are:

- program counter
- code and data segment number (the upper sixteen bits of the virtual address.
- traps word containing bits specifying the enabling or disabling of integer overflow, floating pointer overflow, and other arithmetic traps.

The Save User State instruction saves the program counter in addition to the general registers. The traps word need not be saved, as it is not modified when the process is running. The execution times quoted above for the instructions include moving all sixteen general purpose registers. These instructions specify which registers are to be affected, and along with a count, permit the kernel to save and restore fewer registers.

Message System Performance

A round trip message executes approximately 150 instructions and takes 190 microseconds. Table 1 shows the time spent in each function. Context switch time is estimated to be 30 microseconds. The time spent is roughly divided into three portions: 1/3 in the kernel instructions (including saving and restoring the user state), 1/3 translating virtual addresses to real data and moving the data (for a short 32-byte message), and 1/3 making decisions and context switching.

Round Trip Message		Time in Microseconds
Client -	Send to Higher Priority Process (Run Server)	60
Server -	Send to Lower Priority Process	50
Server -	Receive and Block, Queue Empty	40
Client -	Receive	40
		190

Table 1. Message System Performance

The above round trip time does not quite tell the whole story. The message system functions are embedded inside library code that is bound to the calling program. The procedure call overhead for the round trip is about 20 microseconds. The compiler loads the parameters for the function from memory, stores them on the stack, then makes the procedure call. The message function must then load the parameters from the stack into the registers again.

Making the message system functions built-in functions in the compilers would eliminate much of the overhead caused by storing and loading from the stack, but would limit the flexibility of changing message system functions.

Message system performance has proved to be quite satisfactory and has not proved to be the factor limiting performance in the operating system. Two areas in the message system, however, have been selected for improvement. The first area is hardware. Currently, the processor requires sequential flushing of a small code cache and virtual mapping table on context switch, which requires eight microseconds. New versions of the processor have eliminated this. The second area is to provide a synchronous interface that would reduce the number of kernel calls from four to two. Most message traffic is between servers and clients, and resemble procedure calls. The asynchronous capability of the message system is never used and, without the addition of two kernel calls, performance can be increased by about one third. These two improvements and careful recoding of the message system should decrease round trip message times to 100 microseconds.

6. SYSTEM PERFORMANCE

Table 2 compares ROS performance with UNIX on the VAX 11/780 on several key factors that affect system performance. Both systems were similarly configured, adequate memory was present and disk system speed is roughly equal. All programs were written in C.

Test Name	Ridge 32 4 Mbytes Memory Fujitsu Eagle Disk ROS 3.2	VAX 11/780 8 Mbytes Memory RA81 Disk UNIX 4.2bsd
Write 8MB	499KB/sec	142KB/sec
Read 8MB	611KB/sec	202KB/sec
Copy 8MB	303KB/sec	243KB/sec
Pipe 8MB	435KB/sec	243KB/sec
getpid	52K/sec	5K/sec
sbrk	62K/sec	2K/sec
creat	16/sec	30/sec
fork 8KB	13/sec	28/sec

Table 2. Performance Comparison

The write, read, and copy programs measure file system throughput. The write program creates a dummy file by repeatedly writing the same 8 Kbyte blocking of data until 8 Mbytes have been written. The read program does the opposite, reading all the data into the same 8 Kbyte array in memory. Although these programs are trivial, they do measure raw file system speed. The copy program is the UNIX cp program, making a copy of an 8 Mbyte file on a relatively empty disk. The ROS times are very good, and can be significantly improved by a synchronous message system. These programs illustrate how the message system moves data, and the relationship between virtual memory and the message system.

There are two special features of the system which help make the file system fast. The first is the use of hardware providing copy-on-write pages. If the data is page-aligned, rather than copy the 4096 bytes of data, the virtual memory page table entry pointing to the data is copied instead. Only when a store operation is attempted will the data be copied. In the case of a file system read, the data is never written, so the data is never copied. Performance measurements have shown that the use of copy-on-write boosts file system performance by approximately 20%.

The second feature which makes the file system fast is the use of contiguous disk allocation by the volume manager and read-ahead and write-behind by the virtual memory manager. During sequential disk activity, the virtual memory manager reads up to sixteen pages ahead. Since files are paged into memory by the virtual memory manager, during periods of intense file system activity, all of main memory can be treated as a file system cache. For example, if a new file is created and 1 Mbyte is written into it, it is unlikely that any of the file would be written to disk. When the file is closed, it is flushed from main memory and written to disk in sixteen-page transfers. This use of multiple page transfers provides a big performance improvement. The file system throughput without read-ahead, write-behind, or copy-on-write is approximately 250,000 bytes per second.

In regard to further improving file system performance, it should be noted that for a file system read (or write) the data is copied twice. (If the data is page-aligned, the page table entries are copied twice.) Eliminating the copy into the queue data page should provide another ten percent improvement.

Other test program results from Table 2 are summarized below:

Pipe — This program sends 8 Mbytes of data through a UNIX pipe. The ROS message system is used to create pipes.

getpid — The process id is returned to the calling process. In ROS this is a kernel call, and does not involve the message system. This demonstrates that entering the Ridge kernel is very inexpensive.

sbrk — This is the number of UNIX *sbrk* system calls that can be executed per second. *sbrk* dynamically allocates data. In typical UNIX systems, this is handled by going into the kernel and manipulating system tables. In ROS, no kernel call is made. Instead a run-time space allocator merely bumps the space pointer. Actual memory allocation is made when the data area is referenced and causes a page fault.

creat — Creates new disk files and closes them repeatedly. This measures how fast files can be opened and closed. ROS performance is discussed in Section 8.

fork — Measures the number of UNIX *fork* (clone a process) system calls that can be executed per second. The ROS performance is discussed in Section 8.

7. RELATED WORKS

DEMOS

While both ROS and DEMOS use communication channels, DEMOS is very much capability oriented. Unlike DEMOS, ROS makes no provisions for duplicating links, passing on links, or controlling access rights to links. It was felt that capabilities made the system more difficult to program. Send time has been estimated at 80 microseconds.

Guardian

The Tandem system uses multiple processors with independent memories. A message system was required to implement a multi-processor operating system. Guardian is significant in that it is one of the first commercially successful products to rely on messages. Performance figures from early systems indicate that a round trip message took one millisecond. The system is not a pure message system in that it relies on semaphores to implement performance critical functions that are local to a processor.

Elxsi EMBOS[11]

A multi-processor non-shared memory system derived from DEMOS. Like Ridge, Elxsi put message support in the architecture and instruction set. The message system is used for protection to the extent that there are no I/O instructions and I/O devices are sent messages. Half round-trip message time is 115 microseconds. Bulk data transfer bypasses the message system and uses mapped files instead for higher performance.

Accent[9]

A very flexible message system that sends and receives typed messages via ports. Ports can be independent from processes. The Accent kernel is monolithic and rather large, but integrates a large number of functions into the message system, including copy-or-write support for message passing.

Apollo Domain[1, 7]

Although a network operating system, its model is single-level store, rather than messages. Messages are used among system processes and hidden below single-level store. This is interesting in that ROS is the opposite. Messages are the model, while the virtual memory system implements single-level store underneath. Apollo has incorporated UNIX within its system, while keeping its Domain kernel as a base.

Stream I/O [10]

This is a change to the UNIX I/O system in order to make it more modular and extensible for networking. It is interesting to see UNIX modernized in message-like fashion. Ultra-lightweight processes pass data from module to module in streams. While increasing overhead on a per-character basis, overall system performance was not degraded.

8. CRITICAL REVIEW

On the whole, the design of the Ridge operating system has proved to be very satisfactory. The biggest fear was that the message system performance would be a problem, but this proved not to be the case. In evaluating and improving system performance, the major bottlenecks turned out to be elsewhere and message system overhead was not a factor.

The best feature of the system has been its reliability. Although the system has its share of bugs, there have been remarkably few load-dependent and sporadic failures. This is attributed to protected address spaces and servers that are the sole manipulators of system data structures. The primary file system processes (file managers, directory manager, and volume manager) are synchronous, accepting a request and processing it to completion before going on to the next request. The code for these servers tends to follow a main path, and either it works or it doesn't. Failures are reported in messages back to the requesters, which typically disengage themselves and terminate without damaging the system.

Incorporating external interrupts into the message system also tends to reduce failure. Interrupts are fielded by the kernel then sent in messages to driver processes. The drivers are written as server processes, which handle interrupt messages much like user requests. Enabling and disabling of interrupts and priorities is handled once by the kernel and is of no concern to the driver writer.

Drivers are usually written and debugged as user programs while running under the operating system. Once the driver is working, its path name is added to an initialization file that is read at boot time, which starts the driver and effectively makes it part of the system.

CRASHES AND FILE SYSTEM INTEGRITY

From the beginning, the operating system rarely crashed. When failure occurs, it is usually a failed user monitor, which results in a lost port into the system. Complete failure is caused by the virtual memory manager detecting an inconsistent data structure, which is then reported on the console and the program goes into a loop. These failures can be investigated by using the resident debugger, and sometimes can be repaired.

File system integrity has been extremely high. Only a few complete file systems (all the data on one disk) have been lost in two years counting over three hundred systems. Yet, the system has only one built-in fail-safe protection mechanism. Both the directory manager and volume manager process requests to completion, then flush their data spaces to disk at the end of each transaction. This leaves the file system in a consistent state at the end of file open and close and at the end of process creation and termination (due to creating process data spaces from files). Hence, the file system is only vulnerable to crashes resulting from physical disk errors (uncorrectable errors are extremely rare) or from bugs in the directory manager or volume manager. Simple algorithms and synchronous processing of requests have reduced bugs in these processes to a very low level.

Corruption of data within a file is also rare due to the restrictions in disk access. The volume manager and virtual memory manager are the only processes that directly access the disk. The volume manager controls disk allocation and accesses its tables as part of virtual memory. The only process that uses actual disk addresses is the virtual memory manager.

FUTURES

The initial goal of the operating system was to deliver the utmost in performance to a large dedicated application. This goal was met in that system memory residency is low (kernel is 8 Kbytes of code) and paging and file system throughput are high. Residency is kept low due to the operating system being virtual, pageable, and modular.

Unfortunately, the Ridge operating system is also used in a UNIX development environment which is characterized by executing many programs, most of them quite small. In this area the Ridge operating system's performance is noticeably below 4.2 bsd running on a VAX/780.

The main culprits in system performance are process fork and kill. The reasons for this lie in the virtual memory design and lack of shared memory. In the UNIX fork system call, the complete process space is duplicated, including open files. In the Ridge system this requires creation of a queue segment and opening each queue and link in the descendent that exists in the parent. Each server process that is connected to the parent must be notified of its new client and it must establish a link to the client. The server must also be notified when the client is killed. In the case of the shell, it opens input, output, and standard error files, causing three connection and disconnection messages to the terminal driver when it forks and kills a child process. Opening a link or a queue takes approximately 25 microseconds and, in itself, is not expensive. It is the processing time by the user monitor, clients, and servers that is expensive.

To reduce costs, the file system tables should be moved into the kernel. This will eliminate the need to send notification to servers. Placing file system tables in the kernel will result in sharing data among operating system processes and the kernel. This is a deviation from the original design strategy of no shared memory, but it seems worthwhile in view of the increased performance.

A second performance improvement will come from better data message primitives. At present, a copy-on-write must be broken into multiple four kilobyte *SendData*'s. Thus, a fork requires many trips through the kernel. This can be avoided with a synchronous send primitive, which would reduce the number of entries in the kernel from four to two. Passing an arbitrary amount of data in the messages will speed process forks, which must copy the parent's data space into the newly created child's data space.

Users of real-time systems have found the Ridge kernel attractive, and have used it for some real-time applications. Although not specifically designed for real-time, only small modifications will be needed to make the Ridge a good real-time system. Real-time systems require semantics for process creation, destruction, communication and control which are very similar to those offered by the Ridge kernel. None of the kernel operations require disk access, so times spent in the kernel are inherently predictable. Measurement has shown most kernel calls executing in times ranging from 20 to 120 microseconds. The only function that has unacceptable execution time is sending a single page of data by value, rather than copy-on-write. The planned fix for this is to make the kernel check for interrupts during a copy operation.

REFERENCES

1. Anonymous. *Domain System Software.* Apollo Computer, Inc., Domain/IX Product Brief about Unix, April 1985.

2. E. Basart and D. Folger. *Ridge 32 Architecture -- A RISC Variation.* Proceeding of the IEEE International Conference on Computer Design: VLSI in Computers, October 1983, pp. 315-318.

3. F. Baskett, J. H. Howard and J. T. Montague. *Task Communication in DEMOS.* Proceedings of the 6th Symposium on Operating System Principles, ACM, November, 1977, pp. 23-31. Published as Operating Systems Review 11(5).

4. J. F. Bartlett. A NonStop Kernel. *Proceedings of the Eighth Symposium on Operating System Principles,* ACM, December, 1981, pp. 22-29.

5. D. R. Cheriton and W. Zwaenepoel. *The Distributed V Kernel and Its Performance for Diskless Workstations.* Proceedings of the Ninth ACM Symposium on Operating System Principles, October 1983, pp. 129-140.

6. M. E. Houdek, F. G. Soltis, R. L. Hoffman. *IBM System/38 Support for Capability-Based Addressing.* Proceedings Eighth Symposium on Computer Architecture, May 1981, pp. 341-348.

7. P. J. Leach, et. al. *The Architecture of an Integrated Local Network.* IEEE Journal on Selected Areas in Communications, Vol. SAC-1, No. 5, November 1983, pp. 842-857

8. D. A. Patterson. *Reduced Instruction Set Computers.* Communications of the ACM, vol. 28, no. 1, January 1985.

9. R. F. Rashid and G. G. Robertson. *Accent: A communication oriented network operating system kernel.* Proceedings of the Eighth Symposium on Operating System Principles, ACM, December 1981, pp. 64-75.

10. D. M. Ritchie. *A Stream Input-Output System.* AT&T Bell Laboratories Technical Journal. Vol. 63, no. 8, October 1984.

11. R. Williams. *A message-based operating system.* Systems and Software, February, 1985, pp. 139-142