User Level IPC and Device Management in the Raven Kernel

D. Stuart Ritchie and Gerald W. Neufeld
{sritchie,neufeld}@cs.ubc.ca

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z2
Canada

Abstract

The increasing bandwidth of networks and storage devices in recent years has placed greater emphasis on the performance of low level operating system services. Data must be delivered between hardware devices and user applications in an efficient matter. Motivated by the need for low overhead operating system services, the Raven kernel utilizes user level implementation techniques to reduce kernel intervention for many common services. In particular, our user level send/receive/reply communication implementation generates no kernel interactions per iteration in the best case, and two kernel interactions in the worst case. In more general cases, we observe approximately one kernel interaction for every two send/receive/reply iterations. Device driver support is also done entirely at the user level reducing copy costs and context switching.

1 Introduction

The speed of network channels and storage devices has increased by an order of magnitude in recent years (10Mbps Ethernet to 100Mbps FDDI and 140Mbps ATM). This increased bandwidth places additional burden on the operating system to deliver data between hardware devices and user applications. In order to sustain such speeds, device drivers must be invoked with low latency and be able to communicate high volumes of data to and from applications. We believe that pure kernel mediated architectures such as Mach [ABB+86] and V [Che88], even optimized using continuations [DBRD91], involve significant overhead.

Motivated by the need for low overhead operating system services in high speed protocol processing applications, we have implemented a lightweight kernel for shared memory multiprocessors. Threads, IPC, and device management are implemented at the user level, while task and virtual memory management are implemented in the supervisor kernel. Existing systems such as URPC [BALL90], and the "continuous media" system [GA91] demonstrate the viability of this approach. Our current implementation is based on the Motorola MVME188 Hypermodule, a 25MHz quad-processor 88100 machine [Gro90].

Our goal with the Raven kernel is to provide a lightweight environment for multithreaded parallel applications. The applications that we are currently investigating are high-speed networking and disk activities. In an environment based on threaded parallelism and high device interrupt rates, a great deal of context switching is to be expected when running such applications. We have designed and implemented our system accordingly.

By moving several of the high-use kernel services into user space, less time is spent invoking operations. The general motivation is to reduce the overall number of user/kernel interactions. Several techniques are employed by Raven to do this:

- User level thread scheduling. Rather scheduling threads in the kernel, move the scheduling code into the user space.
- User level interrupt handling. Allow interrupt handlers to upcall directly into the user space. Device drivers can be implemented completely in user space, eliminating the costs of moving device data between the user and kernel.
- User level interprocess communication. By making extensive use of shared memory between client and server address spaces, data copying through the kernel is eliminated.
- Low level synchronization primitives. Provide a simple mechanism to allow an event to be passed from one address space to another. With appropriate hardware, remote processor interrupts can be implemented completely at the user level.

This paper discusses the design and performance of our user level IPC implementation and device driver management. We introduce the design of the overall system, and then discuss the IPC and device management facilities and how they interact. The paper follows with a performance evaluation of our current implementation.

2 Overall kernel design

The Raven kernel is small, lightweight microkernel operating system for shared memory multiprocessors consisting of a supervisor kernel and user level library. The supervisor kernel and user level library (Figure 1 compiles into 52K of executable code and 72KB of data. The size and scope of the Raven kernel is roughly similar to the QNX [Hil92] operating system: the supervisor kernel provides a simple set of abstractions, from which a set of more complex services may be constructed. Unlike QNX, however, the Raven kernel takes advantage of symmetric multiprocessing and user level design techniques.

Two main abstractions are provided by the supervisor kernel: tasks and virtual memory. Task creation, destruction, and scheduling (address space switching), are implemented inside the kernel. A task consists of an address space that is scheduled for execution by the kernel on one or more processors. The kernel also maintains strict control over page tables and free memory lists, so all virtual memory allocations and mapping are provided by system calls.

All other services are provided by the user level: threads, semaphore synchronization, interprocess communication, and device management. These features are accessible to application programs via inline macros or procedure calls, rather than more expensive kernel traps. Threads are preemptively scheduled from processor to processor in an effort to balance work load. Semaphores can be used to coordinate threads in local and remote address spaces. Extensive use of shared memory allows disjoint address spaces to efficiently communicate scheduling information and interprocess communication data. Figure 1 shows the supervisor kernel in relation with the user level library.

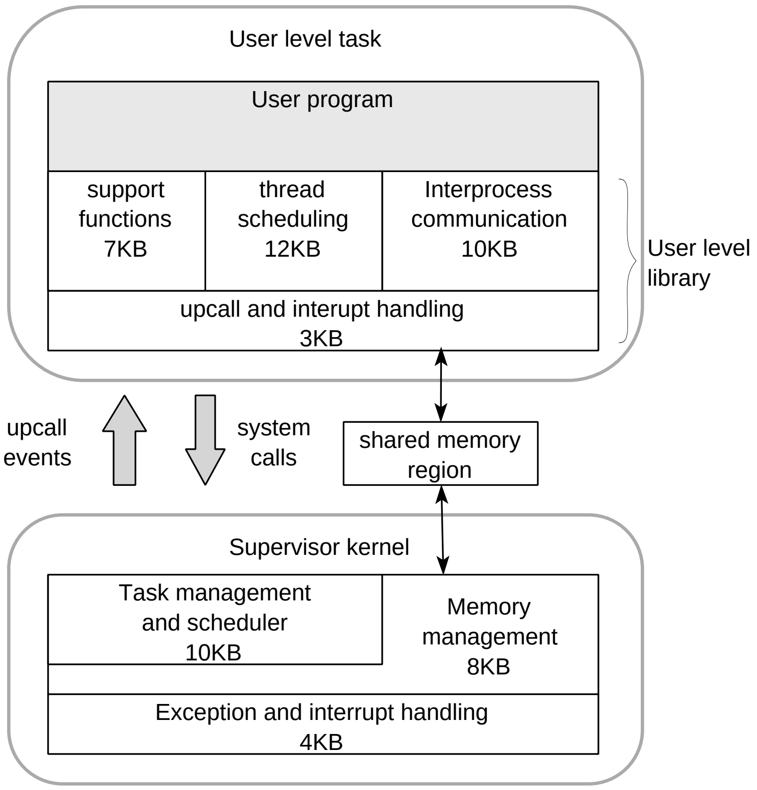

Figure 1: High level system organization.

The user level thread scheduler communicates and coordinates scheduling activities with the supervisor kernel using a per-task shared data structure and system calls. The shared data structure maintains several fields which allow the user level scheduler and supervisor kernel to make scheduling decisions without crossing user/kernel address space boundaries. Under ideal conditions, threads can migrate amongst processors without kernel intervention. However, certain cases exist where kernel assistance is required. One such case occurs when an address space switch is required.

The MVME188 hardware provides a simple and efficient means to deliver interrupts to remote processors in the system. We use this feature throughout the system to aid in the thread and task scheduling decisions required for IPC interactions and thread/task management. For example, when a new thread is created, an interrupt is delivered to the next available idle processor to run that thread. Similarly, invoking the IPC mechanism to send a message results in an interrupt delivered to the processor that is currently executing the desired destination task. Idle processors never scan lists searching for work to do – work is delivered to idle processors in the form of interrupt notifications.

Built on top of this interrupt mechanism is an asynchronous task signalling facility. The task signalling facility provides a mechanism to asynchronously send signal messages from one task to another. This software implementation is roughly similar to Cheriton's hardware work with "address-valued signals" [CK93]. A signal message is a simple two word structure which identifies the signal type and message data. Each user level scheduler maintains a FIFO queue to store signal messages, and implements a set of signal handlers, one for each type. Signal handlers are invoked on the receipt of each signal message. Task signals are the the basis for IPC and semaphore operation throughout the kernel.

A task is delivered a signal by a non-blocking user level library function, `task_signal()`. This function avoids system calls on the local processor by using the remote software interrupt facility to notify destination tasks of the arrival of signal messages. There are three possible ways that signals are delivered:

1. If the destination task is currently executing on a remote processor, an interrupt is issued to that processor.
2. If the destination task is not executing on a remote processor, then issue an interrupt to an idle processor.
3. If there are no idle processors and the destination task is not active, perform a system call to the local processor to initiate a task switch.

Only the last item (3) produces an explicit system call by the local processor. The other items avoid a system call locally, allowing the processor to proceed with its own work. Work is distributed to remote processors in this fashion.

3 Interprocess communication

The user level IPC library provides a port-based synchronous send/receive/reply[1] interface as well as asynchronous send/receive. Both of these interfaces utilize user level shared memory and the task signalling facility described above to reduce the frequency of kernel interactions. Systems such as Mach and Chorus, extend their IPC models across the network. We only consider the inter-machine case.

The port-based approach to interprocess communication uses a port number as the mailbox address for reliable message delivery. Rather than implementing send/receive/reply in terms of two send/receive ports, we have implemented the two models separately. There are several reasons for doing this, one being performance.

Figure 2 shows a high level view of how the main IPC data structures are organized. Two port descriptor tables are allocated by the initial task at boot time, one maintaining entries for synchronous ports, the other maintaining entries for asynchronous ports. The tables are shared amongst all user level tasks in the system. A port descriptor contains information pertaining to the state of the port queue, and a set of semaphores to coordinate client and server operations. At port creation time, the IPC library searches the appropriate table for a free entry and allocates an associated FIFO message queue. The number of messages in this queue and their size is specified at port creation time. The send/receive/reply implementation uses the same queue for storing both send and reply messages – a second queue is not required. After a port is successfully created, the resulting descriptor index is used throughout the system as a port identifier.

The port's message queue is shared in a region of virtual memory between the server and each of the clients involved in the communication. Clients wishing to communicate to a server over a port must initially "establish a connection" to the server by calling a library function that performs the message queue mapping and registration functions.

[1]Our send/receive/reply implementation permits the reply stage to be deferred until a later time, out of sequence with the arrival of messages similar to the V System[Che88].

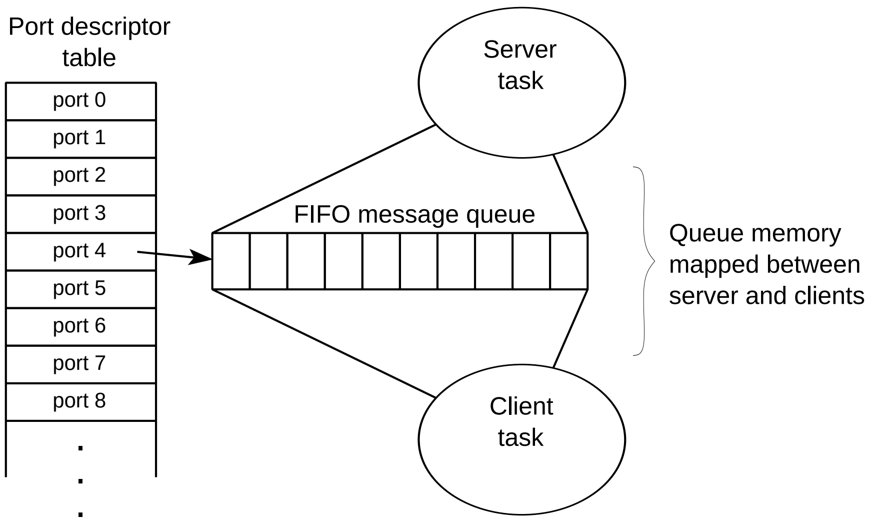

Figure 2: Port descriptor table and message queue mapped into a client and server.

3.1 IPC algorithm

The work performed by an interprocess communication facility can be divided into two parts. The first part manages the movement of message data between the sender and receiver, usually by means of a message queue. Traditional kernel based systems maintain message queues inside the kernel: message data must be marshaled in and out of the kernel on each invocation. Raven's exposure of message buffers to user space allows user code to directly marshal message data. This can save a data copy operation on the receiver side because the receiver has direct access to the message buffers. A second “user level” copy need not be made.

Once message data is queued for delivery, the second part of an IPC transaction involves notification to the recipient that a message has arrived. The notification step passes control to tasks where blocked recipient threads wait, often involving a processor allocation decision (a client task may be switched out so that a server task can run). For example, when a client thread sends a message to a server, the server must be notified. In traditional kernel based systems, this notification step normally requires kernel support. In the Raven kernel, the task signalling mechanism is used as the low level notification system. As noted above, this mechanism avoids kernel intervention in two out of three cases.

Synchronous send/receive/reply transactions are inherently more complex than asynchronous send/receive. The synchronous case requires the client senders to block and wait for a reply from the servers. In the best case, no notifications are required for either synchronous or asynchronous IPC. In the worst case, two notification steps are required for synchronous IPC: the client notifies the server that a message awaits, and the server notifies the client that a reply awaits. For the asynchronous case, only one notification is required: the client notifies the server that a message awaits.

In best case, the IPC library can use properties of the FIFO message queues and multiprocessing to avoid the notification step, and ultimately avoid kernel intervention altogether. Looking at the simpler asynchronous case, we can see how this is accomplished:

1. If a server thread is not currently blocked waiting for a message when a client performs a send, no notification by the client is necessary. When the server thread eventually performs a receive operation, it will notice a message waiting and immediately pull it from the queue without blocking.

2. A non-empty message queue indicates to the client that the server has already been notified that a message awaits. Thus, notification to a server only occurs when a server thread is blocked and does not already have messages waiting for it.

3. However, if a message queue is full, and cannot accept additional client messages, the sending client must block. As the server eventually empties the message queue, it must send notification to the blocked clients indicating that more room exists in the queue.

The use of these properties are summarized by the following algorithms used to implement the asynchronous send and receive primitives. We first examine the send primitive. The send primitive generates at most one notification per invocation:

1. If there are no free message buffers available, block on a a free-list semaphore.

2. Claim the next free message buffer and copy in the message.

3. Deliver a notification signal to the destination server task only if the send queue was previously empty and a server thread is blocked. Otherwise, do not send a notification signal.

The receive primitive generates at most one notification per invocation also:

1. Block the calling thread if there are no messages waiting. Continue otherwise.

2. Dequeue the next message and copy it into the calling thread's buffer.

3. Return the message buffer to the queue and increment the free-list semaphore. If a client is blocked waiting for a free message buffer, deliver a notification signal to the client, indicating that a free message buffer exists.

As noted above, the send/receive/reply mechanism is more complex than send/receive because the sender must always block and wait for a reply. This can result in two notification signals per interaction in the worst case. However, as seen in the asynchronous send analysis above, it is possible to eliminate notification signals. We can apply the same principles to both the send stage and reply stage. This allows send/receive/reply to operate without notification (and therefore without kernel intervention) in the best case. The algorithms for the send/receive/reply primitives are summarized below.

The send operation requires at most one system call per invocation, or possibly a task signal invocation:

1. If there are no free message buffers available, block on a a free-list semaphore.

2. Copy the user's message into the buffer.

3. If the send queue is empty, and a server thread is blocked waiting for a message, deliver a notification signal to the destination server indicating that a message awaits.

4. Block the sending thread and wait for a reply.
5. Wake up when the reply comes and copy the reply message to the user's buffer.
6. Return the message buffer to the queue and increment the free-list semaphore. If a client is blocked waiting for a free message buffer, deliver a notification signal to the client, indicating that a free message buffer exists.

The receive operation does not require any kernel interaction at all.

1. If a message awaits in the port queue, return a pointer to the received message buffer to the user.
2. Otherwise, if the port queue is empty, block the calling thread and wait for a message.
3. Wake up when a message arrives and return a pointer to the message buffer to the user.

The reply operation requires a notification signal only when there are no reply messages queued for the client task. This case is analogous to the sender's situation, except that here a reply is being delivered:

1. Enqueue the reply message.
2. If there are existing reply messages in the queue for the client, simply return.
3. Otherwise, deliver a notification signal to the client to indicate that a reply awaits.

It should be noted that all access a FIFO queues and port descriptors require synchronization. This is accomplished using spin-locks. Spin-locks are efficiently implemented by utilizing the Motorola 88200 cache coherency mechanism. No kernel intervention is required to implement the locks.

4 Device driver management

Hardware device drivers are implemented completely in user space. User level applications register hardware device handlers through the kernel interrupt dispatch mechanism and map in the device registers to their address space. When a device interrupt occurs, an upcall [Cla85] is issued to the task which contains a handler for the interrupt. The handler is executed to satisfy the device and processing resumes.

Each interrupt generated by a device causes an upcall into user space. On the surface, the additional cost of traversing from kernel to user space compared to a pure kernel interrupt handler would appear prohibitive. However, we observe the following advantages with this technique:

- There is no need to copy or map data between the kernel and user level.
- Execution can continue in user space after processing the interrupt. A kernel level handler eventually requires an upcall into user space to allow applications to process data.

- Device drivers can be dynamically loaded and unloaded without the complexity of dynamic linking.
- Device drivers can be implemented directly in the application that uses the device, reducing communication costs and latency.

4.1 Interrupt management

The MVME188 interrupt management hardware is fully symmetric. Any processor in the system can respond to any particular interrupt by setting appropriate bits in the processor's interrupt enable register. In our system with four processors, there are four interrupt enable registers. The kernel manages these registers in an effort to position device interrupts to minimize invocation latency of user level interrupt handlers. The interrupt enable bits follow the migration of their associated application programs.

For example, the Ethernet interrupt handler function is implemented within a user level task. An upcall event is dispatched into this task to execute the handler whenever the Ethernet interrupt occurs. If the Ethernet task address space is not enabled on the processor where the interrupt occurs, the task must be switched in. If the interrupted processor is currently running a different task, then it must be switched out before the Ethernet task can be switched in. This sequence of steps involve address space changes and data structure manipulation that greatly increases interrupt handler latency. The kernel attempts to avoid this situation by positioning interrupt enable bits on processors that are currently executing the associated task.

This interrupt management scheme also allows devices to operate in parallel. For example, one processor can enable its Ethernet bit and another processor can enable its disk or serial port bit. Figure 3 demonstrates this distribution of interrupt handling chores. Processor 1 is currently running the file system server, and thus has the SCSI disk interrupt set. Processor 2 and 3 are running the TCP networking software, and thus are sharing the serial port and Ethernet interrupt.

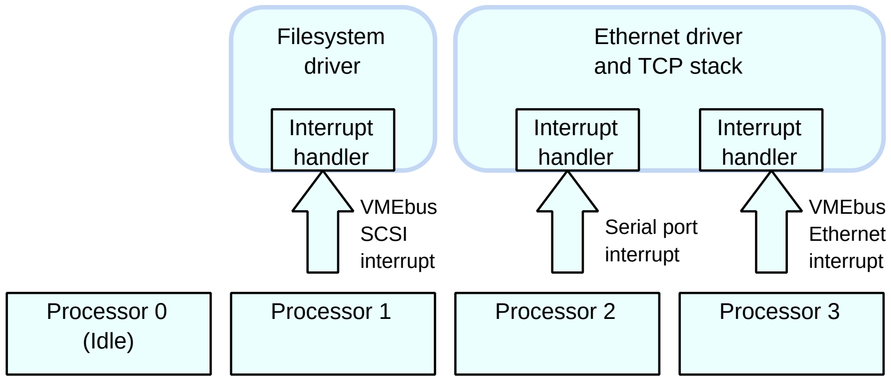

Figure 3: Interrupt management across three busy processors.

4.2 User level preemption

Interrupts preempt user level threads and cause scheduling events to occur. For example, if a timer interrupt occurs, an upcall event is sent to the user level to indicate that it's time to schedule the next ready thread. In a multiprocessor environment, special care must be taken to ensure that the rescheduling of threads in the presence of spin-locks does not adversely affect performance. A naive approach would allow interrupts to occur at any point during user level execution. This can result in very poor performance if threads are using spin-locks for concurrency protection. If a thread is preempted while holding a spin lock, then all other threads that try to access the lock must wait until the original thread is rescheduled and releases its lock. The original thread may not be rescheduled for some time, causing all other threads to waste CPU time, uselessly spinning.

The solution implemented in the Raven kernel involves close participation between the user level and kernel. Whenever the user level acquires a spin lock, a global `lock_count` variable is incremented. Whenever an interrupt occurs that would cause an upcall event into user space, the interrupt handler checks the `lock_count` variable. A non-zero value indicates that a critical section is currently being executed, and control must be returned to the critical section. Before the interrupt handler restores user registers and returns control to the critical section, it sets the `upcall_pending` variable to indicate that an interrupt occurred. When the user level regains control and finishes its critical section, the `upcall_pending` variable is checked, and if set, the thread will save its context and handle the original reason for preemption.

As implemented on the 88100, the two variables `lock_count` and `upcall_pending` are not stored as conventional variables at all. Rather, each of them share the processor's `r28` register. This is done to ensure that access to the variables is atomic. For example, to increment `lock_count`, a single `addu r28,r28,1` instruction is performed. If `lock_count` were a conventional variable, then incrementing it would require the use of a spin lock – which would be recursive, since `lock_count` itself is used within the locking routines.

4.3 Dispatching interrupts

Dispatching interrupts in the Raven kernel is a two-level process. Interrupt dispatchers are implemented at both the supervisor and user level. When a device interrupt occurs on a processor, the register context is saved into the user's thread control block, and the supervisor kernel interrupt dispatcher is called to begin processing the interrupt. If the interrupt is intended for a user level handler, an upcall is generated into the appropriate task where the handler is implemented, and the user level dispatcher takes over. The user level dispatcher calls the appropriate handler. Figure 4 shows the interrupt execution path from the supervisor dispatcher to the interrupt handlers.

The supervisor interrupt dispatcher maintains a table of all the possible interrupt sources and the location of the handler functions. Some handler functions are implemented directly in the kernel, such as the system clock tick. To handle a kernel level interrupt service routine, the dispatcher simply makes a function call to the service routine.

For user level handlers, the procedure is more complicated. The supervisor dispatcher checks the appropriate interrupt entry, and if the currently executing task is registered to handle that interrupt, an interrupt upcall event is delivered to the task. The particular interrupt vector bit that caused the interrupt is passed with the upcall event so the user

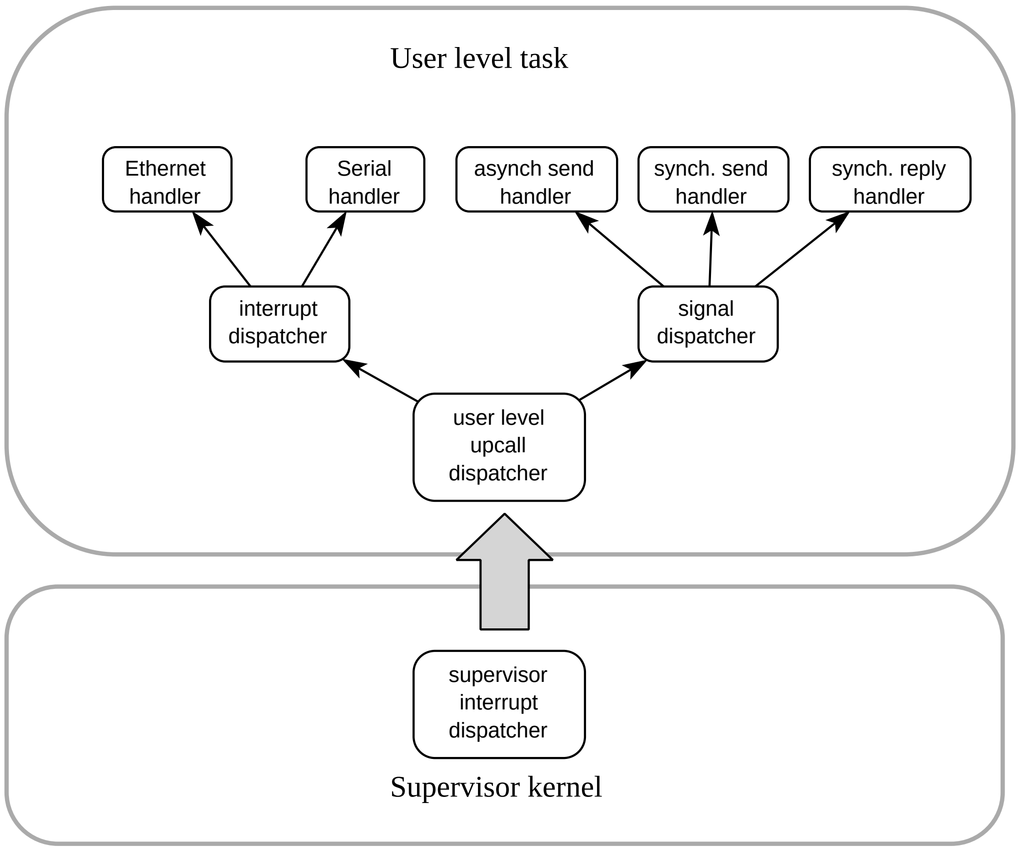

Figure 4: Interrupt execution path from supervisor dispatcher to the user level, including two interrupt handlers and IPC signal handlers.

level dispatcher can identify the proper handler.

If the currently executing task does not handle the interrupt, but another task does, then the current task must be switched out and the interrupt handling task must be switched in. Before the current task is placed back onto the task ready queue, the thread which was interrupted must be cleaned up so that it can be rescheduled. While the register context for the thread has been properly saved by the initial interrupt trap function, the user level thread kernel must be notified that one of its threads has been preempted, so the thread can be placed back on the user level thread ready queue. To do this, the current task is issued an upcall event so that the thread scheduler can place the interrupted thread back onto the thread ready queue. The thread scheduler then immediately returns to the kernel, where the interrupt handling task is finally activated.

4.4 Handler functions and device access

Before an interrupt handler can be invoked, the device register set must be mapped into the application address space and the handler function must be registered with the kernel. This is done during the initialization phase of the device driver. When the device driver terminates, the handler function must be removed and the device register set must be unmapped from the application.

The virtual memory module exports a function to the user level called `vm_map_device()` which allows applications to map device registers into user space. At initialization time, the device driver supplies the physical address and size of the device registers, and a virtual address is returned pointing to the mapped device registers. The function `vm_unmap_device()` allows the device driver to later remove the mapped memory from its address space.

A user level library routine manages the registration of interrupt handlers for device drivers. The caller specifies the function address and the associated interrupt vector. The registration routine then performs a system call into the kernel to notify the supervisor dispatcher of the new interrupt handler, and the appropriate interrupt enable bit for specified device is set. Once this is done, interrupts occurring on the device will be vectored to the user level handler.

User level interrupt handlers are executed within a controlled environment. Preemption is disabled during this time, so the handler is guaranteed uninterrupted access to the device registers. However, handler must not spend more time than is necessary to service the interrupt, or time will be taken away from other system activities. Additionally, the handler must be careful not to execute any library functions that perform thread or task context switching.

Interrupt handler functions typically coordinate their events with user threads using a semaphore and a shared data structure. When an event occurs in the handler that must be communicated to a thread, the event is recorded and the semaphore is signalled. A thread blocked on this semaphore will awake and be able to examine the event information. When a handler function is finished accessing the device, it returns to the dispatch routine where the thread scheduler takes over.

5 Performance

This section presents some simple benchmark results to demonstrate our implementation. We have constructed several test cases that exercise the interprocess communication primitives and interrupt handling features. To summarize:

- A single send/receive/reply interaction occurring between two tasks with no kernel interactions can complete in 42 microseconds. This represents the best case scenario.
- Average send/receive/reply times between several communicating threads over a period of time is measured at 90 microseconds. This represents approximately one kernel interaction per transaction.
- Worst case send/receive/reply of 145 microseconds is achieved by limiting two communicating threads to a single processor.
- User level interrupt handler invocation latency is 14.0 microseconds.

5.1 Interprocess communication

This section measures the performance throughput of the IPC library. The benchmark combines many of the primitive system services: remote interrupt dispatching, task signal notification, semaphores, and task and thread scheduling.

	1 CPU	2 CPU	3 CPU	4 CPU
1-send/1-recv/reply	145 usec	105 usec	105 usec	105 usec
2-send/2-recv/reply	108	95.3	90.3	89.6
10-send/10-recv/reply	89.3	92.5	94.9	95.6

Table 1: IPC performance for synchronous ports, 4 byte data message, 20 element message queue.

The global IPC test cases create two address spaces: a server task, and a client task. The server task allocates a port with a queue containing 20 message buffers. A number of server threads are created to listen for messages on the port. The client task creates a number of client threads and bombards the server with messages.

5.1.1 Synchronous IPC performance

Table 1 contains performance results for synchronous IPC using various combinations of processors and threads. The simple case of 1-send thread and 1-receive thread on a single processor demonstrates the worst case performance of 145 microseconds per interaction. Each iteration requires two address space changes. This performance is improved upon by the addition of multiple processors. In the two processor case, the client and server reside on different processors and can therefore avoid address space switching. Instead, notification signals are delivered via the remote interrupt mechanism. Additional processors do not help improve this case because there is no parallel computation involved.

Synchronous IPC performance increases slightly when more client and server threads are added. In the uniprocessor case, this is because clients and servers can copy messages in and out of the queues at once without switching tasks. For example, the 10 client threads can each copy their messages into the send queue before the task switches. Likewise, the 10 server threads can stack up their reply messages. This has the effect of reducing the overall number of task switches per IPC interaction.

An interesting case appears when 10 client threads and 10 server threads communicate using multiple processors. The IPC interactions actually get slower. We believe that this is a combination of spin-lock contention and poor scheduling decisions by the task and thread schedulers. Rather than balance the client and server threads on an even number of processors, the schedulers position tasks and threads naively. Thus a great deal of task switching results as client and server threads rapidly migrate amongst processors. A better scheduler might be able to recognize communicating tasks and position them on processors accordingly.

The figures in Table 1 are averages over large number of iterations. We modified the dual processor test case above to measure the quickest and slowest send/receive/reply transaction out of all these iterations as 42 microseconds and 120 microseconds, respectively. We believe this discrepancy to be caused by the overlapping of execution within the IPC primitives. The quickest iteration occurs when the client sends a message exactly at the same time as the server enters the receive primitive.

	1 CPU	2 CPU	3 CPU	4 CPU
1-send/1-recv	43.2 usec	33.9 usec	32.1 usec	32.0 usec
2-send/2-recv	44.5	32.2	31.8	31.0
10-send/10-recv	44.3	33.9	32.0	32.5

Table 2: IPC performance for asynchronous ports, 4 byte data message, 20 element message queue.

5.1.2 Asynchronous IPC performance

Table 2 summarizes the results for several asynchronous IPC test cases. The asynchronous message transfers are much faster overall because of the reduced context switching requirements between the client and server. Client threads have no problem keeping the port queue full of data for the server threads. A client can simply loop forever, assuming the server keeps up.

Increasing the number of threads results in a slight performance hit. We believe this is due to the increased number of thread descriptors being managed by the thread scheduler. The thread scheduler is very simplistic, and may position threads on processors in a non-optimal fashion. Also, increasing the number of threads increases the number of stacks and data structures to manage while context switching, possibly affecting cache performance.

As seen in the synchronous case, performance improvements decline as more processors are added to the system. We believe the reason for this is spin-lock contention. The workload performed by the client and server threads is null, so all their effort is spent trying to access shared data structures such as port descriptors and queue. If the client and servers performed some amount of work, as in a real application, most of their time would be spent outside of the IPC primitives, leaving less opportunity for lock contention.

5.2 Interrupt handling performance

The performance of interrupt handling is a critical concern for high speed device drivers and scheduling performance. Interrupt handling must be as lightweight as possible to ensure low latency dispatch times to device drivers. Interrupt handling and dispatching in a monolithic kernel is fairly straightforward: trap the interrupt, save context, and call the interrupt service routine. In the Raven kernel, since device drivers are implemented at the user level, device interrupts must take the journey up into the user level for processing. However, once at the user level, execution can continue with application processing.

An experiment was constructed to measure the execution latency time to dispatch an interrupt to a handler routine. Three different interrupt handler scenarios were measured:

1. A supervisor kernel interrupt handler. Invoking this handler requires a local function call from the supervisor dispatcher.

2. A user level interrupt handler in a task that is activated on the interrupted processor. Invoking this handler requires an interrupt upcall event to be dispatched into the user space.

	Time (usec)	Instructions
kernel invoke	7.21	86
user invoke	14.0	194
user switch/invoke	30.6	421

Table 3: Interrupt service routine invocation latencies.

3. A user level interrupt handler in a task that is not currently activated on the interrupted processor. Invoking this handler requires that the current task be switched out and the interrupt handler task be switched in. Then an interrupt upcall event can finally be dispatched into the interrupt handler task.

Table 3 summarizes the average times, in microseconds, to invoke each service routine. Also, the number of instructions per invocation is shown. The cheapest invocation time of 7.21 microseconds is naturally inside the supervisor kernel. No upcall into user space is required. Also, the user register context can be saved in a cheaper fashion, since it will be directly restored by the supervisor kernel at the end of the interrupt.

Invoking a user space handler is about twice as expensive. The user level register context must be properly saved into the thread's context save area, and an interrupt event must be upcalled to the user level. Once at the user level, the upcall dispatcher must place the previously executing thread on the ready queue, and finally call the handler routine.

Switching address spaces before calling the service routine is the most expensive invocation operation. The old address space must be upcalled to handle any cleanup and placed on the task ready queue before the new address space can be invoked.

At first glance, this benchmark appears to show that user level device drivers are much more expensive than kernel device drivers because of the interrupt dispatching overhead. However, one must also consider that even a kernel device driver needs to communicate with the user level at some point. User level code must eventually be executed to operate on the data provided by the device driver. Depending on the device, this may involve an extra data copy operation to move the data between the user application and kernel device driver. Moreover, there is the additional costs of scheduling and activating the user application when the device driver is ready for more. All of these costs are automatically taken care of by the interrupt dispatcher and upcall mechanism.

6 Conclusion

We have implemented a lightweight multiprocessor microkernel to investigate the benefits of parallel processing in multithreaded applications. This system implements many traditional kernel level abstractions at the user level to decrease the overhead involved in kernel interactions. The system is currently being used in developing a high-performance file server connected to an ATM local area network[NIG+93]

Work is continuing to improve the performance of the Raven kernel and add functionality. One area where performance could especially be improved is the thread and task schedulers. While the current simple and efficient implementation is beneficial for rapid context switch times, it suffers from rapid thread migration in certain cases. This behaviour can result in poor utilization of per-processor caches.

Source code is freely available from the authors upon request.

References

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer Conference Proceedings.* USENIX Association, 1986.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. Tr-90-05-07, University of Washington, July 1990.

[Che88] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CK93] David R. Cheriton and Robert A. Kutter. Optimizing memory-based messaging for scalable shared memory multiprocessor architectures. Technical report, Computer Science Department, Stanford University, 1993.

[Cla85] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 171–180, December 1985.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. 13th SOSP.*, 1991.

[GA91] Ramesh Govindan and David P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. 13th SOSP.*, pages 68–80, Asilomar, Pacific Grove, CA, 13 Oct. 1991. Published as ACM. SIGOPS.

[Gro90] Motorola Computer Group. *MVME188 VMEmodule RISC Microcomputer User's Manual.* Motorola, 1990.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *The Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, April 1992.

[NIG+93] G. Neufeld, M. Ito, M. Goldberg, M. McCutcheon, and S. Ritchie. A parallel host interface for an ATM network. *IEEE Network Magazine*, 1993.