The Raven Kernel: a Microkernel for Shared Memory Multiprocessors [1]

Duncan Stuart Ritchie

sritchie@cs.ubc.ca

Department of Computer Science

University of British Columbia

Vancouver, B.C., Canada V6T 1Z2

Technical Report TR 93-36

April 30, 1993

The Raven kernel is a small, lightweight operating system for shared memory multiprocessors. Raven is characterized by its movement of several traditional kernel abstractions into user space. The kernel itself implements tasks, virtual memory management, and low level exception dispatching. All thread management, device drivers, and message passing functions are implemented completely in user space. This movement of typical kernel-level abstractions into user space can drastically reduce the overall number of user/kernel interactions for fine-grained parallel applications.

[1]This work was made possible by grants from the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the goals of a multitasking operating system is to present the illusion of parallelism to users. To the naked eye, programs running in such an environment appear to run concurrently. At a conceptual level, the system can use the notion of threads and processes to present a parallel model for constructing programs. On uniprocessor hardware, the parallelism seen in these two cases is indeed illusion. It is the relative speed of microprocessors and operating system design techniques such as time-slicing and cooperative scheduling that make this illusion possible.

The recent proliferation of low-cost multiprocessor hardware has made it possible to change this illusion into reality. The parallelism that a user or application developer sees is no longer an illusion – it is real. With multiple processors, separate threads of execution can execute concurrently. While the amount of true parallelism is bounded by the number of processors in the system, the illusion of a general purpose parallel environment can be enhanced by using uniprocessor techniques such as time-slicing and cooperative scheduling.

However, while scheduling techniques can be borrowed from uniprocessors and applied to multiprocessors, their implementation cannot. Traditional uniprocessor operating systems often lack desirable features that are possible with parallel hardware. Also, they can be difficult to adapt to a multiprocessor environment, leading to inefficient operation [JAG86]. There are two main factors which contribute to inefficient operation: the increased requirement for concurrency control within the kernel; and new scheduling possibilities.

This report presents a new operating system kernel which addresses the above factors for a shared memory multiprocessing environment. This design is geared specifically towards uniformly shared memory architectures, and not non-uniform architectures (NUMA) or distributed memory architectures.

This report is organized as follows. The remainder of this introduction chapter introduces the reader to motivation of this work, and describes the overall design of the kernel. Chapter 2 describes the hardware and software run-time environments. Chapter 3 and 4 discuss the implementation of the supervisor kernel and user level kernel, respectively. Chapter 5 presents performance figures for the various kernel services. Chapter 6 provides a look at previous work that influenced the design of the system. Finally, chapter 7 concludes the report with a summary of future work.

1.1 Motivation

The availability of low cost multiprocessor hardware has opened up new avenues for providing higher performance programming environments for applications. However, the availability of general purpose operating system software to take advantage of new techniques is slow to appear.

Traditional microkernel architectures, such as Mach 3.0 and Chorus, are designed with the view that the kernel should only provide a minimum set of primitive abstractions (tasks, threads, virtual memory, device management, etc). These primitives are then used by user space modules to provide additional services to the operating system environment, such as networking protocols and file servers.

When used as general purpose computing environments, this architecture is sufficient. Robustness is the most important requirement, with performance being second on the list. The system is secure against malicious or errant user programs, and performance is adequate.

Recently, however, the speed of network and other input/output devices has increased by an order of magnitude or more. The traditional kernel mediated operating system now exposes its performance problems more than ever. For many applications running under such an environment, the cost of crossing user/kernel boundaries for each primitive abstraction becomes a concern. Typical system call overhead is likely 10 times greater than a procedure call [ALBL91].

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 can be employed 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 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.

1.2 Raven kernel overview

The Raven kernel is a small, lightweight microkernel operating system for shared memory multiprocessors. The kernel executable compiles into less than 32KB of code. The most intriguing part of the Raven kernel design is the dislocation of many traditional kernel services into user space.

Two main abstractions are provided by the kernel: tasks and virtual memory. All other services are provided by the user level: threads, semaphore synchronization, interprocess communication, and device management[1]. Extensive use of shared memory allows disjoint address spaces to efficiently communicate scheduling information and interprocess communication data. Figure 1.1 demonstrates the supervisor kernel in relation with the user level.

[1]The supervisor kernel initially handles interrupts, but they can be dispatched to the user level.

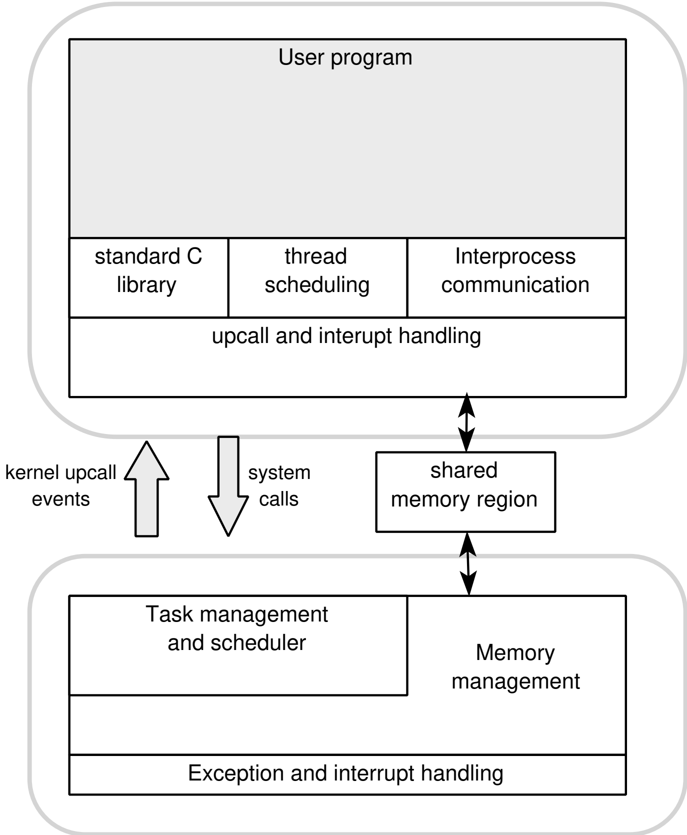

Figure 1.1: High level system organization.

The system is implemented for a four processor, Motorola 88100 shared memory multiprocessor [Gro90]. A special gdb-based kernel level debugger, known as `g88` [Bed90], allows the development environment to be hosted under a friendly Unix account, and provides downloading features to run programs.

Several sample applications have been developed to demonstrate the kernel operation. A user level serial port device driver implements a file server and terminal tty server to a Unix host. An ethernet device driver demonstrates network data throughput through the kernel IPC services.

Chapter 2

Run-time environment overview

The main goal for an operating system is to provide a reliable and convenient work environment. Hidden beneath the operating system is the raw physical hardware, an environment far too exacting for higher level users to deal with. Just as users are given a run-time environment to work in, the operating system has its own run-time environment: main memory, exceptions and interrupts, and the various devices that make up a general purpose computer. As well, the operating system uses tools such as compilers and debuggers. This chapter provides an overall view of the hardware and software environment available to us. The first part of this chapter provides an overview of the current hardware platform. The second part then describes the low level software environment and development tools.

In the spirit of efficiency and simplicity, the Raven kernel makes assumptions about the hardware architecture model: the kernel is designed for shared memory multiprocessor platforms, where each processor has the same view of memory and devices. Non-uniform memory access (NUMA) machines and distributed memory machines require different considerations in terms of memory management, scheduling, and input/output. Some systems, such as Mach [ABB$^+$86], provide hardware compatibility layers to isolate porting details, and this can alleviate porting to different classes of hardware architectures. But this generality has its costs in terms of additional code size and complexity.

2.1 Hardware Overview

The hardware platform used for the implementation of the kernel is known as the Motorola MVME188 Hypermodule [Gro90]. The Hypermodule is a general purpose shared memory multiprocessor, based on Motorola's 88100 RISC architecture [Mot88a]. The Hypermodule contains four 88100 processors, each with a uniform view of 32MB of shared memory. In addition, the Hypermodule contains many devices such as timers, interrupt management, a VMEbus controller, and serial ports to make up a general purpose computer. This section will examine many of the relevant aspects to this hardware, as it pertains to the kernel.

2.1.1 The MC88100 RISC Microprocessor

Each of the four 88100 processors runs at a clock speed of 25MHz. Most instructions execute in a single clock cycle. Using four processors, the Hypermodule can theoretically achieve a performance rating of about 60 MIPS[1].

The instruction set is typical of many 32-bit RISC microprocessors: a simple set of single cycle instructions used to build higher level constructs. All instructions are represented in memory as 32-bit words, simplifying the decoding phase. A delayed branching feature can be used to alleviate pipeline stalls when executing branch instructions.

There are 32 general purpose user level registers, r0 — r31, each 32-bits wide. Register r0 is read-only and always holds a value of zero. In supervisor mode, the 88100 contains 21 additional control registers: cr0 to cr20. These registers reflect the state of the processor mode, data unit pipeline, integer unit pipeline, and general purpose scratch registers. The floating point execution unit contains 11 more registers: fcr0 to fcr8, and user registers fpcr62 and fpcr63.

2.1.2 The MC88200 Cache/Memory Management Unit

The 88200 CMMU augments the 88100 processor by providing instruction and data caches, as well as adding memory management. Each processor on the MVME188 uses two 88200's, giving each CPU 16KB of code and 16KB of data cache. A memory bus snooping protocol allows cache coherency between all caches in the system.

[1] VAX MIPS

The memory management section implements a two level page table scheme with a page granularity of 4096 bytes. Figure 2.2 shows the layout of this page table scheme. Up to 4GB can be mapped to a single address space. Each page in that address space can have various combinations of the following attributes: no translation, cache-disable, writethrough, write inhibit (for read-only pages), and snoop enable. The detailed settings for these attribute bits can be found in the 88200 technical documentation.

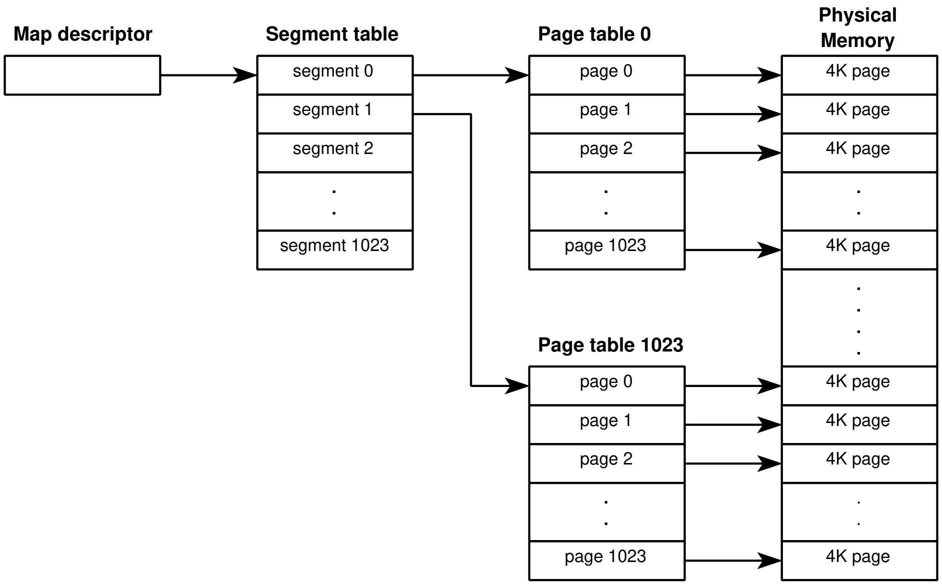

Figure 2.2: 88200 CMMU address translation table format.

Access to the 88200 registers for programming is done through the Hypermodule utility space (discussed below). Each of the 88200's in the system can be accessed separately, allowing cache flushes and page table management to be performed by the host processors.

2.1.3 System Controller

In addition to the processor, CMMU's and memory, the Hypermodule has a system controller board that contains all of the additional functionality and glue that make up a general purpose computer. Components such as timers, serial ports, and the VMEbus controller are found on this system controller board.

The system controller board maps all of its devices and memory into the processor's physical address space. To access a device on the controller board, the processor executes load/store operations to the memory-mapped device registers. Other devices and resources can be added to the physical address space through the VMEbus interface.

Figure 2.3 shows the default physical address space that the Hypermodule resources reside in. The remainder of this section will be devoted to examining each portion of this address space, and noting the features pertinent to kernel operation. For additional detail, consult the hardware manuals [Gro90] and [Mot88a].

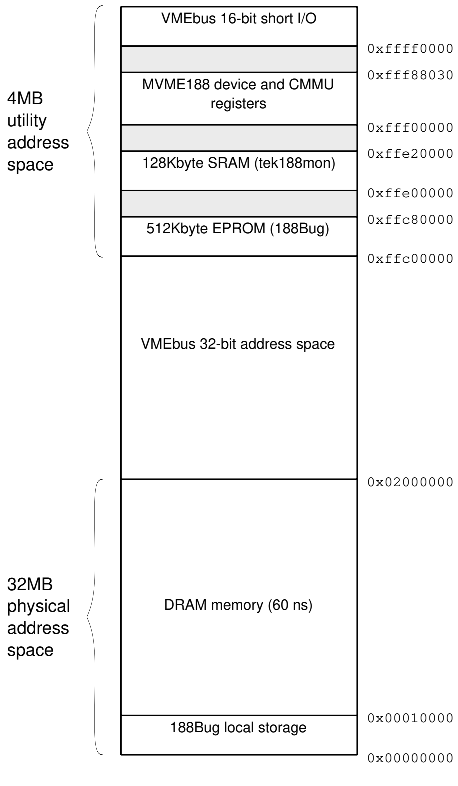

Figure 2.3: Hypermodule physical address map, showing DRAM, VMEbus, and device utility space.

Utility address space

All onboard devices and resources are located in an upper region of memory called the utility space. The utility space address map spans a 4MB region from `0xffc00000` to `0xffffffff`. Each resource within the utility space are given their own section of address space to reside in, padded by null memory space.

Access to the utility space by the operating system kernel is a necessity. While the default base address of the utility space can be modified by switching jumpers, it cannot be made to go away completely. In fact, the address translation hardware of the 88200's contain hardwired entries to always make the utility space available in the supervisor memory map. This is done so that the operating system can find the utility space in the event of translation table errors or software malfunction.

The following devices are made available inside the utility address space. Some of these devices are shared by user level memory maps. For instance, in order to implement a user level serial port driver, the DUART registers must be mapped in user space.

MC68681 DUART/Timer

The MC68681 provides the system software with two RS-232C compatible serial ports and a programmable interval timer. The serial port speeds can be programmed to support varying data rates, but also support 18 preprogrammed rates from 50bps to 38.4kbps. The kernel uses one serial port in polled mode for low level debugging support, and another interrupt driven port to connect with attached terminals or Unix hosts.

The kernel also uses this chip's interval timer. Using the 3.6864MHz clock on the DUART, an interrupt can be generated with a period anywhere from 540 nsec to 563 msec. The kernel allows this period to be configured at boot time, in terms of ticks per second. This value is then used as the system clock tick.

Z8536 Counter/Timer

In addition to the interval timer provided by the MC68681, the Z8536 Counter/Timer device serves as an enhanced timer service. The Z8536 contains three individual 16-bit timers, each using a clock rate of 4MHz. The timers can be programmed separately, or cascaded together to provide higher resolutions. The kernel cascades two of these timers, allowing intervals from 1 usec to 4295 sec (71 minutes) to be timed with 1 usec accuracy. This resolution is good enough to benchmark relatively small sections of code with reasonable accuracy.

The third timer on the Z8536 can be enabled as a watchdog timer for the Hypermodule board. When programmed to do so, the timer will trigger a reset sequence which reboots the machine. This feature allows the system to self-recover from fatal system crashes.

MK48T02 Time-of-Day Clock

As if two timers were not enough, the Hypermodule also contains a time-of-day clock. This clock can be used to maintain the correct wall clock time (it is battery backed and will correctly maintain the time when power is off). Currently, the kernel doesn't use the notion of wall clock time anywhere, so this device is not used.

MVME6000 VMEbus Controller

Access to VMEbus peripherals is provided by the MVME6000 "VMEchip" controller [Mot88d]. This device manages the interface between the Hypermodule memory bus (master) and the VMEbus, and the attached devices (slaves). It allows regions of the VMEbus address spaces to be mapped into the Hypermodule's local physical address space. For example, the Ethernet device driver uses the VMEchip to map in the Ethernet board's device registers at location `0x10000000`.

In addition to providing access to slave devices, the VMEchip contains features that facilitate operation in a multiprocessor environment. If more than one Hypermodule board is installed on the VMEbus, the VMEchip on each board can be used for coordination and synchronization services.

128KB SRAM

The Hypermodule board contains 128KB of non-volatile (battery backed) SRAM, which can be used to retain data and program instructions while the power is turned off.

Currently, the kernel uses this section of memory to store the g88 debug monitor. The debug monitor code rarely changes, so once it is downloaded into the SRAM, it is there for good. The next time the Hypermodule is started, the debugger checksums the monitor area to see if it is complete. If the monitor checksum passes, there is no need to download a fresh copy (which can take a while at serial-port speed).

512KB EPROM and 188BUG Debugger

An EPROM chip module contains a standalone, onboard monitor/debugger known as 188Bug [Mot88c]. This is a low level interactive debugger which can be accessed via the serial port using a dumb terminal. The debugger is non-symbolic but full featured, with the ability to perform raw operations on disk, tape and serial port devices. This allows code such as operating systems to be easily bootstrapped from disk, tape, or serial port.

In addition to the software debugging features, a full suite of diagnostics are provided to test and exercise the hardware.

Since 188BUG is non-symbolic, it can be difficult to debug programs written in high level languages like C. Instead, the kernel uses a gdb-based debugger called g88 to run and test the system. g88 allows kernel code to be downloaded and interactively debugged using the familiar gdb environment. Breakpoints can be set, data examined, just like debugging a regular user level process under Unix.

DRAM address space

The Hypermodule can be configured to hold up to 64MB of 60ns DRAM, in 16MB increments. All of this physical memory is symmetrically visible and available to all processors in the system. Peak memory bandwidth is 44.4MB/sec for reads, and 66.7MB/sec for writes.

The onboard debugger, 188Bug, reserves the lowest 64KB region for its own use (exception table, local variables, code, and stack). This region must be preserved for the 188Big to operate properly. Sometimes 188Bug is useful, so the kernel doesn't intrude on its territory. Useable memory begins at `0x10000` and grows upwards.

VMEbus address space

The Hypermodule VMEbus chipset supports 32-, 24-, and 16-bit VMEbus addressing modes. The 16-bit VMEbus SHORTIO space is hardwired to live in the upper 64KB region of memory, so that master and slave devices always know how to locate this region. All other addresses within the physical address space, between the end of DRAM and the beginning of utility space, can be used for mapping in portions of the 32-bit and 24-bit VMEbus address spaces.

As a convention, the kernel uses 32-bit VMEbus address mapping for external devices. Currently only the Ethernet board is mapped in this fashion, so this convention is certain to change for other kinds of devices.

2.1.4 Interrupt management

In many earlier multiprocessor systems, one special processor was designated as the I/O processor. Doing so would help simplify an already complex memory and interrupt bus. This special processor is specifically wired to accept all interrupts from external devices. While this does leave other processors to do useful work for non-I/O bound operations, a processor that requires I/O must always go through the special I/O processor. This can introduce a throughput bottleneck in the system, and can complicate I/O and device driver code.

The Hypermodule interrupt management scheme 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 (IEN). In a system with four processors, there are 4 interrupt enable registers named IEN0 to IEN3.

The hardware supports up to 32 different interrupt sources. Each bit in the IEN registers corresponds to one of the possible interrupt sources. Some of the common interrupt sources are: DUART timer, DUART serial port, VMEbus IRQ0 – IRQ7, and software interrupts (SWI).

Individual interrupt sources can be set to occur on any combination of the processors in the system. It is possible to configure the IEN registers such that multiple processors receive the same interrupt. In this case, all interrupted processors will halt execution and branch to the interrupt handler when the particular interrupt occurs. This could generate a great deal of contention if spin-locks are used within the interrupt handler. Therefore, certain interrupts should be enabled on only a single processor at a time. The kernel interrupt architecture manages the setting of the interrupt enable bits to minimize latency to interrupt handlers.

For example, the Ethernet device driver is a user level task that requires to be executed every time an interrupt is generated on the Ethernet board. Switching in the task can involve cache flushes and address space changes and is therefore an expensive operation. This task switch is avoided if by setting Ethernet interrupt enable bit properly on the processor which is currently executing the Ethernet task. Doing so localizes the interrupt to a single processor that has the correct address space activated.

If there is more than one processor allocated to an interrupt driver task, only one processor at a time has the device interrupt enable bit set. Which processor it is, is determined by the kernel. If the interruptable processor is relinquished, another processor allocated to the device task is assigned the interrupt bit.

When a processor receives an interrupt exception, one of the first things it needs to do is find out exactly which device caused the exception. The interrupt status register (IST) allows this to be easily accomplished. The IST is bit-for-bit similar to the IEN registers, except that it reflects the status of all interrupts in the system. If a bit is set in the IST, then the corresponding

Register	Compiler and system usage
r0	Read-only 0 constant
r1	Function call return address
r2 – r9	Function call parameters
r10 – r13	Function temporary registers
r14 – r25	Function preserved registers
r26	Temporary scratch register
r27	Temporary scratch register
r28	Locking and upcall status bits
r29	User thread context save pointer
r30	Stack frame pointer
r31	Stack pointer

Table 2.1: GNU C computer and kernel register usage.

device is requesting an interrupt.

Note that there is only one IST, and not one for each processor. The IST gives the global status of all interrupting devices, and the IEN registers give the enable mask for each processor. So to check whether a particular processor received a particular interrupt, the processor must AND the IST its IEN register: `status = ien_reg[cpu] & ist_reg`.

2.2 Software environment

In the hardware overview section, various features of the hardware runtime environment were discussed. In this section, the software development environment is examined. This environment includes the `g88` debugger and simulator, compiler tools, and software conventions such as register usage. Other factors which influence the runtime environment are the processor supervisor/user state settings, and supervisor register conventions.

2.2.1 The gcc compiler and tools

The C compiler used is the ANSI-compliant GNU gcc version 1.37.29. All other tools used for compiling and linking executable code, such as the linker and assembler, are also GNU tools.

The C compiler uses a standard format for processor register allocation. Table 2.1 shows this register convention. Functions can always rely on registers `r10` – `r13` to be available for temporary scratch purposes. Registers `r14` – `r25` can also be used for general purpose storage, but they must be preserved by the called function if they are to be used. The C stack frame contains the frame pointer, saved registers of the previous function, function return address, and local variable storage for the function. This information is useful when interfacing C programs with handcoded assembler.

2.2.2 The g88 kernel debugger

The g88 cross-debugger/simulator [Bed90] is a GNU gdb [Sta89] based kernel debugger. Implemented as an extension to gdb 3.2, g88 allows Hypermodule users to download code and interactively debug their programs from within a standard gdb terminal session. g88 can set breakpoints, step through code, examine variables, etc. – just about everything one expects from a Unix version of gdb.

g88 runs on a Unix workstation and connects to the Hypermodule hardware via two serial ports. One serial port is used for the gdb command protocol, console input/output and downloading code. The other is used as a software controlled interrupt and reset line. This line is connected to the Hypermodule reset and interrupt logic, and is used to reboot and interrupt the target. For example, when control-C is pressed within g88, the interrupt logic is toggled, generating an ABRT interrupt on the Hypermodule. The onboard g88 monitor program recognizes this interrupt, suspends kernel execution, and returns control back to the g88. This allows users to interactively debug their code running on the Hypermodule, with the same control as if it were a regular Unix process.

When starting a session, g88 downloads a monitor program to the Hypermodule memory. This monitor, known as g88mon, handles the g88 serial port communication protocol between the Unix host and Hypermodule. It manages all the gdb features such as breakpoints, single stepping and memory examination. The monitor is designed and implemented to be as unobtrusive as possible. It is downloaded into a portion of the Hypermodule non-volatile SRAM, and resides there until it is erased. The program being debugged does not have any knowledge of g88mon existence in the system.

2.2.3 The g88 Hypermodule simulator

g88 also contains a complete instruction level simulator of the MVME188 Hypermodule. The simulator provides a virtual environment that is a clone of the Hypermodule: four 88100 processors, eight 88200 CMMUs, and all the system controller devices. An environment variable controls the size of the simulated physical memory size. Programs running in the simulator are virtually oblivious to the fact that they are running in a simulated environment. g88 provides the customary gdb interface to the simulator, so programs can be downloaded, executed, and debugged.

When the simulator is used to execute something, no physical serial links are necessary because g88 simulates the hardware directly in its address space on the Unix host. The overhead associated with sending commands across the serial link is eliminated. Thus the downloading of code and overall communication with the "physical" hardware is much faster. This property makes the simulator ideal for debugging and development, where a fast edit/compile/test cycle is desirable. It takes about 60 seconds to download the Raven kernel to the real hardware, compared to a fraction of a second for the simulated hardware. Similarly, overall debugging commands on the simulator are much faster.

Raw execution speed of the simulator is of course much slower than the real hardware (about 100 times slower on a SPARC 1+). However this does not impact the usability of the Raven kernel on the simulator for debugging purposes. The kernel itself and most user level programs at this point are not compute bound, so most executions are fast enough.

The current version of g88 has one major limitation: it can only be used to debug supervisor code. This means that g88 cannot debug user level programs. This limitation stems from the fact that gdb 3.2 is only able to manage one execution context (i.e., one program) at a time. However, g88 is still under development by its author, in cooperation with Horizon Research, for Mach 3.0 work. An upgrade to g88 which will allow user level debugging and performance improvements will soon be available.

Chapter 3

Kernel level implementation

The Raven kernel is split into two distinct entities: the supervisor kernel, and the user level kernel. Each of these entities is implemented as separate executable programs. This chapter discusses the design and implementation of the supervisor kernel part.

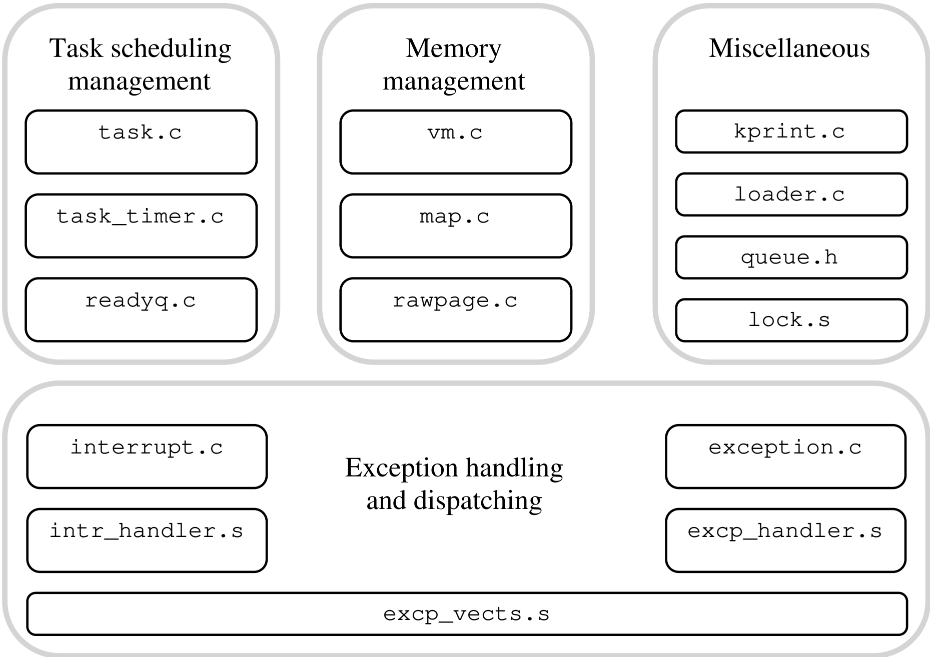

Figure 3.4: Kernel source code organization.

Figure 3.4 shows the modular breakdown of the system, and the source code files involved. The first section in this chapter discusses the overall programming model of the kernel, and the following sections describe the code modules in detail.

Several of the kernel modules export a system call interface to the user level. These system calls are broken into two categories: calls available for general purpose user applications, and calls available for user level kernels only. For modules which export such a system call interface, each description in this chapter will contain a table summarizing the interface provided by that module.

System calls, and internal kernel functions, often return a success or failure condition to the caller. Throughout the implementation of the kernel, the following convention is used for function return values. Functions that complete successfully always return the value `OK`. Functions that fail in some manner during their invocation return the value `FAILED`.

3.1 The interrupt model

The implementation of the kernel is based on the interrupt model, as opposed to the process model. In the process model, a kernel is composed of several cooperating processes, each of which has their own stack and local state variables. Interrupts and preemption are normally allowed during execution. The processes must be scheduled by the kernel, and special cases for preemption locking and concurrency locking must be explicitly coded.

In the interrupt model, the kernel can be viewed as one big interrupt handler. All kernel invocations, including system calls, device interrupts, and exceptions, enter into the kernel though the exception vector table, `excp_vects.s`. From this point, the low level exception handler routines allocate a stack for the processor and dispatches the event.

Any number of processors can be executing within the kernel at the same time. While a processor is executing within the kernel, interrupts and preemption is disabled for that processor. This simplifies the implementation quite substantially, because there is no need for special purpose preemption locking and protection. All calls to the kernel are non-blocking: except for the processor relinquishment call, all kernel calls return immediately to the user after executing. When control returns to user space, interrupts and preemption is re-enabled.

Each processor in the system uses its own dedicated kernel stack. In a four processor system, there are four dedicated kernel stacks. The kernel stacks are located at well-known fixed locations in the supervisor memory space, and are allocated at very beginning of initialization time. Each time the kernel is invoked, the processor stack pointer is set to the top of its kernel stack. The system can assume that there is only one execution context per processor while running inside the kernel. This substantially simplifies implementation issues.

3.2 Low level mutual exclusion

In a shared memory parallel environment like the Hypermodule, many algorithms require that sections of their code have atomicity. These critical sections of code must be executed atomically in *mutual exclusion* with their neighbours, or risk leaving their state inconsistent. In the kernel, these algorithms range from simple enqueue/dequeue operations on a shared queue, to ensuring sequential access to device registers.

Several techniques exist to ensure mutual exclusion. One technique is to avoid critical sections altogether by implementing data structures as *lock-free* objects [Ber91] and optimistic synchronization [MP89]. In this case, a compare-and-swap operation allows data structures to be concurrently accessed with consistency. However, the lack of proper hardware support for compare-and-swap on the Hypermodule hardware does not make this algorithm practical.

Other techniques range from the simple spin-lock to the higher level semaphore. The latter technique relies on the operating system to "schedule around" critical hot spots by relinquishing the processor to another thread when such a spot is reached. This technique requires some cooperation with the operating system, commonly in the form of semaphore data structures, which in themselves require mutual exclusion. Hence the requirement for a more primitive mutual exclusion technique.

3.2.1 Spin locks

The spin lock is a brute force method of providing mutual exclusion around sections of code. The algorithm tests a shared lock variable to see if the lock is free or used. If the lock is free, the lock is claimed by setting its state to "locked", and execution continues. If the lock is not free, continually check until it is free. A lock is freed by storing a "free" value to the lock variable.

While waiting for a lock to become free, the processor cannot do anything else. In a system with many processors, this property can become a bottleneck when frequently accessing a shared resource.

3.2.2 lock_wait() implementation

On many processors, a test-and-set instruction is used to ensure atomicity of the lock variable test and set stage. The test stage is broken into two parts: one instruction loads the value of the lock variable, another instruction tests its value. If the value of the lock variable is altered after the load instruction but before the test, the algorithm will fail. To prevent this problem, the 88100 instruction set includes the `xmem` instruction. The `xmem` instruction atomically exchanges the contents of a register with the contents of a memory location. The load and store accesses of the `xmem` instruction are indivisible: the instruction cannot be interrupted part-way through its execution.

Using the atomic exchange instruction, the lock acquire routine can be safely implemented in the following manner:

```
_lock_wait:                         ; r2 <- lock variable addr
    or      r10, r0, 1              ; set a lock value
lw: xmem    r10, r2, r0             ; atomic exchange
    bcnd    ne0, r10, lw            ; try lock again if not zero.
    jmp     r1                      ; return to caller
```

`lock_wait()` begins by putting a "locked" value into register `r10`. This register is then exchanged with the lock variable stored in memory at the address `r2`. As a result of the exchange, `r10` is loaded with the previous lock value. If this value is non-zero, then the `bcnd` instruction branches to the top of the routine, where the test starts again. This sequence is repeated until a "free" value is found in the lock value. The memory bus transactions generated by the repeated accesses to the lock variable can quickly saturate the memory bus, hindering other processors in the system from doing useful work.

An optimization can be achieved by relying on the data cache to maintain a coherent copy of the lock variable. In this case, initially the processor spins on a cached copy of the lock, generating negligible memory bus accesses. If the lock is freed, the processor performs an `xmem` to grab the lock. If this `xmem` fails, then return to spinning on the cached copy. This algorithm is implemented as follows, and can be found in `lock.s`:

```
void lock_wait(int *lock_addr);

_lock_wait:                          ; r2 <- lock variable addr
    ld      r10, r2, 0               ; read the lock value
    bcnd    ne0, r10, _lock_wait     ; loop if lock is busy.
    or      r10, r0, 1               ; set a lock value.
    xmem    r10, r2, r0              ; atomic exchange
    bcnd    ne0, r10, _lock_wait     ; try lock again if nonzero
    jmp     r1                       ; return to caller
```

3.2.3 lock_free() implementation

Freeing a lock is trivial. All that needs to be done is to store a "free" value to the lock variable. This can be accomplished in a single `st` instruction on the 88100.

```
void lock_free(int *lock_addr);

_lock_free:                          ; r2 <- lock variable addr
    jmp.n   r1                       ; return to caller
    st      r0, r2, 0                ; store "free" value
```

In this routine, the `jmp.n r1` instruction demonstrates the use of delayed branching on the 88100. The instruction cycle immediately after a control transfer instruction, such as jump, is known as the delay slot. This is where the processor figures out where execution should continue. While it does this, another instruction can be executed. In this case, the `st r0, r2, 0` instruction is executed in the `jmp r1` delay slot.

3.2.4 Lock initialization

Before using a spin lock, a lock variable must be allocated and initialized. The lock variable is of type `int` and can be allocated in any appropriate fashion, such as statically at compile time.

Initializing a lock variable is as simple as assigning zero to it:

```
int my_lock = 0;

lock_wait(&my_lock);
....
lock_free(&my_lock);
```

3.2.5 Summary

The simple spin lock can become surprisingly complex, as shown by [And89] and [KLMO91]. These more complex techniques arise from differences in hardware characteristics, such as the number of processors, memory bus architecture, and instruction sets.

In a shared memory system such as the Hypermodule with only four processors, spinning on a cached copy of the lock is sufficient to attain good performance. This algorithm is implemented by the routines `lock_wait()` and `lock_free()`. Modules throughout the kernel use these routines to provided mutual exclusion for shared data structures and hardware devices.

3.3 List and Queue Management

List and queue data structures are primitive and fundamental building blocks for operating systems. Much of the information stored within the kernel, such as task control blocks and memory regions, are kept track of using linked lists. The operations involved are insertions, removals, and traversals. This section describes these operations implemented in the kernel and the data structures involved.

All of the queue management routines are implemented as C macros, so they are easily inlined to avoid procedure call overhead. The routines could be coded in assembler, but inlining ability and portability would be sacrificed.

The queue macros are designed to be type-flexible. This allows a small set of macro routines to be general enough to handle any queue node structure. To do this, the macros require that the caller specify the node type. When the macros are expanded at compile time, the proper type casting is performed using the supplied type. This technique is similar to the queue macros seen in the Mach kernel.

There are some algorithms where an ordered list of items is a basic requirement. For example, the system clock timer maintains an ordered list of task control blocks. The list is sorted by the `wake_time` key. This allows the clock timer to examine only the head node on the list to check for a timer expiry.

The queue macro package does not provide any macros for ordered list or priority queue management. It is felt that such a structure can be efficiently constructed using the supplied, more primitive macros.

In a multiprocessor environment, concurrency control is required to protect against multiple accesses to common data structures. Many of the system queues are shared amongst all the processors. The queue management library does not implement locking, leaving it to be done explicitly by the user. This option can reduce overhead when locking is not required.

The queue and linked list structures are simple enough to allow the use of the spin lock library seen in the previous section. The use of spin locks to manage queue structure access is demonstrated in the code fragments below.

3.3.1 Single linked lists

Most of the data that is kept on the linked lists are arrays of statically allocated control blocks. Some control block data structures are very simple and don't even need the flexibility provided with a double-linked list. For example, the `rawpage_table[]`, which keeps track of physical pages in the system, uses a single-linked list to remember all the free pages. To get a new page, `rawpage_alloc()` dequeues the head pointer from the free list. To free a page back to the pool, `rawpage_free()` enqueues the page onto the free list head. The rawpage routines don't need to be able to remove elements from the middle or end of the list, so a backward link is not required.

Simple linked lists as in the above example are used throughout the system whenever a simple allocate/free operation of some data object is required. Since the lists are so simple, the queue management library doesn't provide any help – it's up to the kernel programmer to provide the head and next pointers for the list structure. This allows the list types to be tailored for use within a code module.

3.3.2 Queues

The most common type of list used in the kernel is the double-linked list. These lists are used extensively as FIFO queues for task, thread, and semaphore management, to name a few. Since the queue operations are so basic to the operation of the kernel, a set of macros and data structures are provided to simplify the job and make the code look a bit cleaner.

All queues have a master handle of type `QUEUE` which maintain the head and tail pointer for the linked list. The other data structure that is used is the `QUEUE_LINK` structure. The `QUEUE_LINK` structure contains the next and previous pointers, and is used as the "link" for each node in the list. Each node in a queue must have at least one of these structures. Using more than one link per node allows nodes to be linked to several different queues at once.

The task control block table, `task_table[]`, provides a good example of how these structures are used. Each `TASK` structure contains several fields of data, two of which are the `QUEUE_LINK` structures:

```
/* next/prev link for queue structures */
QUEUE_LINK sched_link;      /* scheduler queue */
QUEUE_LINK timer_link;      /* timed event queue */
```

These links are used to connect the task control block to the scheduler queue and timer queue[1].

Enqueue operations

There are three possible enqueue operations: enqueue at head of list, enqueue at the tail of list, and enqueue before a given node. For queues that need to pay attention to FIFO ordering, the

[1]The timer queue is used by the system clock timer to notify tasks of timed events.

enqueue/dequeue operation always occur on opposite ends of the queue. The kernel follows the standard convention that nodes are always enqueued at the tail and dequeued from the head, as seen in the following example from `task_create()`:

```
/* queue the task on the suspended queue */
lock_wait(&task_susp_q_lock);
ENQUEUE_TAIL(task_susp_q, task, TASK, sched_link);
lock_free(&task_susp_q_lock);
```

The following macros can be used for enqueue operations:

- `ENQUEUE_HEAD(queue, node, node_type, node_link)`
- `ENQUEUE_TAIL(queue, node, node_type, node_link)`
- `ENQUEUE_ITEM(node, next_node, node_type, node_link)`

  This macro inserts a node directly before the supplied `next_node`. This macro can not be used when `next_node` is either the head or tail of the list – in those cases, the other two enqueue routines should be used.

Dequeue operations

The following two dequeue operations are provided.

- `DEQUEUE_HEAD(queue, node, node_type, link)`
- `DEQUEUE_ITEM(queue, node, node_type, link)`

Note that these macros do not check for empty queues. It's an error to try and dequeue a node from an empty list. To protect against this error, the `QUEUE_EMPTY()` macro can be used to check for the empty/non-empty condition. For example, `task_create()` uses the following code to get a new task control block:

```
lock_wait(&task_free_q_lock);
```

```
if ( QUEUE_EMPTY(task_free_q) )
{
    lock_free(&task_free_q_lock);
    kprint("task_create: no free tasks!\n");
    *task_id = -1;
    return(FAILED);
}

/* get a free task descriptor */
DEQUEUE_HEAD(task_free_q, task, TASK, sched_link);
lock_free(&task_free_q_lock);
```

Miscellaneous queue operations

As the previous example illustrates, the following queue operations can sometimes be useful:

- `QUEUE_EMPTY(queue)`

  This macro evaluates to non-zero if the specified queue is empty.

- `QUEUE_HEAD(queue)`

  This macro returns the first node of the queue.

- `QUEUE_NEXT(queue, node, link)`

  This macro returns the next node on the link after the specified node.

Allocating and initializing a queue

Before a queue list can be created, a master queue handle must be allocated and initialized. The kernel uses the policy of static allocation wherever possible, so the handles are normally allocated at compile time by declaring a variable of the `QUEUE` type. The following code from `task.c` demonstrates this:

```
QUEUE task_free_q; /* queue of free tasks */
QUEUE task_susp_q; /* queue of suspended tasks */

int task_free_q_lock; /* spin locks for the above queues */
int task_susp_q_lock;
```

Once the master queue handle is allocated, it must be initialized to contain the value of an empty list. The `QUEUE_INIT()` macro performs this task, as demonstrated in the following initialization code from `task_init()`:

```
/* create task system queues. */
QUEUE_INIT(task_free_q);  task_free_q_lock = 0;
QUEUE_INIT(task_susp_q);  task_susp_q_lock = 0;
```

3.3.3 Other linked list schemes

The Xinu operating system [Com84] uses an interesting technique to implement double-linked lists for its process queues. In this scheme, the node links are stored in an array of links, completely separate from the process descriptors. There is a one-to-one mapping between links in the array and process descriptors. Both the links array and process table are indexed by an integer. So, if you know which link you are, you automatically know which process descriptor you belong to. Next/previous pointers in the links allow for double-links, and an extra key field allows ordered lists to be constructed.

This way of structuring a queue can make the enqueue/dequeue operations very efficient. However, it does have the limitation that a node can only reside on one queue at a time. The Raven kernel task scheduler and system clock timer require that a task be queued on two queues at once, so this scheme cannot be used.

3.3.4 Summary

Queues provide a means of ordering and organizing data. They are a basic component in operating system kernels. Both single linked and doubly linked lists are used throughout the kernel to organize resources such as task descriptors and memory pools. The routines used in the kernel to manage these queues are implemented as C macros which are portable and efficient.

3.4 Low level console input/output

During the development stages of an operating system, one of the greatest aids is a low level console output routine. Sometimes, the only way to debug at the kernel level is to output debugging strings to the console. The output routine should be simple enough that it can be executed from anywhere, independently of the rest of the system. This allows information to be printed out whether or not the kernel is functioning properly, or from within interrupt handlers and other delicate routines.

Using polled output in the presence of interrupt drivers can have negative consequences. So rather than always sending console output through the serial port, a dedicated portion of memory is set aside for console messages. This section of memory is known as the `kmsg` buffer.

Synpopsis	Interface availability	Description
`kprint()`	user application	Prints out a string to the console (polled output).
`kgetstr()`	user application	Waits for a string from the console (polled input).

Figure 3.5: `kprint` module system call summary.

The `kprint.c` module implements the console polled input/output driver. The table in Figure 3.5 summarizes the system call interface exported to the user level by this module.

3.4.1 Console output

`void kprint(char *str);`

This routine is the lowest level output routine. It takes a null-terminated string of characters as its argument, and uses polled output to send the string to the console.

`void kprintf(char *fmt, ...);`

This is the `kprint()` routine for formatted printing. It passes the format string and arguments to `sprintf()` for formatting, and then calls `kprint()` to output the resulted string. `kprintf()` understands the following format sequences: `%c`, `%s`, `%d`, and `%x`.

Special care should be taken when calling `kprintf()`: the output string should not be greater than 200 bytes, or the stack will be damaged. `kprintf()` allocates a temporary 200 byte output buffer on the caller's stack.

On the real hardware, the g88mon monitor controls communication across the serial port to provide access to the g88 console under Unix. In the simulated environment, the simulator contains a character console device at `0xffff0000`. Access to both interfaces uses strictly polled I/O.

```
void kprint_mode(int mode);
```

This routine controls the suppression of console output strings to the serial port device. In some cases, the blocking nature of polled I/O has negative consequences. Outputting a string across the serial port can take a long time in relation to other devices. For the Ethernet device, the time taken to output a debug message may result in lost packets.

`kprint_mode()` can help prevent this problem by allowing console output to be shut off under program control. This is useful when debugging code where interrupt activity is necessary, such debugging an Ethernet protocol stack. Passing a non-zero value for the `mode` argument allows output to the serial port. Passing a zero value for the `mode` argument suppresses serial port output. In either case, the `kmsg` buffer logs all output strings, which can be viewed at a later time.

3.4.2 Console output buffer

The console output buffer, or `kmsg` buffer, is a large buffer in the kernel memory space which logs all strings that have been outputted to the console using `kprint()`. Even strings that are sent with the `mode` disabled are placed in the `kmsg` buffer. This allows all console output to be viewed at a later time using the debugger.

The `kmsg_base` variable points to the beginning of the `kmsg` buffer. The g88 debugger can be used to print out console strings starting at the `kmsg_base` address. For example:

```
[0] (gdb) x/4s kmsg_base
Reading in symbols for kprint.c...done.
0x1e80000:      (char *) 0x1e80000 "CPU 0 started\n"
0x1e8000f:      (char *) 0x1e8000f "Executing on real hardware.\n\n"
0x1e8002d:      (char *) 0x1e8002d "total system memory: 8192 pages\n"
0x1e80065:      (char *) 0x1e80065 "avail user memory:   7751 pages\n"
[0] (gdb)
```

The size of the `kmsg` buffer is controlled at compile time by the `KMSG_BUF_SEGS` constant. This buffer is allocated in segment sizes of 512KB, to facilitate memory mapping.

3.4.3 Console input

```
int kgetstr(char *buf, int buf_size);
```

In addition to providing a means to output information, the kernel also has a primitive way to input information. This can be useful to ask confirmation questions at boot time, for instance. The caller supplies the preallocated buffer to place the inputted string in `buf` and the maximum length of the string in `buf_size`. The size of the inputted string is returned.

3.4.4 Console I/O initialization

Before doing any low-level console I/O, the structures must be initialized. This is done at boot time by the kernel initialization routine.

```
int kprint_init();
```

This routine sets the console output `mode` to 1, enabling output to the serial port device, and clears the `kmsg` buffer.

3.5 Memory management

During the normal operation of an operating system, memory allocation and deallocation are common tasks. Beneath the operating system lies a contiguous area of physical memory, pieces of which are parcelled off for various uses, and later returned. User level tasks do not work directly with these raw regions, however. The memory management system provides a protected linear address space for user level programs to work in.

The memory management system divides its work into three distinct modules: virtual memory management, memory mapping and cache management, and physical memory management. Figure 3.6 shows the layered relationship of each module. This section describes each module in detail.

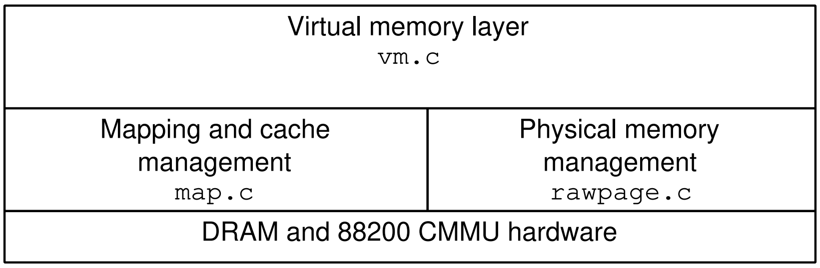

Figure 3.6: Memory management system decomposition.

The virtual memory module provides the user level system call interface to the memory management system. Routines are provided to allocate, deallocate, and share *regions* of memory. A region is a contiguous, page aligned portion of memory in a virtual address space.

The virtual memory module relies on the memory mapping module to manage the 88200 CMMU page tables. Regions of memory can allocated at specific addresses and with various page protection statuses. The processor instruction and data caches must also be maintained throughout memory allocation/deallocation operations. The mapping module also provides user level address space switching functions.

The physical memory module provides a simple and efficient interface to the virtual memory module for allocating and deallocating physical pages of memory. Pages are allocated and deallocated on a page granularity.

3.6 Physical memory management

The physical memory management module is very simple. Its job is to allocate physical pages from the raw memory pool, and return physical pages when the virtual memory layer is finished with them. While the virtual memory layer manages contiguous regions of an address space, the physical memory layer works on a simple page-by-page basis. The allocation and deallocation routines always work with single page units.

The `rawpage_table[]` keeps track of each physical memory page in the system. The table is statically allocated at compile time to contain one entry for each page in the system. When more physical memory is added to the system, more elements need to be allocated in this table. The static allocation strategy is used mainly for simplicity. However, in the future, this static table could easily be replaced with a table that is allocated at runtime, after probing the hardware for the true physical memory size.

The following data structures are used to manage the physical memory pages:

```
typedef struct raw_page_s
{
    int count;                  /* page reference count */
    int lock;                   /* spin-lock to protect access */
    struct raw_page_s *next;    /* next free page */
} RAW_PAGE;

RAW_PAGE rawpage_table[RAWPAGE_TABLE_SIZE];

RAW_PAGE *pages_free_head;      /* linked list of free pages */
int pages_free_lock;            /* spin-lock to protect list access */
```

The `RAW_PAGE` structure provides a handle for each physical memory page in the system. The `count` field implements a reference count for the page: when a page is allocated or shared, its reference count is incremented. When a page is freed, its reference count is decremented. This allows the system to keep track of which pages are used and which pages are unused.

There are three routines provided by the raw page module. All of these routines are available within the kernel only, they are not exported to the user level.

3.6.1 Allocating a physical page

To allocate a free physical page, the following routine is used.

```
void *rawpage_alloc();
```

No parameters are required. The base address of the physical memory page is returned. If there are no free pages, the system panics (something more elegant could be done in future).

Finding a free page is very simple. First, the free list lock is acquired. Then the first entry on the list is dequeued, and the lock is freed. The reference count is initialized. The address of the physical page is computed from the position of the page entry relative to the head of the raw page table. This address is returned to the caller.

3.6.2 Freeing a physical page

Freeing a physical page is equally as simple. Using the page address, the raw page table entry is computed. The entry lock is acquired and the reference count is decremented. If the new reference count is 0, the page entry is queued to the beginning of the free page list.

```
int rawpage_free(void *page);
```

The `rawpage_free()` call decrements the reference count for the given page, and returns it to the raw page pool if the count reaches 0.

3.6.3 Sharing a physical page

A reference count allows pages to be shared by different processes, keeping them from being freed and reused while they are being used elsewhere. If two processes are sharing a page, and one process frees the page from its address space, the page is not returned to the free memory pool. The page is only freed when the last process discards the page.

```
int rawpage_reference(void *page);
```

The `rawpage_reference()` call increments the reference count for the specified physical `page`.

3.6.4 Raw page initialization

At initialization time, the `rawpage_table[]` is initialized and the free pages linked list is created. Even at boot time, many pages are already allocated for the kernel data and code areas. These pages are removed from the free list and their reference count is marked for use. Access to the list of free pages is protected using the `pages_free_lock` spin-lock. The allocate/free routines must acquire the lock before pages can be removed or added from the free list.

```
int rawpage_init();
```

The `rawpage_init()` function is called by the boot processor to initialize the physical memory pool. It returns the total number of free physical pages in the pool.

3.7 Memory mapping and cache management

The map module is responsible for controlling the 88200 memory management unit [Mot88b]. All aspects to do with address translation and cache management are encapsulated in this module. Changes in the memory management hardware would only require changes to this module.

Routines are provided to create and manage the hardware translation tables and caching parameters. These routines are used by the virtual memory layer to provide the user and supervisor space with virtual addressing. These address spaces include physical memory, as well as access to hardware device control registers. None of these routines are user-accessible, they are only used locally within the kernel.

3.7.1 Translation tables

As described in the hardware overview section, the 88200 uses a two-level translation table to allow mapping of a 4GB address space. The first level segment table contains 1024 entries that point to page tables. Each page table contains 1024 entries that provide the translation for a single page of 4096 bytes. Refer to Figure 2.2 for a description of the translation table format.

In addition to address mapping information, the translation table entries contain attribute bits that control cache and protection status. Different settings of these bits are used for different types of memory pages. For example, since code pages are not modified during execution, code pages are marked read-only/cache-enabled. Device control registers are marked read-write/cache-inhibited. These attribute bits, as well as the 88200 control register offsets, can be found in the `registers.h` file. Refer to the 88200 user manual [Mot88b] for detailed information on the attribute bits.

Each 88200 maintains two registers called *area pointers*: the user area pointer (UAPR), and the supervisor area pointer (SAPR). The UAPR and SAPR contain a master pointer to the translation table for the user and supervisor address spaces, respectively. Context switching an address space requires setting the area pointer from one map to another. Throughout execution of the system, the supervisor area pointer remains constant, while the user area pointer changes for each running task. The map module keeps track of area pointers using the `MAP` structure:

```
typedef struct
{
    int map;    /* 88200 area pointer */
    int lock;   /* spin lock for this memory map */
} MAP;
```

The `MAP` structure contains the area pointer and a spin-lock to protect accesses to the translation table. The lock must be acquired by any map routine before changes to the translation table are allowed.

Each processor in the system uses two 88200 units: one for the code caching and translation, and one for data caching and translation. Each 88200 can use their own set of translation tables, or they can share a common single table. The Raven kernel opts to have two tables: one for code, and one for data. Thus each task descriptor in the system contains two map descriptors:

```
    MAP code_map;
    MAP data_map;
```

3.7.2 Translation lookaside buffer (TLB)

In addition to the instruction and data caches, the 88200 units also contain translation table caches, or translation lookaside buffers (TLB). The TLB stores frequently used translations, so that translation table searches are not required for each memory access. On the 88200, the TLB is termed the physical address translation cache (PATC), and contains 56 entries.

But unlike the instruction and data caches, the TLB caches are not automatically coherent across multiple cache units. Therefore, modifications to an activated translation table in the form of attribute changes or freed pages require TLB flushes if the address space is enabled on more than one processor. If this is not done, then remote TLB caches will not contain the correct translation table information.

Another type of translation lookaside buffer in the 88200 is the block address translation cache (BATC). There are ten entries in this cache which can be set programmatically. The entries remain until explicitly reprogrammed. Rather than mapping single 4KB pages, the BATC entries maps contiguous 512KB blocks. The operating system software can use these entries to provide mappings for high-use code or data regions.

3.7.3 Translation table storage area

A special region of the physical memory space is set aside at boot time for translation table storage. The kernel allocates segment and page translation tables for each task in the system from this common storage area. This area is located at the very top of physical memory. Its size is defined at compile time, in a granularity of 128 pages, by the `PAGE_TABLE_SEGS` constant in the file `kernel.h`. This allows the 88200 BATC entries to map the whole storage area into the kernel space at all times. Using the programmable BATC entries help avoid the chicken-and-egg problem that occurs when an unmapped page is allocated to store a page table.

Two macro routines are used to manage the allocation and deallocation of translation table pages. These routines are very efficient: they simply keep track of a free list of pages in the page table storage area. The `PTE_ALLOC()` macro allocates a page table by dequeuing the next free one. The `PTE_FREE()` macro returns a page to the storage area by enqueuing it back on the free queue.

3.7.4 Allocating a translation map

When a user task is created, two address maps are allocated to create the tasks code and data address spaces. The virtual memory layer uses the following routine to allocate these maps:

```
int map_alloc( MAP *map, int attrb );
```

The caller passes in a pointer to a preallocated, empty `MAP` structure and an attribute setting. An empty translation table is allocated using `PTE_ALLOC()`, and using the address of this table along with the attributes, the `map` area descriptor is created.

3.7.5 Freeing a translation map

When a user task is destroyed, the virtual memory layer frees its memory, and then calls this routine to free the tasks' translation tables. Since there are two translation tables for each user task, this routine is called twice to completely free a user task:

```
int map_free( MAP *map );
```

The caller passes in a pointer to the `MAP` structure for the table to be freed. The routine traverses the translation table structure and uses `PTE_FREE()` to release each table back to the free pool.

3.7.6 Enabling an address space

Before a user level task can execute, its address space must be activated. Enabling an address space is simply done by assigning the map area pointer to the 88200 UAPR register. But before this can be done, the previous user level TLB entries must be flushed. Otherwise, TLB entries from the previous address space would pollute the translations of the new address space.

```
int map_enable_task(int cpu, MAP *data_map, MAP *code_map);
```

The first parameter specifies the physical processor number to enable the address space on. This `cpu` number is used to calculate the appropriate 88200 register addresses to assign the provided map descriptors to (there are eight 88200 units in the system, two for each processor). The proper TLB entries are flushed and the new area pointers are assigned to the UAPR registers.

3.7.7 Mapping pages in an address space

When the virtual memory layer allocates physical memory to an address space, a new entry in the address space's translation tables is created. The allocation of memory to an address space includes the operation of allocating new memory regions, in addition to sharing pages between address spaces. The map module provides two routines to allows page mapping on single or multiple contiguous pages.

```
int map_page( MAP *map, void *page, void *logical_addr, int attrb );
```

The `map_page()` routine efficiently maps a single physical page to a logical address in an address space. The `map` parameter specifies the address space to map the page. The `page` parameter specifies the physical address of where the page to map is located. The `logical_addr` parameter specifies the address in the address space to map the page at. `attrb` specifies the attribute bits for the page.

```
int map_pages( MAP *map, void **addr, void **hint, int num_pages, int attrb );
```

This routine allows multiple contiguous pages to be mapped into an address space. Unlike the simpler `map_page()` call, this routine also allocates new physical pages from the rawpage module. `map_pages()` is most commonly used by the virtual memory layer to efficiently allocate contiguous regions of memory for a task. By supplying `*addr = NULL`, the routine will choose the next available free memory space and allocate `num_pages` pages starting there. The `hint` is supplied to provide a good starting location for the free memory search.

3.7.8 Unmapping pages from an address space

When the virtual memory layer removes physical pages from an address space, the associated translation table entries must be removed. For active address spaces, the TLB caches of the associated cache units must be notified of the changes (otherwise remote TLB caches may continue to contain the unmapped translation table entry). Every TLB in the system that is potentially caching the changed entries must be flushed. This includes remote processors that are executing the same address space.

On many multiprocessor systems, flushing a TLB on a remote processor involves sending the remote processor an interrupt and stalling the remote processor until the flush is complete [BRG+88]. This is because some machines do not allow flush operations on TLBs other than their own. Fortunately, the 88200 does allow flush operations to be invoked from remote processors. All the kernel needs to do is write an invalidate command to every 88200 unit in the system that is using the address space. Flushing the whole TLB when only one page table entry is changed can be wasteful. Fortunately, the 88200 allows system software to specify the flush granularity on a page basis. Before issuing the flush command, the page entry to invalidate is specified.

```
int unmap_page( MAP *map, void *logical_addr );
```

The `unmap_page()` routine removes the specified logical page from the specified address space

```
int unmap_pages( MAP *map, void *logical_addr, int num_pages, int free );
```

The `unmap_pages()` routine is the plural form of the above `unmap_page()` routine. In this case, multiple contiguous pages can be removed from an address space, starting at the specified logical address. If the `free` flag is non-zero, then each logical page is also returned to the physical memory pool using `rawpage_free()`. This allows large regions of an address space to be efficiently freed in one call.

If a page table becomes empty after the last logical page is removed from it, that page table is not returned to the page table pool (normally done by the `PTE_FREE()` macro). This could leave several page tables consumed for apparently no purpose. Curing this problem would require some form of garbage collection in the unmap routines. However, this problem is not as bad as it may seem. Any subsequent memory allocations will likely consume the next available entry in these empty page tables. So in fact, not garbage collecting these tables would allow them to be used directly without having to allocate new ones. Thus, an `PTE_FREE()/PTE_ALLOC()` iteration is saved.

3.7.9 Finding free logical addresses

Before the virtual memory layer can allocate memory to an address space, it needs to know where to place the memory in that address space. For example, when a user program calls `malloc()`, the user does not specify the location to allocate the memory. Instead, the next available address is chosen by the `malloc` library. Likewise, user level calls to the virtual memory allocator do not always specify the address to place the memory at. The virtual memory layer must decide where to allocate the memory. It uses the following routine to do this:

```
int map_find_free(MAP *map, void **addr, void **addr_hint, int num_pages);
```

Given a address map `map`, `map_find_free()` will traverse the map starting at address `*addr_hint`, looking for a free contiguous region of pages. The size of the region is specified by `num_pages`. The found address is returned in `*addr`. The `*addr_hint` address is updated to the next page beyond this contiguous region. Returns `OK` if successful, or `FAILED` if a contiguous free region could not be found.

3.7.10 Sharing and moving translations

Another common memory operation that the supervisor and user level tasks require is the ability to share and move memory between address spaces. Sharing memory allows multiple address spaces to read and write the contents of a single region of memory. The move operation allows contiguous memory regions to be passed from one address space to another, removing the mapping from the source. The following `map_share()` routine provides the virtual memory layer support to easily do this.

```
int map_share(MAP *src_map, void *src_addr, MAP *dest_map,
              void **dest_addr, void **dest_hint, int num_pages,
              int attrb, int share);
```

The routine takes a source map, a source logical address, and the number of contiguous pages to map into a destination map. If the destination address is not known, then `*dest_addr = NULL` is passed, and `map_find_free()` finds a suitable address. Passing the `dest_hint` address gives the search algorithm a good place to start. Passing a non-zero value for `share` retains the mapping in the source address space, passing a zero value causes the source region to be unmapped.

3.7.11 Map module initialization

The only state maintained by the map module are the translation tables. Storage for these tables is allocated at initialization time at the top of physical memory. The size of the storage area is configured at compile time using the `PAGE_TABLE_SEGS` constant in `kernel.h`. The `pte_init()` routine builds a free list to manage the allocation and deallocation of translation tables from this storage area.

```
void map_init();
```

This routine is called by the virtual memory initialization code. First, it calls `pte_init()` to initialize the translation table storage area. Then, it programs the 88200 BATC entries to map in the `kmsg_buf` and page table storage areas. These areas will then be available to the kernel when the supervisor address space is enabled.

3.8 Virtual memory management

All address spaces, user and supervisor, are created and managed by the virtual memory layer. The virtual memory layer provides the kernel and user level with an interface to allocate, deallocate, and share contiguous regions of memory within an address space. It uses the map module for setting up address translation, and the rawpage module for managing physical pages. The table in Figure 3.7 summarizes the routines exported to user level by this module.

Synpopsis	Interface availability	Description
`vm_alloc()`	user application	Allocates a region of virtual memory.
`vm_free()`	user application	Deallocates a region of virtual memory.
`vm_share()`	user application	Shares a region of memory between tasks.
`vm_move()`	user application	Moves a region of memory between tasks.
`vm_map_device()`	user application	Maps in a region of device registers.
`vm_unmap_device()`	user application	Unmaps a region of device registers.

Figure 3.7: Virtual memory system call summary.

Figure 3.8 shows a typical memory space for a user level task. A user level memory space contains four basic components: an executable code segment; a data segment for global variables and constants; dynamically allocated heap space; thread stacks; and hardware device mappings. The dark areas denote regions of memory that are mapped. The code and data regions are always contiguous, they contain the executable image as loaded from a file. The heap area starts at `USER_HEAP_START_ADDR` and continues to `USER_HEAP_END_ADDR` (currently `0x700000` and `0xffbf0000` in `context.h`). This heap area gives user programs approximately 3.9GB of virtual address space to work in.

Memory regions in the heap space are allocated and deallocated on demand throughout execution of the user level program. Thus, as Figure 3.8 illustrates, the heap space can become fragmented. A sparsely populated memory space can consume many translation tables, so allocation routines try to minimize this fragmentation by allocating space close to neighbours. However, this feature can be easily overridden, in the event that a specific address is required.

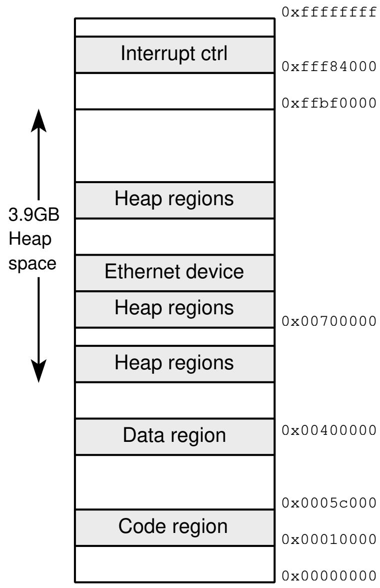

Figure 3.8: Typical user level memory space.

User level device drivers use the virtual memory layer to map in hardware device registers. While hardware devices reside at very specific physical addresses, their logical address can appear anywhere within the user's virtual space. Therefore, devices can be dynamically mapped anywhere within the heap storage area.

3.8.1 Region descriptors

The virtual memory module manages memory in objects called *regions*. A region is a contiguous, page aligned portion of memory in a virtual address space. The following `VM_REGION` structure describes a region:

```
typedef struct
{
    int id;        /* this region descriptor id */
    int task_id;   /* task that this region belongs to */
    MAP *map;      /* page table mapping descriptor */
    void *addr;    /* virtual address of region start */
    int size;      /* size of region */
    int attrb;     /* page attributes for region */
```

```
    /* next/prev link for VM region queue structures */
    QUEUE_LINK link;
} VM_REGION;

VM_REGION region_table[REGION_TABLE_SIZE];
QUEUE region_free_q;
int region_free_q_lock;
```

A region entry contains address mapping and ownership information. The start address, size, attributes, and map specify the mapping information for the region. The link field allows region entries to be queued together into lists. For example, the `region_free_q` keeps track of all unused region descriptors. The statically allocated `region_table` maintains a global pool of region entries. To allocate a new region entry, the `VM_GET_REGION()` macro dequeues the next available region. Old regions are recycled by enqueuing them back on the free list.

3.8.2 Allocating a region of memory

All user level tasks allocate heap memory through one single system call, `vm_alloc()`. To allocate memory, a new region entry is dequeued, and its values are initialized. Then, the `map_pages()` routine performs the physical memory allocation and mapping for the region. Finally, the region entry is enqueued onto the tasks heap list.

Each task maintains a list of heap regions that are allocated in its address space. When a task is destroyed, its memory must be returned to the system. The heap list allows each region to be kept track of, which can be efficiently freed all at once.

```
int vm_alloc(int *region_id, int task_id, void **addr, int size, int attrb );
```

`task_id` specifies the task to allocate memory into. `*addr` is set to contain the starting address to allocate the region of memory at. However, if the caller passes `*addr = NULL`, then the next available contiguous free region of memory is chosen. This address is then returned to the caller in `*addr`. The `size` parameter specifies the size of region to allocate, in bytes. All sizes are rounded up to the nearest page.

The `attrb` parameter specifies the page attributes for the 88200 memory management unit. These attribute settings are discussed in detail by the 88200 technical manual. The `registers.h` file contains all possible bit settings for this parameter. For example, to allocate a page of memory that can be shared amongst all threads in an address space, use `attrb = GLOBAL_BIT | VALID_BIT`.

The newly allocated region identifier is returned in `*region_id`. Further operations on this region of memory are specified by supplying this region identifier to the virtual memory system calls. For example, to free the region, call `vm_free()` with the returned `region_id`.

3.8.3 Freeing a region of memory, `vm_free()`

```
int vm_free(int region_id);
```

To free a region of memory, the `vm_free()` call is used. The call examines the supplied region entry, and uses the `unmap_pages()` routine to remove the address mapping and free the physical pages.

3.8.4 Sharing memory

Sharing memory provides a convenient and efficient way for address spaces to communicate. Memory regions can be shared between any number of address spaces.

```
int vm_share( int src_region_id, int *dest_region_id, int dest_task_id,
              void **dest_addr, int dest_attrb );
```

The caller supplies a source region identifier in `src_region_id`, and a memory address `*dest_addr` in the destination task to map the region at. If the destination address is not known ahead of time, then passing `*dest_addr = NULL` will automatically choose the next available free location. The chosen memory address will be returned in `*dest_addr`. The `dest_attrb` parameter specifies the page attributes for the destination address mapping. For example, a server may wish to provide a read-only page of memory for clients to examine. A region identifier for the newly created region is returned in `dest_region_id`.

3.8.5 Moving memory between tasks

A common operation throughout execution of device drivers and client/server interactions is the movement of data between address spaces. For example, initially a device driver will read a chunk of data from a device, and give it to a server for processing. The server will then send the data to clients. Each step could involve the movement of data across address spaces.

Sometimes, the amount of data passed between clients and servers is small, and therefore an explicit memory copy operation is the most efficient way to pass memory between address spaces. But for high speed devices, and especially block-mode devices with large block sizes, a memory mapping technique can eliminate the data copy altogether. The `vm_move()` system call provides a convenient interface to do this. Memory pages in one address space can be mapped out and mapped in to another address space.

```
int vm_move(int src_region_id, int dest_task_id, void **dest_addr,
            int dest_attrb);
```

The `src_region_id` specifies the source region to move. The destination task for the memory region is specified by `dest_task_id`. `*dest_addr` should be set to contain the starting logical address for the region of to appear in the destination address space. If the destination address is not known ahead of time, then passing `*dest_addr = NULL` will automatically choose the next available free location. The chosen memory address will be returned in `*dest_addr`. The `dest_attrb` parameter specifies the page attributes for the destination address mapping. The region identifier `src_region_id` remains the same in the destination task.

3.8.6 Mapping hardware devices

On the 88100 Hypermodule, access to hardware peripherals and on-board devices is accomplished through memory mapped registers. These registers appear at well-known memory locations in the physical address space. Many of these physical memory locations are hardwired, such as the 88200 registers, while others such as the VMEbus devices, can be configured via jumper settings or programmable registers.

User level device drivers must know exactly where in the physical address space their device registers reside. Using that information, the `vm_map_device()` can map in the appropriate physical memory locations to provide access to these device registers.

```
int vm_map_device( int *region_id, void *phys_addr, void **addr, int size );
```

The caller must specify the physical address of the device in `phys_addr`, and the `size` of the memory region in bytes (rounded up to `PAGE_SIZE` boundaries). The caller can also use `*addr` to specify the desired logical address to map the physical region. If the logical address is not known ahead of time, then passing `*addr = NULL` will automatically choose the next available free location. The chosen memory address will be returned in `*addr`. A region identifier for this newly created region is returned in `region_id`.

The previous memory mapping and allocation routines allowed the caller to specify a page attribute for each memory page. Device register mappings have more stringent requirements: they must be mapped as cache-inhibited to make sure that accesses truly hit the device, and not just the cache. `vm_map_device()` sets all of its page attributes to `CACHEINHIBIT_BIT | VALID_BIT`.

```
int vm_unmap_device( int region_id );
```

The `vm_unmap_device()` system call allows device register mappings to be removed from the callers address space. The region identifier `region_id` specifies the memory region to remove.

3.8.7 Virtual memory initialization

When the kernel boots, initial execution of the system happens within the physical memory space, no memory translation is done. When the virtual memory module gets a chance to initialize, it initializes the lower layers, and then begins to setup the supervisor memory space. During this setup phase, translation is disabled until the whole supervisor translation table is built.

The lower layers of the virtual memory system include the rawpage module and the map module. These modules are simply initialized with the `rawpage_init()` and `map_init()` calls.

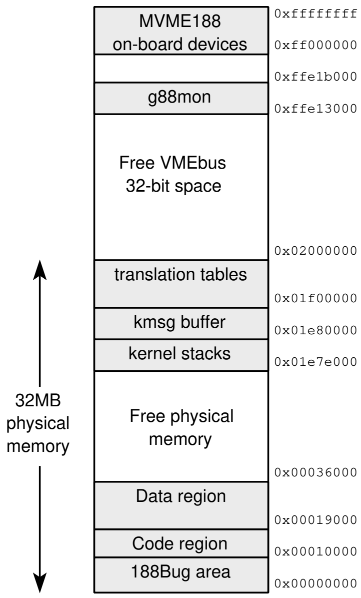

Figure 3.9: Initialized supervisor memory space.

To begin building the supervisor memory space, the `map_alloc()` call is used to create the supervisor code and data address space maps: `kernel_code_map` and `kernel_data_map`. The address space is then built using the `map_page()` call. Figure 3.9 shows the layout of a fully initialized supervisor memory space.

3.9 Task management

A task encapsulates the virtual memory space and processor allocation functions for user level programs. The virtual memory space for a task includes program code, data and heap regions, as well as shared regions and memory mapped device registers. The virtual memory module manages most of a task's address space requirements: address space allocation and deallocation. All other memory operations, such as page-wise dynamic allocation and mapping, are handled directly by the virtual memory module.

The `task.c` module implements the bulk of the task management functionality. This module works in cooperation with all of the other main system components, and ties them together to form a basis for user level development: memory management, task scheduling, interrupt and exception handling, and system call handling. The table in Figure 3.10 summarizes the system call interface exported to the user level by this module.

Synpopsis	Interface availability	Description
`task_create()`	user application	Creates a task.
`task_destroy()`	user application	Destroys a task.
`task_suspend()`	user application	Suspends the execution of a task.
`task_resume()`	user application	Resumes the execution of a task.
`task_info()`	user application	Returns task state information
`task_signal()`	user kernel	Sends an asynchronous signal message to a task.
`task_timer_event()`	user kernel	Registers a task for a timer event.
`task_request_cpu()`	user kernel	Requests a processor for the task.
`task_relinquish_cpu()`	user kernel	Relinquishes control of the cpu to another task.
`task_intr_relinquish()`	user kernel	Relinquishes control to an interrupt driver task.
`task_cleanup()`	user kernel	Cleans up an exiting task.

Figure 3.10: `task.c` module system call summary.

This section describes the task management module, and how it interacts with the rest of the system. Before describing the details behind task management, we first present an overview of the task scheduling environment. This discussion is intended to give the reader a global view of how and why scheduling decisions are made.

3.9.1 Task management overview

All tasks are created using the `task_create()` system call. Each task in the system is allocated its own task descriptor from a global descriptor table, `task_table[]`. The task descriptor contains all the vital information which describes the task, such as name, entry point, and virtual memory space. In addition, the task descriptor contains two pieces of information which assist in the task processor allocation: `num_cpus`, the number of processors currently allocated to the task; and `num_ready_threads`, the number of ready threads in the task waiting for a processor.

When a task receives a processor, the `num_cpus` field is incremented. When a processor is taken away from a task, `num_cpus` is decremented. Likewise, when the user level thread scheduler places a thread on its ready queue, `num_ready_threads` is incremented. When a thread is removed from the ready queue, `num_ready_threads` is decremented.

The task scheduling mechanism uses both of these variables to help decide how a task should be scheduled. There are seven main sources of control in the system which invoke the task scheduler to make a scheduling decision:

1. Processor requests from user programs. When a user level thread scheduler finds itself with ready threads on hand (i.e., `num_ready_threads > 0`), then it will request a processor.

2. Processor relinquishment from user programs. When a user level thread scheduler runs out of runnable threads (i.e., `num_ready_threads == 0`), then it will relinquish its processor.

3. Hardware interrupt handlers. An interrupt may cause an upcall event to be directed at a specific task, in which case the task must be given a processor.

4. Hardware exception handlers. As with interrupt handlers, an exception may occur that requires the attention of a specific task[2].

5. When creating a new task. A newly created task begins with `num_ready_threads == 1`, `num_cpus == 0`.

6. Resuming a suspended task. A suspended task, when resumed with `num_ready_threads > 0`, is in need of at least one processor.

7. After a task is destroyed. When the currently running task is destroyed, another task is scheduled in its place.

[2] For example, an external pager task would be invoked whenever a task generates a page fault.

Tasks that are in need of a processor, when `num_ready_threads > 0`, are either placed on the task ready queue to be given a processor in the future, or are given a processor immediately. The task ready queue is a centralized priority queue with 32 levels. All processors in the system share the same queue through a simple three routine interface: `enqueue_ready_task()`, `dequeue_ready_task()`, and `remove_ready_task()`.

The main scheduling loop within the kernel is in the routine `sched()`. When invoked, `sched()` dequeues the next available ready task and runs it. If there are no ready tasks, then the processor drops into the idle() loop. The idle loop simply runs a forever-loop, with interrupts enabled. So rather than having idle processors poll system queues looking for work, work is delivered to idle processors using the software interrupt service (SWI). When work becomes available, an interrupt is delivered to the next available idle processor. For example, if a task is in need of a processor, then one of the idle processors is delivered a `TASK_RUN_SWI` interrupt. (A global structure, `idle_cpus[]`, keeps track of which processors are idle.)

One of the main task scheduler routines that is responsible for delivering work to idle processors is the `enqueue_ready_task()` routine. Whenever a task is found to require an additional processor, `enqueue_ready_task()` is called to place the task on the ready queue. However, if an idle processor is available, `enqueue_ready_task()` will deliver the processor a `TASK_RUN_SWI` interrupt.

3.9.2 The `KERNEL_INFO` structure

The kernel maintains three important data structures to manage the scheduling of tasks: the `KERNEL_INFO` structure; the task descriptor, `TASK`; and the shared region structure, `SHARED_REGION`. This section focuses on `KERNEL_INFO`; the following section discusses the latter two.

The `KERNEL_INFO` structure, shown below, contains a number of fields that are used throughout the kernel, and at the user level. At boot time, a physical memory page is allocated to store the structure. The page is mapped into the kernel space with read/write privileges, and is accessed through the global variable `kernel_info`. When a task is created, the page is mapped read-only into the tasks address space. All user level code has read access to the information, but it cannot be overwritten.

```
typedef struct
{
    int simulator;       /* nonzero when executing under the 88k simulator */

    unsigned long timer_ticks;          /* clock ticks since boot time */

    char idle_cpus[NUM_CPUS];           /* indicates which cpus are idle */

    unsigned long idle_time[NUM_CPUS];   /* idle time counters for each CPU */
    unsigned long kernel_time[NUM_CPUS]; /* average time in kernel recently */
    unsigned long user_time[NUM_CPUS];   /* average time in user space */

    int page_table_entries;             /* number of free pte entries */
    int physical_pages_free;            /* usable free memory */
    int num_free_regions;               /* free VM regions */
    int num_tasks;                      /* tasks created in system */
    int num_ready_tasks;                /* number of ready tasks (note: */
                                        /* protected by ready_q_lock) */

    /* stores which cpus tasks are running on. */
    /* given a task id, it's easy to find which cpus it's running on */
    char run_cpus[TASK_TABLE_SIZE][NUM_CPUS];

    int lock;
} KERNEL_INFO;

KERNEL_INFO *kernel_info;         /* allocated and mapped at boot time */
```

The structure contains many miscellaneous fields, and a few important ones. `simulator` is set at boot time to signify whether the system is running under the g88 simulator, or on the real hardware. Sometimes, as when dealing with devices, it is important to know this difference. `timer_ticks` is a counter that is incremented at each system clock tick. This can be used to give programs a notion of time.

Many of the fields in this structure are used for resource usage accounting. A user level Unix-style `ps` command could display this information, or log it to a file. The `idle_time[]`, kernel_time[], and user_time[] fields are used to record system activity for each processor. The next five fields are used to count other system resources, such as free memory and the number of tasks running in the system.

The idle_cpus[] field is used to record the idle state of each processor in the system. When a processor enters its idle loop, it sets the associated entry in the idle_cpus[] array. Another processor in the system can read this field and immediately know which processors are idle and are eligible for work. For example, if a new thread is created at the user level, the thread scheduler can read idle_cpus[] and quickly spawn the thread to a remote idle processor by delivering a software interrupt message to the idle processor (via intr_remote_cpu().

The run_cpus[][] field records the running status of each task in the system. When a processor is given to a task, the task's run_cpus entry is set for that processor. This allows the scheduling code to quickly find out which processors are running a specific task. This can be used when destroying a task, for example. The task_destroy() code consults the run_cpus entry for the task, and broadcasts an interrupt to the processors running that task. As another example, the IPC mechanism can easily target its messages to processors that are running the destination task.

3.9.3 Task descriptors

The task descriptor structure, type TASK defined in kernel.h, contains most of the bookkeeping data that comprises a task. The structure is divided into several main components, each of which is maintained and shared between the various kernel modules. The descriptors are private to the kernel memory space; it is not directly shared by user level programs.

```
typedef struct
{
    int         id;             /* 0 to TASK_TABLE_SIZE-1 */
    int         lock;           /* spin lock for this descriptor */
    void        *stack_page;    /* kernel address of user's upcall stack*/

    int         interrupts;     /* bit field of enabled user interrupts */
```

```
    int         exceptions;     /* bit field of enabled user exceptions */
    int         pending_intr;   /* remembers pending interrupt vector   */

    TASK_INFO   info;           /* task statistics and other info */
    TASK_VM     vm;             /* task virtual memory spaces */
    SHARED_REGION *sr;          /* user/kernel shared region pointer    */

    QUEUE_LINK  sched_link;      /* scheduler queue */
    QUEUE_LINK  timer_link;      /* timed event queue */
    unsigned int wake_time;      /* time to wake this task (task_timer) */
} TASK;
```

The following paragraphs and subsections describe each component of this structure. After starting with the smaller miscellaneous fields, the larger component structures are described.

The `id` field identifies the task descriptor. This number is the index value of the descriptor in the `task_table[]` array. It is initialized at boot time, and remains constant throughout the life of the system.

The `lock` field is the spin lock used to provide mutual exclusion in routines that manage the task descriptor. Normally this lock is acquired before any system call or scheduling action is performed on a task. However, there are cases where locking is not required. These cases are defined by their specific purposes.

The `stack_page` field points to the physical page of memory that is allocated for the tasks upcall stack. When a task is created, a user level upcall stack is allocated and mapped into the user level memory space at location `TASK_STACK_ADDR`. When the task is destroyed, the upcall stack is returned to the physical memory pool.

The `interrupts` and `exceptions` fields are maintained by the kernel interrupt and exception dispatchers. `interrupts` and `exceptions` are bit fields which describe the interrupt and exception handling capability for the task. For example, when a task registers an interrupt handler in its address space, the corresponding bit is set in the `interrupt` field. The kernel interrupt dispatcher can use this information for deciding what to do when a particular interrupt arrives.

The interrupt dispatcher uses `intr_pending` to remember a pending interrupt. When a hardware device generates an interrupt, the kernel interrupt dispatcher looks up in its table where to locate the handler for the interrupt. If the handler resides in a task that is currently not active, it must be switched in. The `intr_pending` field is set in the old task to remember the pending interrupt vector. When the old task is switched out and the interrupt handler task is switched in, the vector is examined before upcalling to the new task.

The `sched_link` field is the main scheduler queue link for the task. This link is used to place the task on the task descriptor free queue and ready queue. The `timer_link` field is used by the task event timer system. A task enqueues itself on the `task_timer_q` using this link, and will be issued a timer upcall event when the `wake_time` interval expires.

`TASK_INFO` structure

The first main component of the task descriptor, `info`, contains general information about the task such as scheduling `state`, `priority`, an ASCII string `name` and resource consumption statistics. The reason why these data fields are grouped in their own structure is so that they can easily be marshalled to user level programs. A special system call, `task_info()`, allows user level programs to query this structure for information collecting. For example, the `console` program implements a Unix-style `ps` command for displaying task scheduling and resource usage. Currently, the resources kept track of are memory usage, but this could be expanded to include other interesting information such as interrupt and context switching rates.

```
typedef struct
{
    int state;                  /* task scheduling state */
    int priority;               /* 0 to NUM_READYQ_PRIORITIES-1 */
    char name[TASK_NAME_SIZE];  /* ASCII string name for executable */

    /* some memory statistics */
    int code_size;              /* size of the program code (bytes) */
    int data_size;              /* size of the program data (bytes) */
    int bss_size;               /* size of the program bss (bytes) */
    int virtual_size;           /* pages of memory mapped by this task */
} TASK_INFO;
```

TASK_VM structure

The next component, vm of type TASK_VM, maintains information about the tasks' address space. This structure is managed by the memory management system. The code and data address map descriptors, and allocated virtual memory region descriptors are stored here.

```
typedef struct
{
    MAP code_map;          /* map descriptor for code map */
    MAP data_map;          /* map descriptor for data map */
    int code_region_id;    /* region id for code memory */
    int data_region_id;    /* region id for data memory */
    QUEUE device_regions;  /* region list for mapped devices */
    QUEUE heap_regions;    /* region list for allocated memory */
    int heap_hint_addr;    /* hint address for next heap allocation */
} TASK_VM;
```

The map descriptors code_map and data_map are used by the map module to perform address translations for the task. When a task is created, the maps are allocated, and region descriptors are allocated for the executable code and data areas. When a task is scheduled to run, the map descriptors are passed to the address map activation routine.

The two region queues, device_regions and heap_regions, are used to remember region descriptors that are dynamically allocated by the user throughout the life of the task. This provides an easy way to free all of a tasks memory when it is destroyed: the region lists can be efficiently traversed and the descriptors freed. The heap_hint_addr field is used to remember to next free location in the tasks virtual address space for a heap or device memory allocation to occur. The hint helps reduce the amount of searching through the task translation table structures to find free contiguous regions.

SHARED_REGION structure

The sr field is a pointer to the tasks' user/kernel shared memory region. The shared region structure, SHARED_REGION, contains information that is commonly accessed by both the user and supervisor address spaces for scheduling decisions and other operations. This helps reduce communication costs between the user and kernel spaces.

When a task is created, a physical memory page is allocated for the shared region structure using `rawpage_alloc()`. The page is mapped read/write into both the user and supervisor address spaces. The memory location of the page in the supervisor address space is remembered by the `sr` field. In the user address space, the page is mapped at the well known address, `USER_SHARED_REGION_ADDR`. The user level kernel code can use this constant to refer to the structure.

```
typedef struct
{
    /* parameter passing area for system calls */
    int syscall_parms[NUM_CPUS][SYSCALL_PARMS_SIZE/4];

    int num_cpus;                /* number of cpus assigned to this task */
    int num_ready_threads;       /* number of ready threads in task */

    void *upcall_addr;           /* user upcall dispatch address */
    int upcall_status[NUM_CPUS]; /* upcall dispatcher events */
    int intr_status[NUM_CPUS];   /* interrupt dispatcher events */
    int excp_status[NUM_CPUS];   /* exception dispatcher events */

    int num_signals;             /* number of signals in list */
    int signal_head;             /* signal list head index */
    int signal_tail;             /* signal list tail index */
    int signal_lock;             /* signal list spin lock */
    SIGNAL signals[NUM_SIGNAL_ENTRIES];

    /* global semaphore links */
    GLOBAL_SEM_LINK sem_links[NUM_GLOBAL_SEMS];

    int argc;                     /* number of command line arguments */
    char *argv[NUM_ARGV_VECTORS]; /* array of argv vectors */
    char argv_buf[ARGV_BUF_SIZE]; /* buffer for storing parameters */
} SHARED_REGION;
```

The `syscall_parms` field provides a buffer for passing out-of-line parameters through system calls. For example, when a user calls `kprint()`, the string parameter is copied into the system call buffer. The kernel `kprint()` routine can then access the string parameter from the buffer. Ideally, the kernel should be able to access the user's parameters directly. However, most of the kernel system calls are implemented in C, a language which does not easily permit references to remote address spaces. So kernel system calls which pass parameters by reference must use this indirection. Each processor running a task uses its own parameter buffer, so that system calls can safely execute in parallel. Thus, the `syscall_parms` field allocates enough buffers for every processor in the system.

The `num_ready_threads` field is maintained by the user level threading kernel to count the number of runnable threads available. `num_cpus` is maintained by the kernel to count the number of processors currently allocated to the task. These fields are used by both the thread and task schedulers as the basis for processor allocation decisions.

The `upcall_addr` field is used by the kernel upcall dispatcher as the address to call in the user space for upcall events. Initially, when a task is created, this value is set to contain the entry point for the executable program. This entry point corresponds to the user level kernel initialization code. When the task runs and initializes itself, it can easily change this value to any address within its address space, such as the user level upcall dispatcher.

The `upcall_status`, `intr_status`, and `excp_status` are per-processor fields that are used by the interrupt, exception, and task upcall dispatching mechanism to signal various events to the user kernel. Each field maintains a bit field of possible flags, each flag is set to specify a certain event, or combination of events.

The set of `signal` fields belong to the asynchronous task signalling mechanism. Any task can send a simple signal message to another task using the `task_signal()` system call interface. These fields are used to implement a FIFO queue for a fixed number of signals destined for a task.

The `sem_links` field stores an array of queue links for the global semaphore service. Each link in the array corresponds to a global semaphore entry. When a user thread blocks on a global semaphore, the thread's task is queued on the global semaphore's queue. When the semaphore is signalled, the next task in the queue is dequeued and issued a signal.

The `arg` fields are used to supply Unix-style command line argument information for user programs. When a task is created, one of the parameters to `task_create()` is a user argument structure. `task_create()` copies the user arguments into these fields. When the user level initialization routine executes, the `argv` vector can be built and passed to the user's `main()` thread.

3.9.4 Task descriptor table

Following the general policy of static allocation where possible, the `task_table[]` array stores all of the system task descriptors. The size of the table is defined at compile time. The table is declared in the following fashion in `task.c`:

```
TASK  task_table[TASK_TABLE_SIZE];
QUEUE task_free_q;                  /* free descriptor queue */
int   task_free_q_lock;             /* spin lock for free queue */
```

The table and `task_free_q` are initialized at boot time. Each of the task descriptors are linked together into a free queue. The free queue allows for O(1) allocation and deallocation of task descriptors.

3.9.5 Upcalling to a task

A task is given a processor for a number of reasons. Each reason is termed an `upcall event`, and there are eight different upcall events defined in `upcall.h`. These upcall events are used in various places throughout the kernel:

```
#define RUN_UPCALL        0              /* task can execute now        */
#define TICK_UPCALL       1              /* timer tick upcall           */
#define RELINQUISH_UPCALL 2              /* task relinquish request     */
#define TIMER_UPCALL      3              /* timed event has expired     */
#define SIGNAL_UPCALL     4              /* task signal is pending      */
#define KILL_UPCALL       5              /* task destroy upcall         */
```

```
#define EXCP_UPCALL     6                   /* a user exception occurred */
#define INTR_UPCALL     7                   /* a user interrupt occurred */
```

Each of these upcall events correspond to a bit in the `upcall_status[]` field in the task's shared region. These events are described in greater detail in Section 4.1. When an upcall is issued to a task, the particular event is bit-wise or'ed into the `upcall_status[]` entry for the processor. Previous bits in the entry are therefore retained, and can be accumulated if many upcall events are directed to a task before the task actually gets a chance to execute. At the user level, the upcall dispatcher reads this bit-field and executes the appropriate upcall handlers, clearing the event bits as the handlers complete.

For example, when `sched()` finds a ready task to run, it issues a `RUN_UPCALL` to the task in the following manner:

```
curr_task->sr->upcall_status[cpu] |= 1 << RUN_UPCALL;
upcall_now(curr_task->sr->upcall_addr);     /* no return */
```

The `upcall_now()` routine, from `excp_dispatch.s`, is an assembler code routine that performs the upcall action into user space. Given the user level destination address, `upcall_now()` sets up the processor context, and executes a "return from exception" `rte` instruction.

Before a task can be issued an upcall, its memory space must be activated on the local processor. Also, if the task resides on the ready queue, it may have to be dequeued. The `task_upcall()` routine performs all of the bookkeeping chores to enable a task and upcall into its address space. This routine is used in various places in the kernel to facilitate upcall dispatching.

```
void task_upcall(int cpu, TASK *task, int upcall_vec);
```

The task to activate is specified by `task`, and the upcall event is specified in `upcall_vec`. The task will be removed from the ready queue if `num_ready_threads <= 1`, and its state set to `TASK_RUNNING`. A task is not removed from the ready queue if `num_ready_threads > 1`, which means that the task is eligible for more processors than the one it is currently receiving.

The task's `num_cpus` field is incremented and its `run_cpus[][]` entry is set to indicate which processor the task is running on. Finally, the `map_enable_task()` call activates the code and data memory space, and `upcall_now()` is invoked to pass execution to the user level.

3.9.6 Creating a task

The `task_create()` system call is used to create a task. First, a free task descriptor is dequeued from the `task_free_q`. Then, virtual memory call `vm_task_alloc()` is used to allocate the task's address space and create the shared region storage area. The shared region fields are then initialized. The `syscall_parms` buffer is used to pass the task's `id` and the region identifier for the nameserver to the user initialization routine. The task is either placed on the ready queue to be run, or left in the suspended state to be later resumed.

```
int task_create(int *task_id, int priority, int ready, TASK_EXEC_HDR *hdr,
                int code_region, int data_region, TASK_ARGS *args);
```

The `priority` parameter specifies the task's run priority level, an integer between 0 and 31. Priority 0 is low priority, priority 31 is high priority. A high priority task always receives processors before a low priority task. If `ready` is passed a nonzero value, the new task will be immediately readied and executed on an idle processor, or placed on the ready queue if all processors are busy. If `ready` is zero, the task will be placed in the `TASK_SUSPENDED` state, to be later resumed by `task_resume()`. `task_id` returns the created task's descriptor identifier. `task_create()` returns `OK` if task creation was successful, or `FAILED` if an error occurs.

The `hdr` parameter passes the executable header information. This structure contains the executables code, data, and bss segment addresses and sizes. It also includes the executable entry point. The caller is responsible for querying the filesystem for this information.

The `code_region` and `data_region` parameters pass the executable code and data regions. Before the user calls `task_create()`, the executable code and data must be read from the filesystem into two regions. These regions, plus the `hdr` information, are used by the `vm_task_alloc()` routine to allocate the memory space for executable. In `vm_task_alloc()`, once the address maps are created, `vm_share()` is used to map the code and data regions to the tasks address space. The caller to `task_create()` can then free the code and data regions from their address space, or use them for additional calls to `task_create()`.

The `args` parameter points to an argument structure of the following format. The user must properly initialize this structure before calling `task_create()`:

```
typedef struct
{
    int nameserver_id;              /* region id of nameserver */
    int tty_id;                     /* port id of the tty connection */
    int argc;                       /* user argument count */
    int argv_len;                   /* length in bytes of argv_buf */
    char argv_buf[ARGV_BUF_SIZE];   /* user parameters */
} TASK_ARGS;
```

The `nameserver_id` field is used to pass the nameserver region identifier to the newly created task. When the task initializes, it can use this identifier in `vm_share()` to map in the nameserver. The `tty_id` is used by the user level `tty_driver.c` server to pass on the `tty` connection identifier. When the `tty` server creates a task, a stdin/stdout port connection is created for the task to communicate with the server. When the task initializes, it uses the `tty_id` port to communicate with the server.

The `argc` field is set to specify the number of command line parameters. `argv_len` is set to specify the length in bytes of the parameters in the `argv_buf` field. `argv_buf` is built to contain a contiguous list of null-terminated parameter strings. When the task gets a chance to run, the user level initialization routine uses this information to construct an appropriate Unix-style argv list to pass to the user's `main()`.

3.9.7 Destroying a task

```
int task_destroy(int task_id);
```

Whether a task simply finishes its own execution, or whether a remote task must be killed, `task_destroy()` does the job. `task_destroy()` is responsible for locating the specified task, killing it, and returning its resources back to the system.

On a multiprocessor system, the effort required to kill a task is greater than on a uniprocessor system. The additional difficulty arises when the task to destroy is executing on remote processors. In this case, the remote processors must be interrupted and notified that the task they are executing will be destroyed:

```
/* check if there are other cpus running the task */
if ( dead_task->sr->num_cpus > 0 )
{
    /* interrupt each cpu currently running this task */
    intr_remote_cpus(kernel_info->run_cpus[task_id], TASK_DESTROY_SWI,
                     task_id);
}
```

The remote interruption facility is provided by the `intr_remote_cpus()` call in `interrupt.c`, which in this case, delivers a `TASK_DESTROY_SWI` software interrupt to all processors executing the specified task. The SWI interrupt handler on the remote processors recognize this interrupt and discards the dying task.

However, stopping execution and cleaning up the task's memory space and kernel data structures is not enough to completely remove a task from the system. There are also data structures at the user level which must be cleaned up. For example, if a task has created and is the owner of user level IPC ports, the ports must also be properly terminated. Otherwise, the user level port data structures will be left with stale entries in their descriptor tables. As another example, suppose a task has established a communication channel with a server. The task must be given the chance to gracefully sever the connection with its server.

To offer a graceful shutdown for tasks which are to be destroyed, before the kernel frees the tasks resources, a `KILL_UPCALL` is delivered to the task. The `KILL_UPCALL` handler executes at the user level, and performs any cleanup that is necessary before returning to the kernel to be officially destroyed. While this is taking place, the task's state is set to `TASK_DYING`, indicating that the task should not be scheduled by remote processors. Figure 3.11 demonstrates this sequence of events.

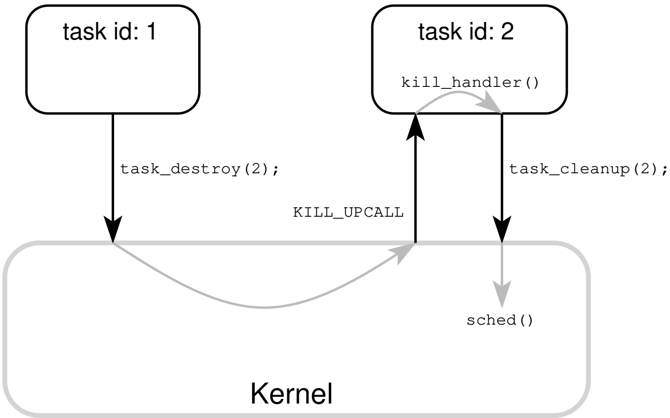

Figure 3.11: Destroying a remote task.

When the `KILL_UPCALL` handler has finished cleaning up the user level state, the system call `task_cleanup()` is used to return control to the kernel. `task_cleanup()` frees all of the task's memory, returns its descriptor to the free list, and jumps into the scheduler loop `sched()` to continue processing other tasks.

If a task is voluntarily killing itself, then `task_destroy()` assumes that the user level code has already cleaned up its data structures, and therefore dispatching a `KILL_UPCALL` is not necessary.

3.9.8 Suspending a task

The act of suspending a task in a multiprocessor system has the same problem as destroying a task: the task may be executing on a remote processor. As with destroying a task, the `intr_remote_cpus()` call can deliver a `TASK_SUSPEND_SWI` interrupt to the appropriate processors. Upon receiving the interrupt, the SWI handler invokes a `RELINQUISH_UPCALL` event into the task. At the user level, the `RELINQUISH_UPCALL` handler places the currently executing thread back on the thread ready queue, and relinquishes control of the processor back to the kernel, where another task is scheduled in its place.

```
int task_suspend(int task_id);
```

The `task_suspend()` call suspends execution of the specified task. If the task is currently executing on a remote processor, the processor will be interrupted and forced to schedule another task. The tasks state is set to `TASK_SUSPENDED`. Returns `OK` if successful, or `FAILED` if the specified task is invalid.

3.9.9 Resuming a task

```
int task_resume(int task_id);
```

The `task_resume()` call resumes execution of a previously suspended task, specified by `task_id`. If an idle processor is available, a `TASK_RUN_SWI` interrupt will be delivered to it, and the resumed task will be executed there. If all processors are busy, then the task is placed on the task ready queue where it will be scheduled sometime in the future. Returns `OK` if successful, or `FAILED` if the specified task is invalid.

3.9.10 Task states

The task descriptor state field, `info.state`, maintains the scheduling state of a task. There are six possible scheduling states, defined in `kernel.h`:

```
/* possible states for a task descriptor */
#define TASK_FREE       0
#define TASK_SUSPENDED  1
#define TASK_READY      2
#define TASK_RUNNING    3
#define TASK_IDLE       4
#define TASK_DYING      5
```

Figure 3.12 summarizes the possible transitions between each state. The use of each state is described as follows:

- `TASK_FREE`: Is the initial state for all tasks. The task descriptor is queued on the free queue, `task_free_q`, using the tasks `sched_link`.

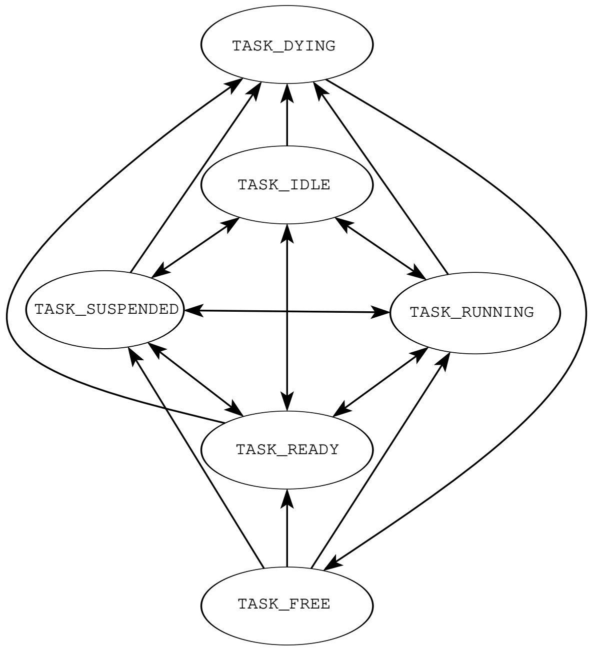

Figure 3.12: Task state transitions.

- `TASK_READY`: The task currently resides on the ready queue. The task also may be executing on at least one processor.
- `TASK_RUNNING`: The task is executing on at least one processor.
- `TASK_IDLE`: The task is not executing anywhere, and does not require any processors.
- `TASK_SUSPENDED`: The task has been suspended. It will receive no processors.
- `TASK_DYING`: The task is being removed from the system. The state will progress to `TASK_FREE` when the tasks resources have been properly cleaned up.

3.9.11 Task ready queue

Tasks that require a processor are placed on the task ready queue. The file `readyq.c` provides a simple interface to a globally shared round-robin priority queue with 32 levels. All accesses to the task ready queue are done through three routines: `enqueue_ready_task()`, `dequeue_ready_task()`, and `remove_ready_task()`. As long as this interface remains the same, the implementation of the ready queue can change.

The ready queue enforces priority levels between 0 and 31. Priority level 0 is low, level 31 is high. A higher priority task will receive a processor before a lower priority task.

Enqueue and dequeue operations are O(1) in the most frequent case. This efficiency is achieved by the use of 32 independent FIFO queues, rather than a single sorted priority list. Figure 3.13 displays a typical ready queue layout. Using a sorted priority list would guarantee an O(1) dequeue operation because the first element on the list is always the highest priority task. However, the enqueue and remove operation requires a sort operation to rebuild the list, which for a heapsort structure, requires O(n log(n)) access time (not including extra code complexity overhead).

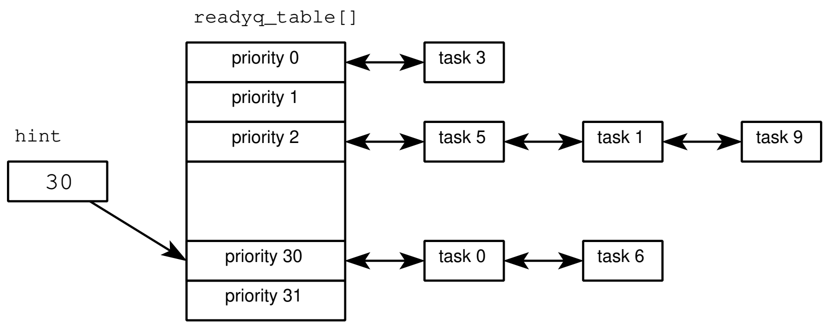

Figure 3.13: Task ready queue structure.

```
QUEUE readyq_table[NUM_READYQ_PRIORITIES];
int hint;
int ready_q_lock;
```

The `readyq_table[]` maintains a statically allocated array of FIFO queues. Each queue is used to store task descriptors of a single priority level. A single spin lock, `ready_q_lock`, is used to protect the queue from concurrent accesses. The `hint` integer is used to cache the highest priority non-empty FIFO queue, and is used to speed up the dequeue operation.

```
TASK *dequeue_ready_task();
```

The `dequeue_ready_task()` operation dequeues the next available ready task and returns its descriptor. The routine consults the `hint` variable to find the highest level priority with a non-empty queue. If the `hint` level queue is empty, a linear search counts down the `hint` until the next non-empty queue is reached. The task is dequeued and returned to the caller.

```
void enqueue_ready_task(TASK *task);
```

The `enqueue_ready_task()` operation tries to find an idle processor to execute the specified task. If all processors are busy, then the task descriptor's state is set to `TASK_READY` and is placed on the end of the appropriate queue. If the task's priority is higher than the `hint` variable, then the `hint` is updated to this priority value.

```
void remove_ready_task(TASK *task);
```

A task that resides in the ready queue structure may need to be removed for reasons other than a processor becoming available to run the task. Sometimes a task must be removed out of order, from the middle of the queue. For example, a task that is being destroyed or suspended must be removed from the ready queue. The task descriptor to remove from the ready queue is specified by the `task` parameter.

3.9.12 Task signalling

The task signalling mechanism provides a primitive building block for the construction of user level communication services. A signal is a simple non-blocking one-way message delivered from one task to another. The `task_signal()` system call performs two operations: wakes up a destination task, and delivers it a message. The following structure, from `shared_region.h`, defines a signal message:

```
typedef struct
{
    int signal;        /* signal message type */
    int data;          /* message type-defined data */
} SIGNAL;
```

Any task in the system can be the recipient of multiple signal messages from multiple sources. To ensure the non-blocking, reliable delivery of a signal message, a FIFO queue of signal descriptors is allocated to buffer messages. The queue is stored as `signals[]`, in the shared region structure of each task descriptor. Signals are enqueued by the kernel, and dequeued by the user level signal dispatcher as a result of a `SIGNAL_UPCALL`.

In the event that many senders bombard a single task with signals, the signal queue of the destination task may become filled. Further signal messages will be discarded. Thus the delivery of signals is not totally reliable. However, reliability can be ensured for a bounded set of interactions given a reasonable queue size. The signal queue fits within the task's shared region page, where it has room for about 400 entries, at 8 bytes each.

Each task contains a set of signal handlers as part of its user level upcall dispatch tree (c.f. Figure 4.20). One signal handler routine is used for each signal message type. Originally, the signal mechanism was designed to allow handlers to be added and removed dynamically, allowing user programs to add their own signal handlers at run-time. However, a static scheme was implemented for simplicity. The system currently supports six different signal message types:

```
#define GLOBAL_SEM_SIGNAL_SIGNAL       0   /* global semaphore signal */
#define GLOBAL_SEM_DESTROY_SIGNAL      1   /* global semaphore destroy */
#define GRPC_PORT_SEND_SIGNAL          2   /* global rpc send handler */
#define GRPC_PORT_REPLY_SIGNAL         3   /* global rpc reply handler */
#define GRPC_PORT_DESTROY_SIGNAL       4   /* global rpc destroy handler */
#define GLOBAL_PORT_SEND_SIGNAL        5   /* global port send handler */
```

These particular signals are used by the user level IPC and semaphore library to synchronize their cross-address space communication events. For example, when a client invokes an RPC message to a server, the server must be informed of the waiting message. In this case, the client RPC library would issue a `GRPC_PORT_SEND_SIGNAL`, specifying the port identifier in the signal's `data` field.

Delivery of a signal

The user level can deliver a signal to a task by two means. The user level kernel determines which of these two methods is most appropriate:

1. Issue a `TASK_SIGNAL_SWI` interrupt to a processor:

   ```
   intr_remote_cpu(cpu_list, TASK_SIGNAL_SWI, task_id,
                   GRPC_PORT_SEND_SIGNAL, port_id);
   ```

   If the destination task is executing on a remote processor, or if there is an idle processor, then deliver an interrupt to that processor. The `SWI_service()` handler on the remote processor will queue the signal and dispatch an upcall to the destination task. It is possible that the remote processor can become busy before the interrupt is actually sent. In this case, the `SWI_service()` handler will properly dispatch the signal to the destination task and return to the existing work.

2. Issue a `task_signal()` system call:

   ```
   task_signal(task_id, GRPC_PORT_SEND_SIGNAL, port_id);
   ```

   If the destination task is not executing anywhere, and there are no idle processors, then the kernel level system call `task_signal()` must be invoked on the local processor to manually queue the signal and wake the destination task. The destination task is woken by placing it on the ready queue. When the task eventually acquires a processor, its upcall dispatcher is executed.

In either case, when the destination task obtains a processor, the user level upcall dispatcher notices that the `num_signals` field is nonzero, and begins to process the signals. After each signal entry is dequeued by the dispatcher, the `signal` field is used to execute the appropriate handler. The signal handler can then perform the desired operation using the supplied `data` field. The user level IPC and semaphore libraries contain handlers for each of the above signal types.

3.9.13 Task timer service

The timer service allows tasks to register for a timed event. User level schedulers can use this service to implement a sleep facility. A list of tasks, `timer_q`, sorted by their wakeup time, is maintained by the `task_timer.c` module. This list is shared with the system clock interrupt service routine, `DTI_service()`, to ready tasks with expired wakeup values. The `timer_q_lock` protects against concurrent accesses to the list. When a timed event expires, the associated task is readied and delivered a `TIMER_UPCALL` event.

```
QUEUE timer_q;          /* list of tasks waiting for a timer event */
int   timer_q_lock;

int task_timer_event( unsigned long wake_time );
```

`task_timer_event()` is called by user level thread schedulers to register for a timed event. The `wake_time` field specifies the future wakeup time. By using the current system clock time, provided by `kernel_info->timer_ticks`, the future wakeup time can be calculated by adding the desired number of clock ticks. The clock tick interrupt rate is determined at boot time, and can be configured to any value within the hardware limits: 8.6 microseconds to 563 milliseconds.

Only one timed event is registered per task. Other timed events are queued by the user level sleep service. Registering an earlier timed event will reset the previous setting to use the nearer value.

The task descriptor field `timer_link` is used to link the task into the `timer_q`. A linear search through the `timer_q` is done to locate the proper waketime slot.

3.10 Kernel management for global semaphores

The task signalling mechanism provides a way for tasks to send low level event messages to each other. This service can be used to notify remote tasks that an event has occurred. Global semaphores, on the other hand, provide a way for threads in a task to *wait* for a remote event to occur. This service can be used as building block for higher level communication services.

Synpopsis	Interface availability	Description
`kernel_sem_enqueue()`	user kernel	Enqueues the caller task onto the semaphore.
`kernel_sem_dequeue()`	user kernel	Dequeues the next task from the semaphore.
`kernel_sem_destroy()`	user kernel	Dequeues all tasks from the semaphore.

Figure 3.14: Kernel level global semaphore support routines.

For example, a client thread wishing to send a message to a server may have to wait for a free port buffer to become available. The client thread performs a `sem_wait()` on the port queue, and when the server releases a buffer, it issues a `sem_signal()`. The client thread is then unblocked to dequeue a free port buffer and send its message.

While the user level global semaphore library implements the call `sem_wait()` and the call `sem_signal()`, some kernel level assistance is required to preserve the ordering of `sem_wait()` calls. Threads must be unblocked by `sem_signal()` in the same order that they were blocked by `sem_wait()`. Threads from multiple tasks may be blocked on the same semaphore. So `sem_signal()` must decide which task should be delivered the signal message. To do this, the kernel implements a task queuing service, consisting of three system calls summarized in Figure 3.14.

These calls manage a set of task queues, one queue for each global semaphore. The queues are stored in the `task_sem_q[]` table, declared in `kernel_sem.c`:

```
QUEUE task_sem_q[NUM_GLOBAL_SEMS];
```

When a thread blocks on a global semaphore, the user semaphore library routine calls `kernel_sem_enqueue()` to queue the threads's task descriptor onto the appropriate `task_sem_q[]` entry. When the semaphore is signalled, `kernel_sem_dequeue()` is invoked, which dequeues the task descriptor that contains the blocked thread. The task is then delivered a `GLOBAL_SEM_SIGNAL_SIGNAL` message using `task_signal()`. The task will then be scheduled and its upcall dispatcher will unblock the thread.

In the event that a global semaphore holding blocked threads is destroyed, the `kernel_sem_destroy()` call is invoked to traverse the appropriate `task_sem_q[]` list, and send `GLOBAL_SEM_DESTROY_SIGNAL` signals to each task. Each task will be scheduled and allowed to unblock its threads.

Threads in a task can be blocked on multiple semaphores, so a task descriptor may be queued on several `task_sem_q[]` queues at once. To allow this, each task descriptor contains a list of global semaphore links, `sem_links[]`, in its shared region structure. A task is queued on the same semaphore at most once, so the number of links required equals the number of semaphores. The `GLOBAL_SEM_LINK` is defined as follows in `shared_region.h`:

```
typedef struct
{
    signed int task_id:16;              /* this link's task identifier */
    signed int num_threads_waiting:16;  /* num threads in this task */
    QUEUE_LINK link;                    /* semaphore queue link */
} GLOBAL_SEM_LINK;
```

The `task_id` field identifies the task descriptor that belongs to the link. When `kernel_sem_dequeue()` removes a task from a semaphore queue, the `task_id` field is used to identify which task contains the blocked thread.

The `num_threads_waiting` field counts the number of threads in the task that are blocked on the semaphore. This count is used by the user level semaphore library and the `kernel_sem_dequeue()` routine to decide when a link should be enqueued or dequeued.

3.11 Interrupt management

This section describes the kernel level interrupt handling and dispatching mechanism. Some interrupts are handled directly by the kernel, such as the system clock tick. Other interrupts, such as VMEbus interrupts, can be dynamically registered by tasks to call a user level routine. A table of interrupt handler routines is maintained to keep track of user level and kernel level handlers.

The `interrupt.c` module implements the interrupt dispatcher and user level system calls use to register interrupt handlers. The table in Figure 3.15 summarizes the system call interface exported to the user level by this module.

Synpopsis	Interface availability	Description
`intr_register_user()`	user kernel	Registers the task and enables the interrupt.
`intr_deregister_user()`	user kernel	Disables the interrupt bit and removes handler.

Figure 3.15: Interrupt handler registration system calls.

All interrupts on the 88100 are funnelled through one single exception vector, INT. From there, the low level interrupt handler properly saves the execution context and branches to the interrupt dispatcher. The interrupt dispatcher finds the recipient of the interrupt, and either upcalls into user space to handle it, or calls a local kernel function.

3.11.1 User level preemption

Interrupts preempt user level threads and cause scheduling events to occur. For example, if a timer interrupt occurs, a `TICK_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. Since spin locks are used throughout the user level kernel to protect data structure accesses, this problem must be solved.

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 `lock_count` variable is incremented at the user level. 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 user. But before the interrupt handler 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 branch to the upcall dispatcher. The upcall dispatcher will perform the deferred event(s).

As implemented, 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, as shown in Figure 3.16. 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` was a conventional global variable, then incrementing it would require the use of a spin lock – which cannot be done, since `lock_count` itself is used within the locking routines.

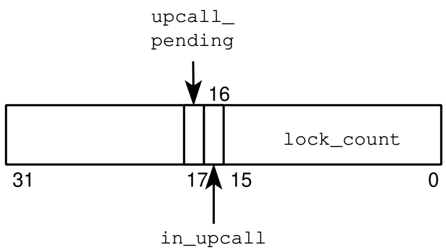

Figure 3.16: Register r28 usage.

A third variable is contained within the `r28` register: `in_upcall`, bit 16. This bit is set to prevent preemption while the user level is processing an upcall event. The interrupt dispatcher checks this bit whenever an upcall is attempted. If the bit is set, then an upcall is already is progress. Upcalls cannot be nested, so the user's context is restored, and control returns back to the point where the interrupt occurred, and upcall processing continues. Any `upcall_status[]` bits that were set as a result of the interrupt will be properly dispatched.

3.11.2 Handling interrupts

Due to the asynchronous nature of instruction execution on the 88100, low level interrupt handling is a fairly complex operation. Certain situations arise where instructions in the data unit and floating point pipeline are not properly completed. The interrupt handler must complete these instructions. It is possible that some of the instructions may cause an exception fault to occur, so the interrupt handler must be able to detect this and handle the exceptions. Faulted instructions in the data unit pipeline must be decoded and completed by simulating the instructions in software.

When a device issues an interrupt, and the processor interrupt disable bit (IND) is not set, the processor stops fetching further instructions and tries to empty its pipelines. The IND bit is automatically set to disable interrupts while the initial interrupt handler executes. Interrupts remain disabled while execution continues throughout the kernel.

After the processor finishes cleaning up its internal state, execution resumes at the INT vector handler in the exception vector table, `excp_vects`. The exception vector table contains 1024 entries, one for each of the possible 88100 exception vectors. The INT vector saves the user's stack pointer in a temporary scratch register, and branches to the kernel interrupt handler `INT_handler()` in `intr_handler.s`:

```
exc1:   br.n    _INT_handler      ; hardware device interrupt
        stcr    r31, cr20         ; cr20 <- user's stack pointer r31
```

The first thing the interrupt handler does is check if the interrupt belongs to g88. g88 uses the ABRT and SWI7 interrupt for its functioning, so any of those interrupts are immediately redirected to the g88mon handler. The ABRT interrupt is set whenever the user presses control-C on the console. The SWI7 interrupt is used to propagate `g88mon` interrupts to all processors. All other interrupts are processed by the kernel.

Interrupts are normally disabled while execution runs within the kernel. The only exception to this rule is when the kernel is executing the idle loop. If an interrupt occurs during the idle loop, no register state needs to be saved because the idle loop does not do anything useful. The interrupt handler checks the processor EPSR[3] register mode bit to see if the processor was in supervisor mode at the time of the interrupt. If so, then the idle loop was running, and the handler can jump directly to the interrupt dispatcher `intr_dispatch_s()` without saving any state.

If execution was interrupted at the user level, the interrupt handler must save the user's processor context. The user's context is comprised of 34 user level registers: `r1` to `r31`, `fpcr` and `fpsr`, and the execution address. This context is saved in one of two possible places:

- Temporary storage on the kernel stack. If user level preemption has been deferred by user lock management (i.e., if `lock_count` and `in_upcall` in register `r28` are non-zero), then the register context is temporarily saved onto the kernel stack. These registers are restored when the interrupt handler completes and returns control to the user.

- The user level thread context saved area. If user level preemption is not deferred (i.e., if `lock_count` and `in_upcall` in register `r28` is zero), then the register context is saved into a buffer maintained at the user level. A pointer to this buffer is retained in processor register `r29`. When the user level thread scheduler runs a thread, it sets `r29` to point to the threads context save area. The interrupt handler saves the context directly into the user supplied buffer.

[3]The EPSR register is the 88100 exception time shadow register for the PSR register. The EPSR reflects the value of the PSR before the exception occurred.

3.11.3 Dispatching interrupts

Once the processor context is saved, `INT_handler()` calls the interrupt dispatcher in `interrupt.c` to process the interrupt. There are two interrupt dispatchers:

- `void intr_dispatch_s( int cpu, int intr_vec, int intr_status );`

  This dispatch routine handles all interrupts while the processor was executing the idle loop (i.e., when the processor is in supervisor mode). This dispatcher is fairly simple, because there are no requirements to clean up a currently executing task.

- `TASK *intr_dispatch(int cpu, int intr_vec, int *context, TASK *curr_task);`

  This dispatch routine handles all interrupts occurring while the processor executes at the user level. The routine has additional duties because the currently executing task must be properly managed.

The interrupt dispatch module maintains a table, `intr_table[]`, which stores the handlers for each of the 32 possible interrupts. Both of the interrupt dispatch routines use this table to locate the appropriate interrupt service routine to handle the interrupt. The elements of this table are defined by the `INTR_HANDLER` structure:

```
typedef struct
{
    TASK *user_task;
    void (*routine)(int, int, TASK *);
    int lock;
} INTR_HANDLER;

INTR_HANDLER intr_table[NUM_INTERRUPTS];
```

The `user_task` field designates the task that should be invoked to handle the interrupt. This field is set to `NULL` if the interrupt handler is local to the kernel. If the handler is local to the kernel, `routine` contains the address of the interrupt service routine.

The routines `intr_register()` and `intr_deregister()` manage the `intr_table[]` entries to allow a user level task or local kernel function to be the recipient of any interrupt.

To handle a kernel level interrupt service routine, the dispatcher simply makes a function call to the service routine. But for user level handlers, the procedure is more complicated. An address space switch may be required to activate the proper interrupt handler task.

The dispatcher checks the appropriate interrupt entry, and if the currently executing task is registered to handle that interrupt, then an `INTR_UPCALL` event is given to the task. The particular interrupt vector bit is recorded in the shared region `intr_status[]` field. The user level interrupt dispatcher examines this field to determine the proper interrupt handler for the event. This action is demonstrated by the following code from `intr_dispatch_s()`:

```
task->sr->intr_status[cpu] |= 1 << intr_vec;
task_upcall(cpu, task, 1 << INTR_UPCALL); /* no return */
```

`task_upcall()` is a non-returning call in the task scheduler that activates the specified task and issues an upcall into its address space.

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. But 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 `INT_handler()`, 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 `RELINQUISH_UPCALL`. The user level upcall handler will then take the currently executing thread and place it back onto the thread ready queue. The upcall handler then immediately returns to the kernel via `task_intr_relinquish()`. At this point, the current task is returned to the ready queue, and the interrupt handler task is activated and issued an `INTR_UPCALL` to handle the interrupt.

The Figure 3.17 flowchart summarizes the interrupt dispatching process.

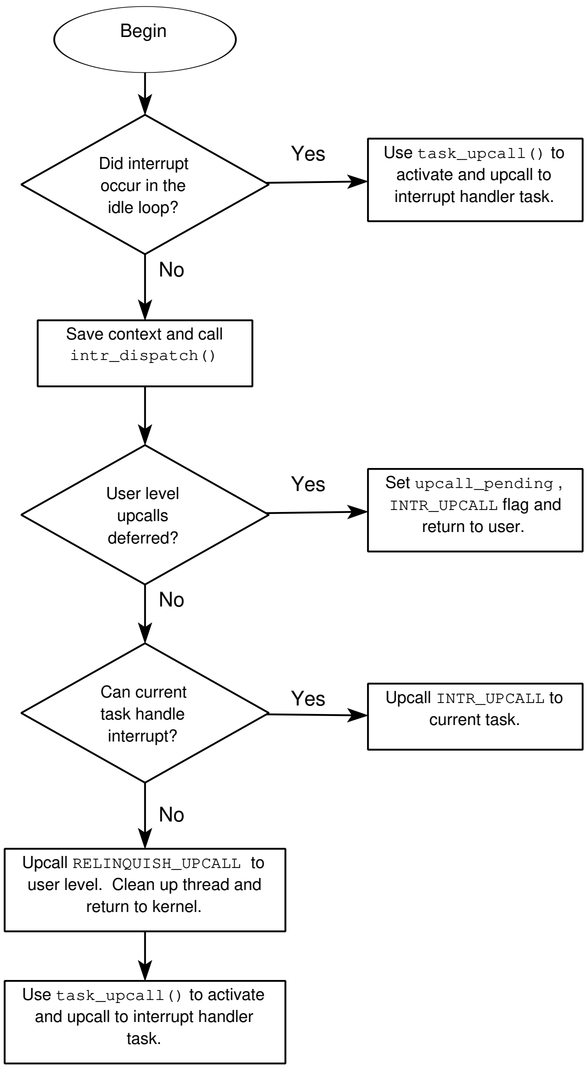

Figure 3.17: Kernel interrupt dispatching process.

3.11.4 Kernel managed interrupts

Several interrupt sources are managed directly by kernel level service routines. These interrupt vectors are the system clock tick timer (DTI), and software interrupt (SWI).

System clock tick

The system clock tick is the heartbeat generator for the kernel. The tick hardware used is the timer component of the onboard MC68681 DUART chip. The Hypermodule connects the timer output to the DTI interrupt vector. The MC68681 is initialized at boot time to a default or user specified interrupt rate.

The clock interrupt service routine, `DTI_service()`, checks the task `timer_q` list, and readies the tasks that have expired times. If an idle processor is available, the processor is delivered a `TASK_RUN_SWI` interrupt to run the task.

After all such tasks have been readied, a `TICK_UPCALL` event is issued to the currently executing user level on the local processor. This upcall event is used by the user level as a thread preemption timer to timeslice threads. The user level kernel counts the number of `TICK_UPCALLS` it receives, and relinquishes its processor when the value reaches a quantum. There is a certain amount of trust placed in the user level to properly relinquish control when its quantum has expired.

Clock ticks are not issued to each processor in the system at once. Only one processor at a time has its clock tick interrupt vector enabled. When the tick occurs on a processor, the `DTI_service()` routine advances the tick interrupt to the next processor. Thus, tick interrupts propagate around all processors evenly in a round-robin fashion. This helps reduce the lock contention that would occur if all processors tried to access the task data structures that the same time.

Remote processor interrupts

The Hypermodule contains a software interrupt register (SWI), which allows any processor in the system to interrupt any other processor. A remote processor is issued an interrupt when its associated bit is written in the SWI register. An external data structure, `SWI_MSG` in `shared_region.h`, managed by the kernel software, allows event messages to be recorded about the interrupt.

```
typedef struct
{
    int msg;            /* SWI event message type */
    int task_id;        /* event task id */
    int data;           /* event type data */
    int more_data;      /* more event type data */
    int lock;           /* spin-lock to protect this entry */
} SWI_MSG;

SWI_MSG swi_msgs[NUM_CPUS];
```

Each processor in the system maintains its own `swi_msg` entry in a global `swi_msgs[]` table. To deliver an interrupt to a remote processor, a message and data is recorded into the processor's `swi_msg` entry. A lock must first be acquired on the `swi_msg` entry before access to it is permitted. The local processor then sets the appropriate bit in the SWI register to interrupt the remote processor.

The remote processor will receive the interrupt, and use the `SWI_service()` routine to handle it. `SWI_service()` examines the `swi_msg` entry and performs the appropriate action based on the message and data.

The `swi_msgs[]` table is allocated on a single physical page which is shared read/write between all address spaces in the system. Also, the SWI register is mapped into all address spaces. This allows any address space to efficiently deliver a remote interrupt. System calls from the user level are not required.

The following four interrupt messages can be issued using the `swi_msg` entry:

- `TASK_SUSPEND_SWI` – issued by `task_suspend()` to suspend the execution of a task on a processor.
- `TASK_DESTROY_SWI` – issued by `task_destroy()` to suspend execution of a task on a processor.
- `TASK_RUN_SWI` – issued by thread and task schedulers to initiate execution of a new thread or task.

- `TASK_SIGNAL_SWI` – issued by `task_signal()` to send a signal event to a task.

The interrupt module provides two routines to manage the delivery of remote interrupts. The caller specifies a `char` array list of processors to interrupt. A non-zero entry in the list indicates that the associated processor should be delivered an interrupt.

- `void intr_remote_cpus( char *cpus, int msg, int task_id );`

  This routine is used to send an interrupt to a list of processors, specified by the list `cpus`. The `swi_msg` entry for each processor in the list is acquired, the message is written, and an interrupt is delivered.

- `int intr_remote_cpu( char *cpus, int msg, int task_id );`

  This routine is used to send an interrupt to one of the specified processors in the `cpus` list. If the `swi_msg` lock cannot be acquired (i.e., if the `swi_msg` is currently being used to deliver an interrupt by another processor), then the processor is skipped and the next one in the list is tried. The routine returns `OK` after the first interrupt is delivered, or `FAILED` if no `swi_msg` entry could be acquired.

These two routines are not exported to the user level. Since the `swi_msgs[]` table and the SWI register are mapped into the user address space, similar routines can be implemented there, avoiding system calls to the kernel.

3.12 User/Kernel Shared memory regions

The Raven kernel relies on shared memory regions to help reduce communication costs between the user and kernel levels. These regions contain high-use information that is frequently accessed when making scheduling decisions at the kernel and user levels. Instead of invoking system calls to pass this information between the user and kernel, read and write operations to the shared regions are used to perform the same effect. Simple read and write operations are much cheaper than system calls.

However, the use shared memory regions at the user level opens up system security holes. Errant program behaviour and rogue processes can adversely modify the information contained in the shared regions, causing any number of execution problems ranging from improper scheduling of tasks to fatal system crashes. This section summarizes the shared regions and describes techniques used to help avoid their abuse.

3.12.1 The shared regions

There are three shared memory regions between the user level and kernel. The table in Figure 3.18 summarizes these regions and their protection status.

Region name	User Protection	Description
`KERNEL_INFO`	read-only	General purpose kernel state.
`SHARED_REGION`	read/write	Scheduling information between user/kernel.
`SWI_MSGS`	read/write	For passing SWI interrupt messages.

Figure 3.18: User/Kernel shared memory regions.

The `KERNEL_INFO` structure is a read-only page that is mapped into all user level address spaces. It is mapped read/write into the kernel address space.

The `SHARED_REGION` structure is a read/write page that is pair-wise mapped between the user/kernel address space. Each task maintains its own `SHARED_REGION` structure for scheduling purposes.

The `SWI_MSGS` structure is a read/write page that is mapped into all address spaces. It is used to pass software interrupt event information between processors. All address spaces require access to this page so that software interrupts can be invoked without kernel intervention.

3.12.2 Abuse Prevention

The current technique used by the Raven kernel to reduce the tampering of shared memory regions is to map the regions at non-obvious locations in the virtual address space. If regions are allocated in sparse areas of the address space, chances are that erroneous or malicious program behaviour will result in memory access exceptions before the shared regions are discovered. However, this technique is certainly not failsafe.

In order to fully protect shared memory regions at the user level from invalid access, the processor must have the ability to mark sections of memory in the user space as privileged. Accessing these sections would require the executing code to have the same level of privilege. The user level kernel could use this feature to enable its code and shared regions with a certain level of privilege, higher than the level given to user application code. However, most modern microprocessors, including the 88100, do not allow this kind of flexibility, so alternatives must be considered.

One alternative to mimic multiple levels of privilege within the same address space is to dynamically change the address mapping tables as the privileged code executes. When a function in the user kernel requires information from a shared region, it could map in the appropriate page, access the data, and unmap the page. Special purpose address mapping system calls could be coded very efficiently in assembler for this purpose. However, additional costs such as TLB misses and flushing could make this approach impractical. If the translation tables were accessible to user programs, then these system calls could be avoided, but the security situation would be even worse due to the exposure of translation tables.

Other techniques to improve security in this area are being investigated.

Chapter 4

User level kernel implementation

Each user level task running in the system requires a user level kernel library. This library contains all of the operating system services that are not implemented in the supervisor kernel, such as: thread management, synchronization primitives, and interprocess communication. This chapter discusses the implementation of the user level kernel, and how it interacts with the supervisor kernel and user programs.

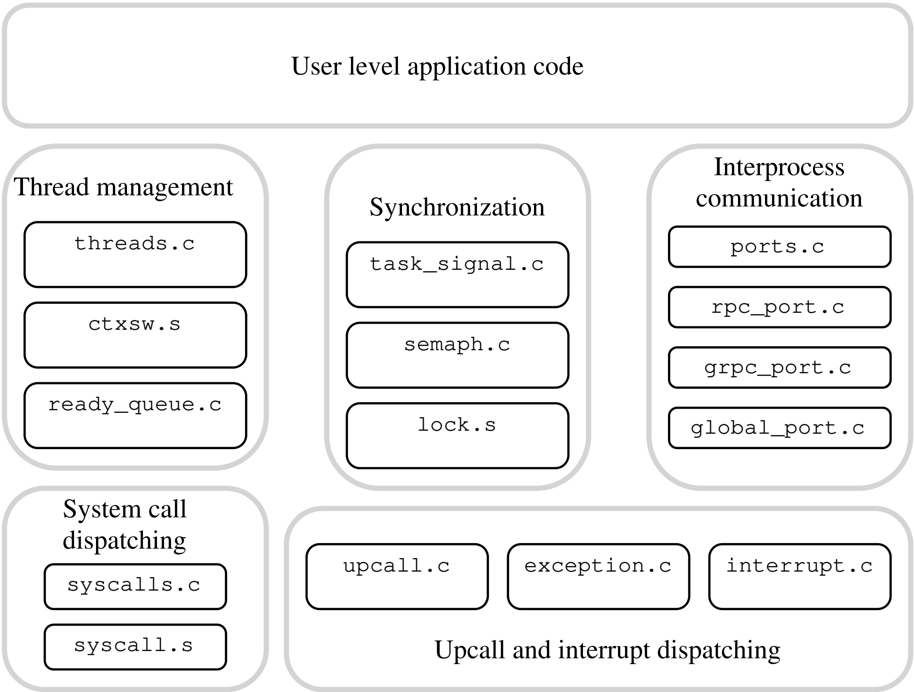

Figure 4.19: User level kernel source code organization.

Figure 4.19 shows the modular breakdown of the user level kernel, and the source code files involved. This section begins by discussing the user level upcalling dispatching mechanism. We then then continue by describing the threading environment and support modules.

4.1 Upcall handling

When a task is loaded into the system, `task_create()` allocates a memory space, as in Figure 3.8, and preallocates an initial stack for the task to execute in. This stack is known as the *upcall stack*, and is used by the processor to handle upcalls events into the task. Since upcall events to the same task can occur in parallel on multiple processors, a separate upcall stack for each processor is required. The upcall stacks are located at well known location at the end of heap space.

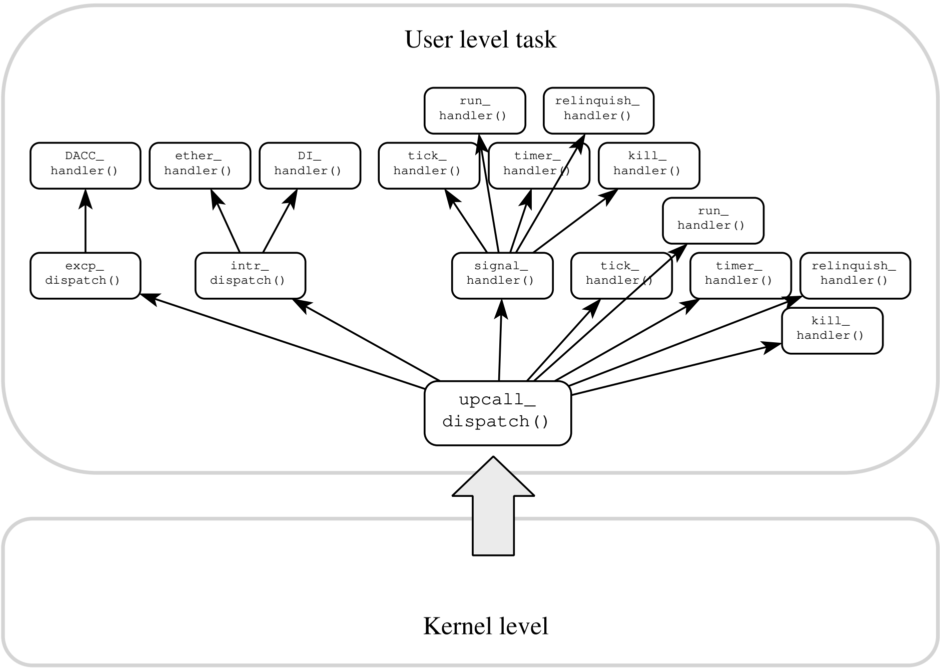

Figure 4.20: User level upcall dispatching routines.

When the kernel upcalls into user space, it places the upcall event flags into the task's shared region `upcall_status[]` entry[1]. All upcalls into the user space are funnelled through a single dispatch routine, `upcall_dispatch()` in `upcall.c`. `upcall_dispatch()` scans the

[1]The `this_cpu()` function returns the local processor number.

`upcall_status[]` entry for event bits and handles each of the flagged events by calling the appropriate upcall handler, clearing the event bits as it goes. Figure 4.20 shows a typical set of upcall handlers, including an exception handler routine and two interrupt device driver routines.

Preemption is deferred at all times on the local processor during upcall handling. This prevents nested upcalls to occur. For example, if an interrupt occurs during upcall dispatching, the interrupt event is recorded, but control returns directly to the user level to proceed with upcall dispatching. The upcall dispatcher will notice the occurrence of any event by examining its `upcall_status[]` entry, and handle it appropriately. Preemption is deferred by setting the `in_upcall` flag in the processor register `r28`. The kernel level upcall dispatcher examines this flag before issuing any upcall, and if set, does not issue an upcall.

The upcall mechanism maintains a table of upcall handlers, `upcall_table[]`. This table is comprised of a static list of upcall handler functions, one for each upcall event type. The kernel task management module defines 8 possible upcall events that must be handled by the dispatcher. The following Figure 4.21 describes the actions performed by each of the upcall event handlers.

A second upcall dispatch routine `upcall_resume()` is provided to invoke dispatching by the user level spin lock library. When a spin-lock is held, upcall preemption is deferred until the lock is released. The lock release code checks to see if an upcall is pending, and if so, branches to `upcall_resume()` to dispatch the pending upcalls.

While `upcall_dispatch()` is invoked by the kernel in the upcall context, `upcall_resume()` is invoked within the calling thread's context. So before proceeding with upcall dispatching, `upcall_resume()` must first save the thread's context and put the thread back on the ready queue.

4.1.1 User level interrupt and exception handlers

All of the entries in the `upcall_table[]` are defined at compile time to point to well known handler routines in the user level kernel. However, two of these upcall handlers, `excp_handler()`

Upcall Event	Handler function	Description
RUN_UPCALL	run_handler()	The task has been given a processor, and can begin scheduling threads.
TICK_UPCALL	tick_handler()	A system clock tick occurred, schedule another thread. A quantum count is incremented, and if expired, the processor is relinquished to the kernel.
RELINQUISH_UPCALL	relinquish_handler()	The task scheduler has requested that this task relinquish the processor.
TIMER_UPCALL	timer_handler()	The task timer service has indicated that a timed event in this task has expired. Check the list of sleeping threads and ready any threads that have expired waketimes.
SIGNAL_UPCALL	signal_handler()	At least one message is waiting in the shared region signal queue. Dequeue each signal message and call their signal handlers.
KILL_UPCALL	kill_handler()	The task scheduler has destroyed this task, so the user level state must be cleaned up, and control returned to the kernel.
EXCP_UPCALL	excp_handler()	A processor exception has occurred. The handler checks the `excp_status[]` entry to determine the exception code and branches to the appropriate exception handler.
INTR_UPCALL	intr_handler()	A device interrupt occurred that is registered for this task. The `intr_status[]` entry is used to determine the interrupt vector and user interrupt handler.

Figure 4.21: Summary of upcall events and the user level upcall handlers.

and `intr_handler()`, furnish a level of indirection to user provided handlers.

The user provided handlers are invoked from within the context of the upcall dispatching mechanism. Preemption is deferred, and execution runs on the upcall stack. Handlers that execute within this context must not invoke thread management routines that may cause a thread context switch to occur. Thus, handler routines must be careful not to block on semaphores or such. Spin locks are allowed because they do not involve context switching.

Interrupt handers

The `interrupt.c` module maintains a table of user provided interrupt handlers, `intr_table[]`. Each entry in the table contains a pointer to a handler routine:

```
typedef struct
{
    void (*routine)(int);
    int  lock;
} INTR_HANDLER;

INTR_HANDLER intr_table[NUM_INTERRUPTS];
```

The following two routines are provided by the `interrupt.c` module to manage the entries in `intr_table[]`:

- `int intr_register( int intr_vec, void *handler );`

  This routine registers a user supplied handler routine to be called whenever the hardware interrupt vector `intr_vec` is signalled. The list of hardware interrupt vectors are listed in the kernel file `registers.h`. The kernel system call `intr_register_user()` is called to register the interrupt with the kernel and set the interrupt enable bit for the vector. Returns `OK` if successful, or `FAILED` if an error occurred.

- `int intr_deregister( int intr_vec );`

  This routine disables the interrupt vector `intr_vec` and removes the user level handler routine from the `intr_table[]`. The kernel system call `intr_deregister_user()` is used to disable the interrupt enable bit for the vector. Returns `OK` if successful, or `FAILED` if an error occurred.

Exception handers

The `exception.c` module is basically a mirror of the `interrupt.c` module. It provides the same functionality, except that its attention is directed towards processor exceptions rather than interrupt exceptions.

One use for the user level handling of exceptions is to trap the data access exception DACC. This exception occurs whenever a memory load or store operation happens outside of the user level address space. This could be used to implement a user level paging mechanism. Or, a handler could be constructed that would detect thread stack overflows and automatically allocate more room. Handlers for other exceptions, such as code violations and division by zero, could be written to destroy the offending thread context, rather than destroying the whole task.

`exception.c` maintains a table of user provided exception handlers, `excp_table[]`. Each entry in the table contains a pointer to a handler routine:

```
typedef struct
{
    void (*routine)();
    int  lock;
} EXCP_HANDLER;

EXCP_HANDLER excp_table[NUM_EXCEPTIONS];
```

Similar to the `interrupt.c` module, two routines are provided to manage the entries in `excp_table[]`:

- `int excp_register( int excp_vec, void *handler );`

  This routine registers a user supplied handler routine to be called whenever the exception vector `excp_vec` is signalled. The list of hardware exception vectors are listed in the kernel file `registers.h`. The kernel system call `excp_register_user()` is called to register the exception with the kernel. Returns `OK` if successful, or `FAILED` if an error occurred.

- `int excp_deregister( int intr_vec );`

  This routine removes the user level handler routine from the `excp_table[]`, and calls the kernel system call `excp_deregister_user()` to deregister the exception from the kernel. Returns `OK` if successful, or `FAILED` if an error occurred.

4.2 User level spin-locks

The user level spin locking code, `lock.s`, provides the same interface as the kernel version, but it is implemented quite differently at the user level. The user level spin-lock mechanism needs to prevent thread preemption while a lock is held. If a thread is preempted while holding a lock, other threads in the system will busy-wait trying to acquire the lock until the thread is rescheduled and the lock is released. To solve this problem, a special locking protocol is implemented between `lock_wait()/lock_free()` and the kernel level upcall dispatcher mechanism.

```
void lock_wait(int *lock);
```

The `lock_wait()` call tries to acquire a lock by spin-waiting on a cached copy of the lock, just like the supervisor kernel version. However, when it does acquire the lock, the `lock_count` variable in register `r28` is incremented to notify the kernel level upcall dispatching mechanism that a lock is held. If an interrupt or other task scheduling event occurs, the kernel upcall dispatcher checks the `locks_held` counter and defers the upcall event if it is non-zero. The upcall dispatcher will not preempt the user level when the `lock_count` is non-zero.

```
void lock_free(int *lock);
```

`lock_free()` writes a 0 value to the lock variable, and decrements the `lock_count` variable. If `lock_count` reaches zero, and an upcall is deferred (bit 16 of `r28` is set), then control is passed immediately to the `upcall_resume()` call. `upcall_resume()` invokes the user level upcall dispatcher, where the deferred upcall is properly handled.

4.3 Thread management

The thread management library manages the creation, destruction, and scheduling of lightweight threads of control. Threads are preemptively timesliced, and and freely migrate from processor to processor, in an effort to balance workload. A central 32 level round robin priority queue manages the scheduling ordering of threads.

4.3.1 Thread descriptor structure

The thread descriptor control block structure, `TCB` defined in `kernel.h`, comprises the data that makes up a thread:

```
typedef struct
{
    int id;                         /* thread identifier */
    int state;                      /* scheduling state */
    int priority;                   /* priority level, 0 to 31 */
    char name[THREAD_NAME_SIZE+1];  /* string name for thread */

    int stack_region;               /* VM region of thread stack */
    int *context;                   /* thread register context area */

    int sem_wait_id;                /* sem this thread is waiting on */
    unsigned int wake_time;         /* time to wake this sleeping thread */

    QUEUE_LINK link;                /* queue link in system queues */
    int lock;                       /* spin-lock for this thread */
} TCB;

TCB thread_table[THREAD_TABLE_SIZE];
TCB *thread_free_q;
int  thread_free_q_lock;
```

All thread descriptors are stored within the static `thread_table[]` array. At initialization time, free thread descriptors are linked into the `thread_free_q`, to make descriptor allocation an easy dequeue operation.

Threads are referred to by their index into this table, the identifier field `id`. The `state` field maintains the threads scheduling state. `priority` contains the scheduling priority for the thread, a value from 0 to 31. The `name` field stores a string name for the thread (useful for debugging purposes).

Each thread is allocated a page-aligned context save area and stack from the kernel memory allocator, `vm_alloc()`. The region descriptor belonging to the stack is saved in the `stack_region` field. The address of the context save area is stored in the `context` field. The thread's stack is positioned immediately below this context buffer. Figure 4.22 shows this relationship.

The `sem_wait_id` field is used by the local semaphore library to store the identifier of the semaphore that the thread is blocked on. If the thread is ever destroyed while blocked on a semaphore, `sem_wait_id` is consulted to remove the thread from the appropriate semaphore

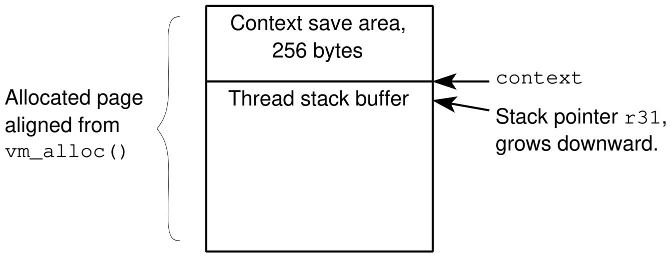

Figure 4.22: Thread context save area and stack buffer.

entry.

The `wake_time` field is used by the `thread_sleep()` facility to record the future wakeup time of the thread.

The `link` field is a queue link that links the thread descriptor into the various user level thread management queues. There are several of these queues: the ready queue, semaphore queues, and IPC queues.

Finally, the `lock` field provides a spin lock to protect against concurrent accesses to the thread descriptor data structure.

4.3.2 Thread context switching

The low level thread context switching routines provide the basic mechanism to load and save the thread execution context. The thread context is a collection of 29 general purpose registers: `r1` – `r25`, the frame pointer `r30`, the stack pointer `r31`, and floating point registers `fpcr` and `fpsr`. In addition to the processor registers, a thread instruction pointer is also maintained.

Two assembler routines are provided by the `ctxsw.s` module to assist in the saving and loading of thread contexts: `load_context()` and `save_context()`.

```
void load_context(int *context, int *thread_lock);
```

This routine loads the processor registers with the context buffer provided by `context`. The thread descriptor must be previously locked to prevent writes to the context area while it is being loaded (if an interrupt occurs in the middle of loading a thread's context, the interrupt handler must not overwrite this context or inconsistency would result). Processor register `r29` is set to point to the context buffer. (At interrupt time, the kernel interrupt handler consults `r29` to find the context save address.) When the context is finished being loaded, the thread descriptor is unlocked (hence the reason for passing the `thread_lock` parameter), and a jump instruction branches to the address contained in the thread instruction pointer. `load_context()` never returns to the caller.

A thread's context is saved in one of two ways: by the kernel interrupt handler when a interrupt occurs, or by the thread scheduler to switch out a running thread. The thread scheduler uses `save_context()` to save the calling thread's context registers.

```
int save_context(int *context, int *thread_lock);
```

`save_context()` saves the calling thread's register context into the supplied `context` buffer. Once saved, the thread descriptor lock is released. `save_context()` returns a non-zero value to the caller. When the thread context is loaded again sometime in the future, the thread instruction pointer returns back to the same location where `save_context()` was called from. Except in this case, a zero value is returned. Therefore, the caller must check the return value to see if execution is continuing, or if the thread has been restored. The following code demonstrates this:

```
/* save the thread's context and drop through */
if ( save_context(running_thread->context, &running_thread->lock) )
    /* When thread context is reloaded, control returns here. */
    /* This return statement jumps back to the thread.        */
    return;

/* normal execution drops through to here */
```

4.3.3 Thread ready queue

The thread scheduler uses a 32 level round robin priority queue to order ready threads. Priority level 0 is low, priority level 31 is high. A low priority thread will never be scheduled to run if there is a higher priority thread waiting. However, this ordering does not span tasks. A low priority thread in a remote task will be allowed to run even though there are higher priority threads in the local task.

The `ready_queue.c` module implements a priority queue structure and interface that is similar to the kernel level task ready queue. The `readyq_table[]` keeps track of 32 queues, one queue for each priority level. The `hint` variable is used to cache the highest priority level with a non-empty queue, to help speed up dequeue operations. A spin-lock `ready_q_lock` protects against concurrent accesses to the ready queue structures.

```
static QUEUE readyq_table[NUM_READYQ_PRIORITIES];
static int hint;
static int ready_q_lock;
```

Three routines are provided which operate on the above structures:

- `TCB *dequeue_ready_thread();`

  Dequeues the next available ready thread from the ready queue, and decrements the shared region `num_ready_threads` counter.

- `void enqueue_ready_thread(TCB *thread);`

  Puts the thread descriptor `thread` into the `THREAD_READY` state, enqueues it onto the ready queue, and increments the shared region `num_ready_threads` counter. If an idle remote processor is available, an `TASK_RUN_SWI` interrupt will be delivered to that processor to run the thread. Otherwise, if a processor has not been previously requested, the kernel system call `task_request_cpu()`, drops down to the task scheduler to request another processor for the task.

- `void remove_ready_thread(TCB *thread);`

  Removes the specified thread descriptor from the ready queue structure, and decrements the shared region `num_ready_threads` counter. This is used by the `thread_destroy()` and `thread_suspend()` routines to remove a thread from anywhere within the ready queue.

4.3.4 Thread scheduling

The thread scheduler is the heart of the user level kernel. When the upcall dispatcher finishes handling all of the upcall events, control is passed to the scheduler. Ready threads are dequeued from the ready queue, and executed via `load_context()`. When a thread blocks, the scheduler saves the thread's context, and schedules a new thread.

There are two main thread scheduling routines: `sched()` and `sched_no_save()`. These routines are used internally by the thread package, and are not intended for general purpose use by user programs. A third interface, `thread_sched()` is available to user programs for voluntarily relinquishing control.

```
void sched(TCB *running_thread);
```

The `sched()` routine is responsible for saving the `running_thread` context and dispatching a new thread. The caller is assumed to already have placed the `running_thread` descriptor on the proper queue; this routine only saves the context, it does not do any queuing. After saving the caller's context, `sched()` dequeues the next ready thread, sets its state to `THREAD_RUNNING`, and performs `load_context()` to execute the thread. If there is no ready thread available, the `task_relinquish()` system call relinquishes control of the processor back to the kernel task scheduler.

```
void sched_no_save();
```

`sched_no_save()` is the same as `sched()`, except that the caller's context is not saved. This is used primarily as the final call by the upcall dispatcher, which doesn't run in a thread context, to begin scheduling threads. `sched_no_save()` dequeues the next ready thread, sets its state to `THREAD_RUNNING`, and performs `load_context()` to execute the thread. If there is no ready thread available, the `task_relinquish()` system call relinquishes control of the processor back to the kernel task scheduler.

```
void thread_sched();
```

`thread_sched()` is a user-callable thread library routine. This call performs a voluntary thread relinquishment. The caller thread is placed on the ready queue, and `sched()` is called to save the thread context and schedule another thread.

4.3.5 Thread creation

All threads are created using the library call `thread_create()`. `thread_create()` allocates a free thread descriptor from the `thread_free_q`, allocates a stack, and sets up the thread's initial execution context. The execution context is built from the thread entry point, exit point, initial stack address, and parameters. The exit point for a thread is called when the thread "runs off the end" of its function, and is set to the `thread_destroy()` call.

```
int thread_create( int *thread_id, void (func)(), char *name,
                   int priority, int stack_size, int ready, int num_args, ...);
```

`func` specifies the entry point for the thread. `name` is a null-delimited string which identifies the thread (useful for debugging purposes). `priority` specifies the ready queue scheduling priority.

`stack_size` specifies the size of the buffer to allocate for the thread's stack. Currently, all stack buffers are allocated using the kernel memory allocator, `vm_alloc()`. Thus, stacks are always allocated on page boundaries. The smallest stack size is therefore 4096 bytes.

The `ready` field specifies the scheduling state that the thread should be placed in after it is created. A non-zero value passed in `ready` signifies that the thread be placed on the ready queue and executed if a processor is available. If zero is passed in `ready`, the thread is placed in the `THREAD_SUSPENDED` state, to be later resumed by `thread_resume()`.

The `num_args` field specifies the number of parameter arguments to pass to the created thread. The thread arguments are specified immediately after the `num_args` field as a variable argument list.

If the thread creation was successful, the thread identifier is returned in `thread_id`, and `OK` is returned. Otherwise, `FAILED` is returned if an error occurred.

4.3.6 Destroying a thread

The `thread_destroy()` routine removes a thread from the local task. The thread's stack is freed, and the thread descriptor is placed back onto the `thread_free_q`. In a multiprocessor environment, however, the specified thread to destroy may be executing on a remote processor. In this case, an interrupt must be delivered to the remote processor to suspend execution of the thread.

If the currently executing thread is destroying itself, then additional work needs to be done to properly free the threads stack buffer. The currently executing thread cannot free its own stack, or else execution will have no stack to continue with. Instead, the thread descriptor is placed in the `THREAD_DYING` state, and is queued onto the `thread_dying_q`. A special low-priority `idle_cleanup()` thread is created at initialization time, which waits for threads to be placed on the `thread_dying_q`, and frees the thread stacks when they become available.

```
int thread_destroy( int id );
```

The thread to destroy is specified by `id`.

4.3.7 Suspending and resuming a thread

A thread can suspend execution of itself using the `thread_suspend()` call. This routine places the thread descriptor into the `THREAD_SUSPENDED` state, and will not be executed again until it is resumed via `thread_resume()`.

```
int thread_suspend();
```

This function suspends the caller thread indefinitely.

```
int thread_resume( int id );
```

This function resumes normal scheduling priority of the specified thread.

4.3.8 Sleeping a thread

The thread sleeping facility is used to suspend the execution of threads for a predetermined length of time. The sleep facility is built on top of the task timer service, which generates upcall events at specified times.

The `timer_handler()` upcall routine and `thread_sleep()` routine share a sorted queue, `thread_sleep_q`. Sleeping threads are placed on this queue, sorted in the order of wakeup time. When `timer_handler()` is called, expired threads on the `thread_sleep_q` are placed on the ready queue and executed.

```
int thread_sleep( unsigned long sleep_time );
```

`thread_sleep()` puts the caller thread to sleep for the specified number of clock ticks. The number of clock ticks per second is a value that is interactively set at kernel boot time.

4.4 Semaphore management

Semaphores provide a way to synchronize actions and communicate events between threads. A common use for semaphores is to provide mutual exclusion around a piece of sensitive code. Other uses include controlling producer/consumer type problems between threads. For example, the interprocess communication system uses semaphores to synchronize message passing events between threads.

The user level semaphore library, `sem.c` and `global_sem.c`, provides two types of semaphores through the same function call interface. Lightweight semaphores, local to an address space can be created for exclusive use between threads in the same address space (used by the local IPC implementation). Global semaphores can also be created, which allow threads in remote address spaces to synchronize between each other (used by the global IPC implementation). The global semaphore implementation requires special hooks into the kernel, and is therefore slightly more costly in terms of performance. A parameter in the `sem_create()` call allows the caller to specify whether a local or global semaphore should be created.

4.4.1 Local semaphores

Local semaphores are used only between threads in the same address space. They are specially optimized for the local case, and no kernel involvement is required. The semaphore library keeps track of each semaphore using a statically allocated table of descriptors:

```
typedef struct semaph_s
{
    int id;          /* semaphore identifier */
    int state;       /* SEM_FREE or SEM_USED */
    int sequence;    /* for deletion protection */
    int count;       /* semaphore count variable */
    int lock;        /* spin-lock protecting this entry */
    QUEUE wait_q;    /* queue of waiting threads */
    struct semaph_s *next_sem;
} SEMAPH;

SEMAPH sem_table[SEM_TABLE_SIZE];
SEMAPH *sem_free_head;              /* semaphore free list */
int     sem_lock;                   /* spin-lock protecting free list */
```

The total number of semaphores in a task is bounded at compile time by the `SEM_TABLE_SIZE` constant in the `sem.h` file. Each semaphore descriptor maintains a count value and a queue for storing blocked threads. The `sequence` field allows the semaphore routines to check whether a deletion has occurred. Each semaphore descriptor is linked into a free list, `sem_free_head`, making semaphore allocation a simple operation.

4.4.2 Global semaphores

Global semaphores can be used to synchronize events between threads in separate address spaces, as well as their local address space. The global semaphore routines are similar in semantics to the local case. However, they differ completely in implementation, some kernel support is necessary.

Each address space shares a global semaphore table, `global_sem_table` defined as:

```
typedef struct
{
    int lock;
    int free_sem_q;
    GLOBAL_SEM sems[NUM_GLOBAL_SEMS];
} GLOBAL_SEM_TABLE;

GLOBAL_SEM_TABLE *global_sem_table;
```

This structure is allocated by the first task in the system, and is shared between all further created tasks. The main element in this structure is the table of semaphore descriptors, `sems`. The `free_sem_q` field implements a list head pointer to start a linked list of free semaphore descriptors.

The first task that is loaded into the system uses `vm_alloc()` to allocate a region for the table. The region identifier is then stored in the global nameserver. Future tasks that are loaded into the system query the nameserver for the region identifier, and map the table in using `vm_share()`.

The global semaphore descriptors stored in `sems` comprise the semaphore's main data structure unit:

```
typedef struct
{
    int id;                 /* semaphore identifier */
    int owner_task;         /* owner task_id of this semaphore */
    int sequence;           /* destroy sequence counter */
    int count;              /* semaphore count */
    int lock;               /* spin-lock protecting this entry */
    int next_sem;           /* semaphore free list next pointer */
} GLOBAL_SEM;
```

Other data structures involved in global semaphore operation reside in the local shared region structure, and in the kernel. These data structures are used to support the queuing of blocked threads in disjoint address spaces. Threads in different tasks can block on the same global semaphore. The `sem_signal()` signal operation will unblock one of the threads, but which one? A fair protocol would unblock the thread that has been waiting the longest.

The local semaphore implementation supports this fairness by using a FIFO queue to sort the blocked thread descriptors. A queue is also used in the global semaphore case to provide fairness. However, this queue stores task descriptors, not threads.

Each global semaphore in the system has a task queue associated with it. This queue links together all the tasks in the system that contain threads which are blocked on the semaphore. Since multiple threads in the same task can block on several different semaphores, each task in the system could be linked onto several different global semaphore queues. Thus, each task needs to maintain a table of queue links, one for each possible global semaphore.

The table of queue links for each task is located in the task's kernel/user shared region, as `sem_links[]`. When a thread in a task blocks on a global semaphore, the task is linked onto the semaphore queue using its local link from `sem_links[]`. When a thread in another task invokes the `sem_signal()` operation, the next task in the queue is dequeued and delivered a `GLOBAL_SEM_SIGNAL_SIGNAL` message using the `task_signal()` mechanism. The signal handler for this message then wakes up and runs the appropriate thread.

The semaphore task queues are maintained within the kernel. Whenever a user thread blocks on a semaphore, a system call is required to place the local task on the semaphore queue: `kernel_sem_enqueue()`. However, since the task is now queued on the semaphore, further threads that block on the same semaphore do not invoke the kernel operation. Likewise, the `kernel_sem_dequeue()` system call removes the next task in the queue and delivers it a signal.

4.4.3 Waiting on a semaphore

The `sem_wait()` call decrements the semaphore count variable. If the count remains zero and above, then the call returns immediately. Otherwise, the calling thread is enqueued onto the semaphore's queue, and the thread scheduler is called to run the next thread. The thread will remain on the semaphore's queue until it is unblocked by `sem_signal()` or `sem_destroy()`, or until the thread is destroyed.

```
int sem_wait( int sem_id, int no_block );
```

The semaphore is specified by `sem_id`. If `no_block` is set to a non-zero value, then `sem_wait()` will always return to the caller, regardless of the value of the count variable. This function returns `OK` if successful, `WOULD_BLOCK` when `no_block` is set and the call would normally block, `DESTROYED` if the semaphore was deleted, or `FAILED` if the specified `sem_id` is invalid.

4.4.4 Signalling a semaphore

The `sem_signal()` call increments the semaphore count variable, and readies the next waiting thread from the semaphore's queue using `enqueue_ready_thread()`. The readied thread will eventually resume execution when a processor is available.

```
int sem_signal(int sem_id);
```

The `sem_id` parameter specifies the semaphore to signal.

4.4.5 Allocating a semaphore

The `sem_create()` call allocates a free semaphore descriptor and initializes the descriptor entries. The next free semaphore descriptor is dequeued from the `sem_free_head` queue. The initial value of the semaphore count can be specified by the caller. Either global or local semaphores can be allocated by specifying the `global` parameter.

```
int sem_create( int *sem_id, int count, int global );
```

The initial semaphore count value is specified by the parameter `count`. Passing a non-zero value as the `global` parameter will create a global semaphore, otherwise a local semaphore will be created.

4.4.6 Destroying a semaphore

When a program is finished with a semaphore, it should be returned to the system using the `sem_destroy()` call. Destroying a semaphore requires more work than enqueuing the descriptor on the free list, however. Any blocked threads waiting on the semaphore must be released, or they would remain blocked forever. The blocked threads are dequeued one at a time, and readied with the `enqueue_ready_thread()` call. Before releasing the blocked threads, the semaphore `sequence` field is incremented. When a thread resumes execution in `sem_wait()`, it checks the sequence value against the previous value. If the values differ, then `sem_wait()` can return a `DESTROYED` value.

When destroying a global semaphore, however, any remote tasks that are queued behind the semaphore must be dequeued and delivered a `GLOBAL_SEM_DESTROY_SIGNAL` message. Since the global semaphore queues are maintained within the kernel, the `kernel_sem_destroy()` system call performs this function.

4.4.7 Kernel intervention

The global semaphore library requires some special purpose kernel support for its operation. One of the main goals of the system is to reduce the overall number of these kernel interactions. This subsection summarizes the situations where kernel system calls are necessary during the global semaphore operations.

The following steps are executed by the semaphore wait operation. A kernel call is only necessary if there are no blocked threads on the semaphore.

1. Return if semaphore count is greater than zero.
2. Block calling thread.
3. Call `kernel_sem_enqueue()` if this is the first thread in the local task blocked on the semaphore.

The semaphore signal routine makes a system call only if there is a waiting thread:

1. If the semaphore count indicates a blocked thread, call `kernel_sem_dequeue()`.

2. Return to caller.

4.4.8 Miscellaneous semaphore operations

The following operations can be useful in certain situations.

- `int sem_count( int sem_id );`

  This function returns the count value of the specified semaphore.

- `int sem_reset(int sem_id, int count);`

  This function allows the caller to change a semaphore count value.

4.4.9 Semaphore library initialization

The `sem_init()` semaphore library initialization routine first initializes the local semaphore data structures. The `sem_table[]` is initialized and a `free_list_q` is created.

Then, and the global semaphore initialization routine is invoked. This routine allocates or maps in the global semaphore table, and initializes the local data structures. The global nameserver is queried to see if the `GLOBAL_SEM_REGION_NAME` string exists in the database. If so, the region identifier for the semaphore table is returned, and `vm_share()` is used to map in the memory. If the string is not registered, then a new table is allocated with `vm_alloc()`, and its region identifier is stored in the nameserver database.

4.5 Interprocess communication

The user level kernel supports both port-based synchronous send/receive/reply and asynchronous send/receive communication models. These models are sufficiently different in implementation that each require its own library interface: the synchronous port library, `rpc_port.c` and `grpc_port.c`, and asynchronous port libraries, `ports.c` and `global_ports.c`.

The port based approach to interprocess communication uses a port number as the mailbox address for messages. In a client/server model, the server waits for messages to be received on a port, the client sends messages to a port. When a server cannot receive messages as fast as they are sent, the messages are buffered by the port library. A port must be properly created before any IPC can take place through the interface.

For performance reasons, the port user level libraries distinguish between local ports and global ports. Local ports are used only for communication within a single address space. The implementation of these ports are based on local semaphores and the local thread scheduler. Local ports are more efficient than global ports because all of their interaction is limited to the local address space.

Global ports are used for communication across address spaces, as well as within. These ports make extensive use of shared memory between the client/server address spaces to help reduce communication costs. In addition to reducing data copying, the shared memory also helps reduce the number of kernel interventions required to invoke the port communication protocols.

Local port messages are always passed by reference to reduce data copying. However, since pointers don't make sense in remote address spaces, global port messages cannot use pointers. Instead, to emulate pass-by-reference, a virtual memory region descriptor of the data buffer can be passed, and the `vm_share()` and `vm_move()` system calls can be used to map the region into the local address space. This technique can eliminate data copying when moving data between address spaces. For small pieces of data, however, the overhead of memory mapping outweighs the data copy, and therefore this technique should be reserved for larger chunks of data.

4.5.1 The synchronous port library

The user level library supports the synchronous style send/receive/reply protocol. Using a transaction identifier returned by the receive operation, server threads can defer the reply stage until a later time. The transaction identifier is used by the reply call to identify the proper client message to reply.

The following two subsections describe the implementation of the local and global port libraries. Then, the user interface is described.

Local synchronous port implementation

The local synchronous port library maintains a table of port descriptors `rpc_port_msgs[]`. This descriptor is used to keep track of threads blocked on sends, receives, and replies.

```
typedef struct port_s
{
    int id;                    /* port identifier */
    int state;                 /* USED or FREE status */
    int sequence;              /* incremented at each port_destroy */

    int send_sem;              /* semaphore to block receiver */
    QUEUE send_msg_q;          /* queue of sender messages */
    QUEUE reply_wait_q;        /* queue of threads waiting for a reply */

    struct port_s *next_port;  /* next port descriptor link */
    int lock;                  /* spin lock protecting this port */
} RPC_PORT;
```

The semaphore `send_sem` is used to block server threads on the receive operation. This semaphore counts the number sender threads currently blocked on a send. When a client thread sends a message, the message is queued into the `send_msg_q`, and the `send_sem` is signalled to wake the server thread. The client thread is placed on the `reply_wait_q` and blocks waiting for a reply. The server thread dequeues the client from this queue and unblocks it when it performs the reply operation.

Each thread in a task is allocated its own synchronous message descriptor. All local messages are passed by reference. This message descriptor maintains pointers to the passed data buffers, and a queue link for attaching to a port:

```
typedef struct
{
    int id;                     /* port id this message is queued on */
    char *send_data;            /* pointer to send data */
    int   send_len;             /* length of send_data */
    char *reply_data;           /* pointer to reply data */
    int  *reply_len;            /* pointer to reply data length */
    QUEUE_LINK link;            /* link for port msg queue */
} RPC_MSG;
```

Global synchronous port implementation

Each task in the system shares a region of shared memory that contains a table of global port descriptors. When a global port is created, a free descriptor is allocated from the free list. A region buffer for storing messages is allocated by the owner task, and is shared to all clients that want to interact on the port.

This port descriptor maintains head/tail pointers to the shared port message buffers. Each port descriptor contains three queues: the `msg_free_q` links the free message descriptors together; the `msg_send_q_head` links the send messages together; and `msg_reply_q_head` which links the reply messages together.

A global semaphore in each port descriptor is used to control access to the port's `msg_free_q`. When a client thread sends a message, a message buffer is dequeued from the shared buffer. If there are no free buffers, then the thread blocks on the `msg_free_sem`. When a server releases a message buffer back to the free list, it performs a `sem_signal()` operation to unblock a waiting thread.

The task signalling facility is used to communicate low level send/receive/reply events to the client and server address spaces. These events are not sent on every interaction, but only in the worst-case moments discussed below. The task signalling facility, `task_signal()`, is used to communicate these events to remote address spaces. The following three signal messages are defined for this use:

- the `GRPC_PORT_SEND_SIGNAL` message is sent by a client thread to a server task with a blocked server thread to wake the server thread and tell it that a message awaits. If a send message is already present in the queue, then the library assumes that the server has already been delivered a signal message, so sending another is not necessary.

- the `GRPC_PORT_REPLY_SIGNAL` message is sent by a server thread to a client task to indicate to the client task that a server is replying to a client's message. When the client task receives this signal, it can wake the client thread, and deliver the reply message to it.

- the `GRPC_PORT_DESTROY_SIGNAL` message is sent to all client tasks that are blocked on a port when the port is destroyed. The client threads can then be unblocked and told that the port was destroyed.

The `task_signal()` mechanism can deliver these messages without kernel intervention on the local processor. If a remote processor is executing the destination task, then that processor is interrupted by the user level `intr_remote_cpu()` service. Otherwise, if a remote processor is idle, then that processor is interrupted to handle the signal message. If all processors are busy, then a system call is required to deliver the signal.

Sending a synchronous message

```
int rpc_port_send( int port_id, char *send_data, int send_len,
                   int *reply_data, int *reply_len);
```

This function queues a message of length `send_len` bytes on `port_id` and blocks waiting for a reply. For a global port, the `send_data` buffer is copied into the ports shared buffer region. For a local port, a pointer to the data is passed. `reply_data` is assumed to point to pre-allocated buffer space for the reply message, whose length is specified in `reply_len`. Upon successful return, the global port reply data is copied in `reply_data` buffer and the length in `reply_len`. The call returns `OK` if successful, `DESTROYED` if the port was destroyed while waiting for a reply, and `FAILED` if the port does not exist.

The following client thread demonstrates this call:

```
void client(port)
{
    char reply_msg[20];
    char *send_msg = "send message";
    int reply_len;

    rpc_port_send(port, send_msg, strlen(send_msg)+1, &reply_msg, &reply_len);

    printf("reply message is:%s length:%d\n", reply_msg, reply_len);
}
```

Before the client thread can queue a message into the shared port buffer, the buffer must be mapped into the client's address space. `rpc_port_send()` will automatically do this mapping on the first message send. However, when a client is finished communicating across a global port, the port buffer must be unmapped explicitly. The `rpc_port_dereference()` call, described below, does this.

Receiving a synchronous message

```
int rpc_port_recv( int port_id, char **recv_data, int *recv_len,
                   char **reply_data);
```

This function blocks the calling thread and waits for a message to arrive on the specified port. When a message arrives, `recv_data` and `recv_len` is returned to contain a pointer and length of the received data, and the `reply_data` field points to the reply buffer. This buffer is used by the server to copy the reply message into. When successful, the function returns a `msg_id`, which identifies the synchronous transaction. This `msg_id` is supplied to the `rpc_port_reply()` function to issue a reply on the transaction. The following code demonstrates this interaction:

```
void server(int port)
{
    char *recv_msg;
    int recv_len;
    char *reply_data;
    char *reply_msg = "reply message";
```

```
    int msg_id;

    msg_id = rpc_port_recv(port, &recv_msg, &recv_len, &reply_data);

    printf("received message:%s length:%d\n", recv_msg, recv_len);

    /* copy reply message into supplied reply buffer */
    strcpy(reply_data, reply_msg);

    rpc_port_reply(port, msg_id, strlen(reply_msg));
}
```

Replying a synchronous message

```
int rpc_port_reply( int port_id, int msg_id, int reply_len);
```

This routine is called by the server thread to reply to an `rpc_port_send()` call. The `msg_id` specifies the transaction number obtained from `rpc_port_recv()`. `reply_len` specifies the length of data copied into the `reply_data` buffer from `rpc_port_recv`.

Creating a synchronous port

```
int rpc_port_create( int *port_id );
```

This is the creation routine for local synchronous ports (global ports are sufficiently different to require a separate routine). The created port is returned in `port_id`.

```
int grpc_port_create( int *port_id, int max_data_len, int num_msg_bufs);
```

This is the creation routine for global synchronous ports. The maximum port message size, `max_data_len`, and queue size, `num_msg_bufs`, are specified so that a queue buffer can be allocated. The created port is returned in `port_id`. Returns `OK` if successful, `FAILED` if there are no free ports.

Destroying a synchronous port

```
int rpc_port_destroy( int port_id );
```

This function destroys the specified synchronous port. Only the port creator's address space can destroy the port. All clients blocked on an `rpc_port_send()` are released.

```
int rpc_port_dereference( int port_id );
```

When a client address space is finished sending messages to a global port, it can "dereference" the port. This causes the shared memory region between the client and server to be unmapped from the client address space.

Kernel intervention

The interprocess communication library is designed to reduce the number of user/kernel interactions required to interact between client and server address spaces. Pure kernel implementations require at least three kernel calls per send/receive/reply interaction: one call at each of the send, receive, and reply stages.

The local port communication library does not require any kernel support at all. The global port library does however require kernel system calls in certain cases. Global interprocess communication is based on the task-to-task asynchronous signalling mechanism. The amount of kernel intervention required is directly based on the amount of kernel intervention in the signalling mechanism. An invocation of the signal mechanism breaks down into the following steps:

1. If the destination task is running on a remote processor, deliver it an interrupt and return.
2. If there is an idle processor, deliver it an interrupt and return.
3. Otherwise, make a system call to perform the signal.

The signalling mechanism uses software interrupts to deliver messages to remote processors, avoiding system calls on the local processor. A system call is required only if the third step is reached.

The synchronous IPC mechanism tries to reduce the number of kernel calls by invoking task signals only when necessary. The following lists summarize the stages of the synchronous operations and where system calls and task signals potentially occur.

The `rpc_port_send()` operation requires at most one system call per invocation, or possibly a task signal invocation:

1. `sem_wait()` for a free send buffer, causing a system call only for the first blocked thread (described in 4.4.7).
2. Copy message into buffer.
3. If the send queue is empty, deliver a signal to the destination task.
4. Block sending thread.
5. Wake up and copy reply message to local buffer.
6. `sem_signal()` free buffer queue.

The steps followed by the `rpc_port_recv()` operation does not require any kernel support at all:

1. If a message awaits, return a pointer to the message buffer.
2. If the port is empty, block the thread.
3. Wake up and return a pointer to the message buffer.

`rpc_port_reply()` requires a task signal invocation only when there are no reply messages queued for the client:

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

4.5.2 The asynchronous port library

Asynchronous send/receive communication can be used between threads when no reply message is required. This method can more efficient than synchronous interactions because there is potentially less context switching involved between the client and server thread. Client send messages are copied into a buffer, to be picked up by the server. Client sending threads do not block unless the port queue becomes full.

Threads communicate their message data through a FIFO message buffer. In the local port case, this buffer is dynamically allocate into the local memory space. In the global port case, a shared region is allocated between client and servers, similar to the global synchronous ports.

The local port implementation uses local semaphores to synchronize client/server access to the port message buffer queues. The sender thread waits for free message buffer to become available. The free message buffer is dequeued, and the send message is copied into it. The message is queued onto the port send queue, and the send semaphore is signalled to wake the server thread.

The global port implementation relies on a shared region of port descriptors to manage the port queue. One global semaphore is used to synchronize access to the port free message queue. Client threads wait on the free message queue semaphore for buffers to become available. The server threads block waiting for sent messages.

When a client thread sends a message, a `GLOBAL_PORT_SEND_SIGNAL` message is delivered to the server task to notify server threads that a message is waiting. This signal is sent only when server threads are blocked, and when no messages are queued in the send buffer. That is, the signal is sent only when the send queue is empty.

Sending an asynchronous message

```
int port_send( int port_id, char *msg, int msg_size, int no_block );
```

Sends a message pointed to by `msg` to the specified port. The message buffer of length `msg_size` is copied into the port's message queue. If the message queue is full, this routine will return immediately without queuing the message if `no_block` is set to `NO_BLOCK`. If `no_block` is `BLOCK`, the calling thread will be blocked until room is made available in the message queue. Returns `OK` if the message was queued successfully, `WOULD_BLOCK` if the queue was full, `DESTROYED` if the port was destroyed while blocked, or `FAILED` if the port does not exist.

The following client thread demonstrates this call:

```
void client(port)
{
    char *send_msg = "send message";

    port_send(port, send_msg, strlen(send_msg)+1, BLOCK);
}
```

Before a client thread in a remote address space can send messages on a global port, the port message queue region must be mapped into the client's address space. The `port_send()` routine will automatically map in the port message queue on the first invocation. When communication across the port is later discontinued by the client, the port queue must be explicitly unmapped. The `port_dereference()` call, described below, performs this operation.

Receiving an asynchronous message

```
int port_recv( int port_id, char *msg, int *msg_size, int no_block );
```

Receives a message from the specified port. If the port queue is empty, this will return immediately if `no_block` is set to `NO_BLOCK`. Otherwise, the caller thread will be blocked until a message arrives. `msg` must be a preallocated buffer to hold at least `msg_size` bytes. The message is automatically copied into this buffer when it is dequeued from the port. Returns `OK` if a message was successfully received, `DESTROYED` if the port was destroyed while blocked, `WOULD_BLOCK` if the caller thread would be blocked, or `FAILED` if the port does not exist.

The following server thread demonstrates this call:

```
void server(port)
{
    char recv_msg[20];
    int recv_size;

    port_recv(port, recv_msg, recv_size, BLOCK);

    printf("message received:%s length:%d\n", recv_msg, recv_size);
}
```

Creating an asynchronous port

```
int port_create( int *port_id, int max_msg_size, int queue_size, int global );
```

This is the asynchronous port creation routine. The `max_msg_size` parameter specifies the maximum length in bytes of the messages passed on this port. `queue_size` specifies the maximum number of messages that can be queued up on the port. If `global` is `PORT_GLOBAL`, then a global port will be created. Otherwise, if `global` is `PORT_LOCAL`, then a local port will be created. Returns `OK` if successful, `FAILED` if there was a parameter error, or if there are no more port descriptors.

Destroying an asynchronous port

```
int port_destroy( int port_id );
```

Destroys the specified local or global port. Remote and local threads blocked on an operation to the port will be woken with a `DESTROYED` failure result. Returns `OK` if successful, `FAILED` if the port does not exist, or if the port did not originate in the caller's task.

```
int port_dereference(int port_id);
```

When client threads in an address space are finished accessing a port, this routine will clean up the shared memory region associated with the port. Returns `OK` if successful, `FAILED` if the port does not exist.

Kernel intervention

Similar to the synchronous port library, the asynchronous library only requires kernel support for supporting global communication. The task signalling mechanism is used in the same manner to help reduce kernel intervention.

The `port_send()` call may require a system call if the `sem_wait()` call blocks, or a signal may be delivered:

1. `sem_wait()` for a free message buffer.
2. Copy message into buffer.
3. Deliver a signal only if the send queue is empty and a server thread is blocked.

The `port_receive()` call may require a system call by `sem_signal()` to wake any blocked sender threads:

1. Block if there are no messages.
2. Wake up, dequeue message and copy it into the caller's buffer.
3. Issue `sem_signal()` on the port free buffer queue.

4.6 The global nameserver

The user level library maintains a simple nameserver data structure that is shared read/write by all address spaces in the system. The nameserver database stores (string,integer) pairs, using the string as a search key. User programs can register strings, with them associated an integer value. The integer value can be used to pass handles for various objects, or for other miscellaneous purposes. For example, global ports can be registered, so that remote address spaces can query for server port identifiers.

The following structure is used for each entry in the `nameserver_table[]`:

```
typedef struct
{
    char string[NAMESERVER_STRING_SIZE];
    int  data;
} NAMESERVER_ENTRY;
```

The complete nameserver interface is provided through the following three function calls: `nameserver_register()`, `nameserver_deregister()`, and `nameserver_find()`.

4.6.1 Registering with the nameserver

Any null-character delimited string, of up to `NAMESERVER_STRING_SIZE` characters, can be stored in the nameserver database. The register routine simply performs a linear search through the `nameserver_table[]`, and finds the first available slot.

```
int nameserver_register( char *str, int data );
```

The null-terminated string `str` is stored in the `nameserver_table[]`, with its associated data value, `data`. If the table is full, `FAILED` is returned. The string and data value are copied into the table entry. The string is truncated if not enough space is available. Returns `OK` if successful.

4.6.2 Deregistering a string

The deregister routine removes a string and its data value from the nameserver table. A linear search find the table entry, which is then invalidated.

```
int nameserver_deregister( char *str );
```

The null-terminated string `str` specifies the string to invalidate. Returns `OK` if successful, or `FAILED` otherwise.

4.6.3 Nameserver initialization

The task initialization upcall must initialize the nameserver data structure. The initialization code checks to see if a nameserver table already exists in the system. The nameserver region identifier is passed to a task through the `TASK_ARGS` structure when calling `task_create()`.

The first task in the system allocates the table using `vm_alloc()`. All child tasks are passed the nameserver table region identifier through `TASK_ARGS`. Child tasks, when initializing, uses the region identifier to map in the table with `vm_share()`.

4.7 User/User Shared memory regions

Just as the supervisor kernel implements shared regions between the user and kernel spaces, the user kernel library also makes use of shared regions between user/user address spaces. These regions are used by the interprocess communication library and global nameserver to facilitate data sharing between tasks. The supervisor kernel has no knowledge of these regions.

These regions are susceptible to the same problems that the user/kernel shared regions face, although the affects of such problems are slightly less critical. Damaging a user/user shared region may prevent interprocess communication across a port, but it won't fatally crash the system as damaging the user/kernel shared region would.

4.7.1 Globally shared regions

The user level kernel library shares three regions between each task in the system. The first task that is loaded into the system and executed is responsible for allocating each region. Other tasks that are subsequently loaded share these regions. Each region is shared with read/write protection:

- The global nameserver region, which contains a table of key/value pairs for passing handles and other information.

- The asynchronous port descriptor region, which contains a table of asynchronous port descriptors.
- The synchronous port descriptor region, which contains a table of descriptors for the synchronous port library.

4.7.2 Client/Server shared regions

The synchronous and asynchronous port creation routines allocate a buffer in the server task to queue messages between the client and server. Clients that wish to communicate across the port first map in the buffer region to their address space. So these regions are only made available to those address spaces that ask for it.

Other shared memory regions can be easily created by user programs for application specific purposes.

Chapter 5

Performance Evaluation

This chapter presents some performance benchmarks for various aspects of the kernel operation. We begin by describing the tools used to obtain performance data. Then, several benchmarks are presented and analyzed.

5.1 Benchmark tools

There are two tools available in the system which can provide performance measurements. The Hypermodule's on-board Z8536 Counter/Timer can be configured as a 32-bit timer, with microsecond resolution. The following routines are available within the kernel and at the user level to control the timer:

```
unsigned long timer_start();
unsigned long timer_stop(unsigned long start);
```

The timer hardware is initialized at boot time to continually cycle through a 32-bit counter register. The counting frequency is 2MHz, giving the counter a cycle time of about 35.7 minutes with 1/2 microsecond accuracy. The `timer_start()` function takes a snapshot of the counter register, returning the counter value. This value is then passed to `timer_stop()`, where another snapshot is taken, and the time between snapshots is returned. This value should then be divided by 2 to convert to microseconds.

The second tool is only available while running under the simulator. The simulator provides a special device used to support an instruction tracing feature. This feature allows a processor's execution stream to be saved for later analysis, such as counting the number of instructions executed. A simulator device allows the tracing to be turned on and off, at program control, and categorized into slots. The following macros can be used by any code to control the instruction tracing device:

```
/* 'x' is the category slot */
#define SIMTRACE_ON(x) (*(int *) SIMTRACE_CTRL = x)
#define SIMTRACE_OFF(x) (*(int *) (SIMTRACE_CTRL+4) = x)
```

5.2 Function calls vs. System calls

To help put the performance results in perspective, here we consider the cost of a null system call compared to the cost of a null procedure call. A null procedure with four parameters requires at least 6 instructions on the 88100: four instructions to build the argument list, one instruction to call the routine, and one instruction to return.

```
/* func() is the null workhorse */
void func(int a, int b, int c, int d)
{
    return;
}
```

The kernel system call interface funnels all system calls through a single trap vector. An integer passed across the trap is used to determine the correct system call. The initial system call handler has the job of preserving user level state and setting up the kernel execution context. This involves some register saving and processor configuration. In all, 56 instructions are required to invoke a null system call with 4 parameters.

The basic setup cost and administration for system calls is an order of magnitude greater than a local function call. Table 5.2 summarizes this comparison.

5.3 Thread management performance

This section details the costs of thread creation and scheduling. Thread management operations are primitives that closely affect the performance of tightly coupled parallel applications. In

	Instructions
local function call	6
kernel system call	56

Table 5.2: 4 parameter user level function call vs. kernel system call.

	1 CPU	2 CPU	3 CPU	4 CPU
`thread_create()`	38.1 usec	35.7 usec	34.3 usec	34.1 usec
`dispatch and destroy`	22.3 usec	16.2 usec	13.5 usec	11.4 usec
`thread_sched()`	12.8 usec	8.2 usec	6.5 usec	5.1 usec

Table 5.3: Thread management performance.

these applications, threads are used extensively to perform jobs in parallel, and schedule events from one to another. Often, the raw performance of the underlying thread scheduler can become a bottleneck.

5.3.1 Thread creation performance

This benchmark measures the overall performance of the `thread_create()` routine. A simple program was written which creates 100 threads, but does not execute them. Execution times were collected for four different processor configurations: one, two, three, and four processor versions. In each case, the same number of worker threads is used to create the null thread as there are processors. For example, the four processor measurement uses four server threads to create 25 null threads each, totalling to 100 threads.

Table 5.3 summarizes the results. The basic uniprocessor thread create call completes every 38.1 microseconds. This creation time could be significantly improved by modifying the thread stack allocator. Currently, `thread_create()` performs the system call `vm_alloc()` to allocate its thread stack. An algorithm that allocated a large number of thread stacks at once would reduce the expensive `vm_alloc()` calls.

When more processors are added to the system, performance improves because there of the parallel creation, but only slightly, until levelling off after about 4 processors. This is probably caused by a spin-lock contention bottleneck, seen in three places during thread creation: locking the free thread descriptor queue to allocate a descriptor; locking the task descriptor to perform the `vm_alloc()` system call; and locking the task's address map descriptor to perform the `vm_alloc()` system call. In addition to reducing the amount of code executed, removing the `vm_alloc()` call for stack allocation may also reduce lock contention.

5.3.2 Thread resumption and destruction performance

Using the 100 threads created for the previous benchmark test, the threads are resumed using `thread_resume()`, executed, and finally destroy themselves using `thread_destroy()`. The last thread to destroy itself prints out the execution time for the complete run.

Table 5.3 summarizes the execution times for one, two, three, and four processors. A thread is executed and removed in 19.3 microseconds. This execution time includes putting the thread on the ready queue, invoking the scheduler to dispatch it, and invoking the `thread_destroy()` routine to kill the thread. Adding more processors has a more significant effect on the execution time, compared to the thread creation benchmark. While there is still lock contention for the ready queue list, this contention is not nearly as long as found in the `vm_alloc()` system call. Greater parallelism is achieved in this case.

5.3.3 Thread context switching

This benchmark measures the basic context switching performance of the `thread_sched()` library call. 10 threads are created, and each of them performs the following tight loop:

```
void worker( iterations )
{
    while ( iterations-- )
        thread_sched();
}
```

100,000 iterations were performed by each thread, and the overall execution time was recorded. This number is divided by 100,000 to determine the approximate performance of the `thread_sched()` call. The calling thread's context is saved, and its descriptor is placed on the ready queue. The next available thread is dequeued and its context is restored.

Table 5.3 shows the average performance of `thread_sched()`. A basic context switch from one thread to another on a uniprocessor occurs in 12.8 microseconds. The ready queue offers some lock contention for when additional processors are added. The remainder of the processor time is spent saving and loading the 29 thread context registers[1].

5.4 Interrupt handling performance

The performance of interrupt handling is a critical concern for high speed device drivers and kernel 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 service routine. The kernel interrupt handler was modified to take a timestamp at the earliest convenience. This timestamp is then compared with the timestamp acquired at the beginning of the interrupt handler. Three different interrupt handler scenarios were measured:

1. A kernel level interrupt handler. Invoking this handler is a local kernel call.
2. A user level interrupt handler in a task that is activated on the interrupted processor. Invoking this handler requires an upcall into the user space.

[1] A small optimization can be made during the register save operation because the register save is occuring during a well known point in the thread scheduler. This allows less registers to be saved than the whole processor context.

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

Table 5.4: 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. Performing this operation requires two upcalls into user space and a system call.

Table 5.4 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 kernel. No special setup is required to upcall into user space. Also, the user level register context can be saved in a cheaper fashion, since it will be directly restored by the 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 user level context save area, and an upcall must be performed 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 service 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.

5.5 Task signalling performance

The asynchronous task signalling facility is used throughout the kernel to provide synchronization and event passing between tasks. The performance of this facility is an important factor for global thread synchronization and interprocess communication.

The signalling mechanism relies on interprocessor interrupts or a system call to deliver signal messages. The sequence of steps performed by a signal invocation is summarized as follows:

1. If the destination task is running on a remote processor, deliver it an interrupt and return.
2. If there is an idle processor, deliver it an interrupt and return.
3. Otherwise, make a system call to perform the signal.

The remote interrupt mechanism allows the local processor to offload all of the queuing and task invocation work to the destination processor. The initiating processor can continue with its own work while the signal is processed. If signal invocation makes a kernel call to perform its work, the local processor handles the signal message queuing and readying, and dispatching of the destination task.

A benchmark was constructed to measure the difference between a software interrupted signal message and a kernel mediated signal message. Two tasks were created. One task contains a signal handler that will be invoked by the other task. Three test cases were measured: a kernel mediated signal, a software interrupt with the remote task activated on the interrupted processor, and a software interrupt with the remote task not activated on the interrupted processor. The results for a single invocation are shown in Table 5.5.

	Time (usec)
kernel signal	32.4
user active signal	13.2
user inactive signal	15.6

Table 5.5: Task signalling invocation latencies.

The kernel mediated signal is more than twice as expensive as the software interrupt versions. This time is mostly consumed by the task switch that must occur on the local processor. The initiating task must be switched out, and the destination task must be switched in a delivered the signal. Sending a signal to a remote processor that is running the task is the least expensive. No task switching is required, but there is the cost of handling the interrupt and saving the interrupt thread's context. Interrupting an idle processor is only slightly more expensive. The destination task must be activated, but there is no thread state to save.

5.6 Interprocess communication performance

The performance of the interprocess communication primitives is a vital factor for high-throughput client/server applications. In these types applications, IPC performance is the most common bottleneck.

The user level IPC libraries make extensive use of shared memory and scheduling primitives that do not require kernel intervention. Two experiment programs were constructed to measure the performance of the IPC libraries under various conditions. The first set of experiments test the local interprocess communication primitives. Then a second set of experiments test the global IPC primitives. These libraries are sufficiently different in implementation that it makes sense to test and analyze them separately.

5.6.1 Local communication

Both synchronous and asynchronous port based communication was tested. The performance of these libraries weighs heavily on the performance of the low level thread scheduling modules. The operations performed by the local IPC primitives are mostly thread context switching and enqueue/dequeue operations.

The test program creates a port and some threads to communicate across the port. The number of client and server threads was varied, as well as the number of physical processors. A 4 byte message is passed several thousand times each case, and the individual results averaged.

The first set of tests measures the asynchronous port performance. A port with a 20 element queue was created. The results of these tests are shown in Table 5.6. The first test creates 1 sender thread and 1 receiver thread (denoted 1-send/1-recv). While the average send/receive time for this case is 13.2 microseconds, the actual latency between a single send/receive pair is much higher. The 13.2 microsecond time is achieved because the sending thread can fill up the port queue at full speed, and then transfer control to the receiver thread which can spend all of its time draining the queue. Thus the overall time is greatly reduced due to the amortization of message buffering in the queue.

The addition of more processors to this case does not help much. In theory, the sender should be able to supply data fast enough to keep the receiver occupied. However, by examining the port status descriptors at various intervals during the benchmark, it appeared that the port remained full during most of the computation. This caused senders to block, thus forcing context switches between the senders and receivers. A more balanced workload on the sender side would have helped reduce this problem.

The synchronous message passing case requires much more work per transaction, because the sender always blocks waiting for a reply. Therefore the 1-send/1-recv/reply uniprocessor case is heavily bounded by the thread scheduling performance. Adding more processors causes additional work to be done. This appears to be due to scheduling overhead. When a client thread places a message on a port queue, the client will deliver a remote interrupt to the next

	1 CPU	2 CPU	3 CPU	4 CPU
1-send/1-recv	13.2 usec	12.6 usec	12.7 usec	12.7 usec
10-send/10-recv	14.3	12.0	10.3	10.0
1-send/1-recv/reply	32.6	32.9	33.0	33.0
10-send/10-recv/reply	18.1	17.6	17.0	16.8

Table 5.6: Performance of asynchronous and synchronous local ports, 4 byte data message.

available idle processor to wake the server thread. Since there is only one server thread and one client thread, the cost of managing the context switch on the local processor turns out to be less expensive than delivering an interrupt to a remote processor. Interrupting a remote processor to run a thread causes that processor to examine the ready queue, increasing lock contention for other processors in the system.

If the client and server sides have more threads to work with, as in the 10-client thread 10-server thread case, performance is greatly improved because the message queue can be maintained at a non-empty state, and servers can read messages as fast as clients can place them.

However, none of these local port cases show much improvement when more processors are added. Most of the time spent seems to be wasted on lock contention. There are two hot spots: the port spin-lock or the thread scheduler ready queue lock. The workload that the client and server threads perform are basically null operations, so most of the execution time is spent chasing after locks within the thread scheduler.

5.6.2 Global interprocess communication

This section measures the performance throughput of the global interprocess communication service. This test combines many of the primitive system services: remote interrupt dispatching, task signal dispatching, global semaphores, and task and thread scheduling.

The global IPC test cases create two address spaces: a server task, and a client task. The server task allocates a port descriptor containing 20 message buffers and advertises it on the

	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
1-send/1-recv/reply	145	124	124.1	123.8
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 5.7: IPC performance for asynchronous and synchronous global ports, 4 byte data message.

nameserver. A number of server threads are created to listen for messages on the port. The client task queries the nameserver for the port descriptor and creates a number of client threads to bombard the server with messages.

Table 5.7 contains performance results for various combinations of processors and threads. The simple case of 1-send thread and 1-receive thread synchronous send/receive/reply demonstrates the worst case performance of 121 microseconds per interaction. Each iteration requires two processor relinquishments and two upcalls. This figure is improved slightly in the 2 processor case, because the client and servers reside on separate processors most of the time. Invoking the remote task to signal a message is done via a software interrupt, and an address space switch is not required.

However, synchronous performance does not increase by the expected amount when more threads and processors are added. In fact, performance is reduced when more processors are added to the 10-sender 10-receiver case. In a uniprocessor system, the all the sender threads block, then all the receivers return replies. However, when processors are added, the thread and task schedulers seems to constantly jump around in a form of hysteresis, causing more scheduling events to occur than optimal.

The asynchronous message transfers are much faster overall because of the reduced context switching requirements between the client and server. The client threads have no problem keeping the port queue full of data for the server threads. Increasing the number of threads

	Async IPC	Sync IPC
total IPC calls	200,000	300,000
kernel calls	16,000	56,000
user interrupts	68,000	108,000

Table 5.8: IPC primitive breakdown.

results in an apparent slight performance hit. This is possibly due to the increased number of thread descriptors being managed throughout the system. Both the single thread and multiple thread case are able to keep the port queue full, so have multiple threads on a single processor does not help.

As seen in the local case, performance sharply declines as more processors are added to the system. The reason for this again is lock contention. The workload performed by the client and server threads is null, so all their effort is spent trying to access the port queue. Much of the overall processor time is spent waiting on the port queue.

Kernel Intervention

The global interprocess communication library attempts to reduce the number of kernel interactions by using a task signalling mechanism that can send event messages without kernel intervention. To illustrate this, the number of low level primitive invocations was measured for the 2-thread server, 2-thread client, benchmarks shown above for synchronous and asynchronous IPC.

These measurements are broken into the following three categories: the total number of IPC primitive calls, the number of kernel mediated calls, and the number of software interrupt mediated calls. Table 5.8 summarizes the results for asynchronous and synchronous benchmarks.

These results demonstrate the reduced dependency on kernel interaction of the user level IPC library. Better results could be achieved using an improved task and thread scheduler. Such a scheduling system could properly schedule threads that are communicating so as to reduce the number of context switches and event messages.

	Time (usec)	Instructions
`vm_alloc()` 4KB	38.5	389
`vm_alloc()` 64KB	149	1516
`vm_alloc()` 1MB	2080	20637
`vm_free()` 4KB	23.3	303
`vm_free()` 64KB	145	1575
`vm_free()` 1MB	2140	21745
`vm_share()` 4KB	30.1	303
`vm_share()` 64KB	116	1126
`vm_share()` 1MB	1650	14682
`vm_move()` 4KB	34.0	324
`vm_move()` 64KB	132	1264
`vm_move()` 1MB	1720	15353

Table 5.9: Virtual memory operation execution times.

5.7 Memory management performance

This section performs some experiments to measure the execution time of various virtual memory system calls. The system calls tested are: `vm_alloc()`, `vm_free()`, `vm_share()`, and `vm_move()`. Each call is benchmarked by repeatedly calling the routines using region sizes of 4KB, 64KB, and 1MB.

The calls were all performed within a single thread, so no parallel performance numbers were measured. However, one can deduce that the parallel performance of these operations will be relatively poor due to high lock contention. Approximately 90% of the work done by these operations occur within a critical section, protected by the address map lock, or by the task descriptor lock. This lock contention is on a per-address space basis, so parallel invocations will happily coexist if allocations occur amongst disjoint address spaces.

Each call was executed a number of times in a tight loop. Table 5.9 shows the average execution time and instruction counts for each call, for varying region sizes.

The time to execute each call on a single page size of 4KB is relatively high compared to the time for larger page sizes. This is due to the high setup costs for each call. Using multiple

Frame size (bytes)	Time (sec)	efficiency
128	18.6	45.1%
512	9.03	92.9%
1024	8.71	96.3%
1500	8.57	97.9%

Table 5.10: Time to transfer 10MB of data over Ethernet.

page sizes allows the per-page cost to be amortized over a large number of pages, thus reducing the relative execution time drastically.

5.8 Ethernet driver performance

A simple test program was constructed to test the performance of the Ethernet device driver. The test program runs on two machines connected by Ethernet. One machine continuously sends data to the receiver machine. 10 megabytes of data is transmitted between the machines using a range of Ethernet frame sizes. Table 5.10 summarizes the results of the test, showing the percentage efficiency compared to the 10Mbps Ethernet speed.

Chapter 6

Related Work

This chapter examines recent work in the field of multiprocessor operating systems, and how it relates to the design of the Raven kernel. The operating systems topic is rather broad. The research emphasis in recent years has turned away from implementing feature rich environments, to finding more efficient and streamlined ways of doing things. For example, rather than building an operating system that contains everything that anyone would ever need, recent research in the field identifies the basic operating system components and improves upon them. The Raven kernel was designed and implemented in the same spirit.

Amongst these basic operating system components that are particularly relevant to multiprocessor systems are critical section management, thread management and scheduling, and interprocess communication. Much attention to these areas has been spent in recent years to improve the performance and their characteristics.

6.1 Low-level mutual exclusion

There is a large body of research work related to the implementation and analysis of mutual exclusion synchronization primitives on shared-memory multiprocessors. Early work in this area detailed software algorithms where the only atomic operations provided by the hardware are memory read and write, [Dij65] [Knu66]. The main disadvantage of these software dominated approaches is their inefficiency. Since then, however, more powerful atomic operations supplied by hardware has made mutual exclusion more efficient. Before looking at these operations, consider where mutual exclusion is used.

Operating systems rely on low-level mutual exclusion algorithms to protect against parallel access to system data structures and hardware devices. In a uniprocessor system, a common use for mutual exclusion are to protect data structures that are shared between interrupt handlers and device drivers. Disabling preemption by masking interrupts around sensitive code is an effective way to provide this capability. However, the addition of multiple processors in a system complicates the situation. Disabling interrupts alone does not stop other processors in the system from accessing the protected resource.

The technique of spin-locking has long been an elementary operation that can provide mutual exclusion between separate processors. The algorithm is to spin-wait for a shared lock variable to become available, and then mark it unavailable, thus claiming the lock. When the critical section is over, the lock variable is marked available again. In many systems, the lock "acquire" stage requires an atomic operation, such as memory exchange or test-and-set.

While lock contention can be reduced by carefully designing critical sections to minimize their overlap, spin-locking can be wasteful of available computing resources because of the busy wait nature of the algorithm. In addition to completely consuming local processor cycles, the spinning read/write cycle of test-and-set can generate a constant barrage of memory transactions. In a shared memory multiprocessor environment where main memory accesses share a common bus, this activity can degrade the performance of all processors in the system. Experiments in [And89] show that this algorithm is worthwhile for systems with less than six processors. However, this experiment only shows the impact of memory bus contention against spinning processors, and not processors doing other work.

A simple optimization to the spin lock involves the use of memory caching to reduce global memory bus contention. This technique relies on cache coherency to maintain proper copies of lock variables. Rather than accessing the lock variable directly in memory, the lock variable is read into the local processor cache. All spinning occurs out of the cache. When a lock value changes, the system's cache coherency algorithm propagates the new value to the appropriate caches. On some systems however, the cost of maintaining cache coherency can become a bottleneck. The Raven kernel implements the above method because the Hypermodule hardware and the 88100 provide the necessary cache coherency protocols.

In the absence of cache coherency, spinning memory transactions can be reduced by using a backoff algorithm. If acquiring a lock fails, then delay for a period of time and try again. This algorithm is similar to the Ethernet's exponential backoff [MB76]. However in this case, performance can still be poor for a small number of spinning processors because the lock acquire stage will continue to backoff even when the lock is released. The waiting processor remains consumed by the backoff delay. This algorithm is not appropriate in the current implementation of the Raven kernel because of the small number of processors in the system. Also, the overhead required to implement backoff timing would consume a high proportion of lock acquire stage.

The technique of *queuelocks* has been shown to reduce memory bus contention even in the presence of many processors, [And89], [GT90], [MCS91a]. The idea behind queuelocks is to make each thread spin only for one other thread to release a lock. If one thread waits for a lock holder, another thread will wait for the first waiting thread. This relationship allows each thread to spin on a different memory location. However, the advantages of this method are offset by the additional overhead costs in the bookkeeping of lock queue data structures. This overhead is not justified in the current implementation of the Raven kernel because of the small number of processors.

Experiments have shown that spin-locking on global locks does not scale well beyond eight processors [And89] [KLMO91]. While memory bus contention is significantly reduced using caching, performance eventually becomes bounded by spinning processor cycles. This bottleneck appears to become a factor in systems with more than eight processors. Also, the cost of cache coherency in some systems can impose other bottlenecks to the system.

An alternative to spin-waiting involves techniques based on wait-free synchronization [Her91] [Her90] [MCS91b] and data structures known as lock-free objects [Ber91] [MP91]. The idea here is optimistic: allow concurrent data accesses without blocking. After a modification to a data structure is made, the algorithm checks to see if structures are consistent, and if not, the operation is rolled-back. However, these algorithms require additional hardware support beyond simple test-and-set or compare-and-swap to operate efficiently. The Hypermodule and 88100 instruction set does not directly support these algorithms, but they can be constructed using more primitive features.

6.2 Threads

Operating systems have long provided lightweight threads of control to support a general purpose concurrent programming model for address spaces. Threads are used in uniprocessor systems as a structuring aid and to help overlap input/output with computation. In multiprocessor systems, threads are also used to exploit true parallelism. Several techniques for thread management and their associated performance characteristics are measured in [ALL89].

Thread management is usually implemented as either a kernel level service, or in user address spaces as a threading library linked with executables. Kernel level threads benefit from better integration with the other kernel supported services, such as priority scheduling and input/output. The kernel maintains control over all scheduling decisions, so thread priorities can be obeyed across address spaces. Threads performing input/output system calls or interprocess communication can be properly blocked and rescheduled as their operations complete.

Traditional microkernel architectures such as Mach [TR87] and the V-Kernel [Che88] demonstrate the use of kernel threads. However, the performance of these systems inherently suffer due to the costs of crossing user/kernel boundaries to perform thread management functions. Every single thread context switch and library call requires a kernel call. In addition, since the kernel level interface is usually intended to be used by all varieties of user programs, the threading interface must be general purpose and cannot take advantage of any local special purpose optimizations.

The Raven kernel implements threads at the user level to avoid the above performance problems and provide more convenient interfaces to the user.

Pure user level threading implementations can perform thread management operations at least an order of magnitude faster than their kernel level counterparts. The cost of invoking thread operations is at most the cost of a local procedure call, which in some cases can be optimized to inline macro routines. Many such user level threading packages exist for the Unix environment [Gol86] [Doe87] [SM90]. These packages multiplex a number of user defined threads on top of a single kernel implemented process. While these packages avoid kernel invocation for most thread services, they introduce problems of their own:

- Blocking system calls stop all threads in the address space. While `select()` can be used to alleviate this problem for routines such as `open()`, `close()`, `read()`, and `write()`, other potentially blocking calls such as `mkdir()`, `rename()`, `ioctl()`, `stat()` and asynchronous events such as page faults are more difficult to deal with.

- Poor performance resulting from improper scheduling decisions imposed by the kernel during low-level thread mutual exclusion. Spin locks are commonly used between threads in the same address to provide lightweight mutual exclusion (blocking semaphore management is too heavyweight for some operations). If a spin lock is acquired and held by a thread which is subsequently switched out, other threads in the system trying to acquire the lock will hopelessly busy wait until the holder is allowed to complete its critical section. A thread can be switched out for a number of reasons, such as the expiry of a time quantum, or the arrival of other external interrupting conditions.

One technique which tries alleviate parts of the above problems allows lightweight user level threads to be executed on top of kernel supported middle-weight threads of control. This technique is used by Mach's C-Threads library [CD90] and SunOS's LWP [PKB$^+$91] [SS92]. The user level threads reside as data structures in the user level address space. Kernel level threads are used as virtual processors to execute the user level threads. User level threads can be successfully scheduled around blocking system calls, but low-level synchronization problems still exist.

To solve the scheduling problems that low-level synchronization code introduces requires some special support by the operating system that can detect when it is inappropriate for context switching to occur. In the Psyche operating system [SLM89], *first class* user level threads [MSLM91] share locking information between the lock management routines and the kernel. Soon before preemption is required, the kernel provides the user level with a *two-minute warning* flag. User level synchronization code can check this flag prior to acquiring a lock, and voluntarily relinquish control to the kernel if it deems necessary. While this technique does not completely remove inappropriate scheduling decisions, the number of them is significantly reduced. The solution implemented in the Raven kernel eliminates this problem.

The scheduler activations technique [ABLL92] provides a more sure-fire way of preventing lock synchronization problems by allowing critical sections to complete before preempting the processor. This is the same idea used by the Raven kernel, but scheduler activations implements it quite differently.

When an event occurs that would normally cause preemption during a critical section, an upcall into the user space occurs. The user level kernel recognizes that a critical section is in progress by checking the interrupted address, and jumps directly to the code that will complete the critical section. This code is in fact a copy of the original, except that the tail end contains a relinquishment call to the scheduler.

Instead of upcalling to the user level during a critical section, the Raven kernel defers the preemption by setting an "upcall deferred" bit and returns to the user execution. At the end of the critical section, the user checks the bit and relinquishes if it indicates so.

The scheduler activation technique allows lock operations to be as efficient as possible, because they do not require to manage any preemption status variables. However, additional overhead is required by the upcall handler to dispatch control to the appropriate copy of the critical section. The instruction pointer at the time of preemption must be examined and compared with a list of critical section handlers. This makes nested locking and critical sections with multiple return points difficult to manage. Special compiler support is required to automate the code copying. The upcall dispatcher in the Raven kernel avoids this difficulty altogether.

6.3 Interprocess communication

In traditional operating systems, the kernel has mediated the interprocess communication mechanism. User programs wishing to communicate with remote services were required to invoke kernel operations to perform the communication protocol. This process is now seen as being inefficient due to the increased relative costs of crossing user/kernel boundaries compared to simple procedure calls. Interprocess communication systems are concentrating on reducing data copying costs and latencies by using memory mapping techniques and software supported scheduling mechanisms.

Recent versions of the Mach 3.0 kernel have improved on typical kernel mediated IPC implementations by introducing the *continuation* [DBRD91]. Continuations facilitate the passage of execution control through the kernel scheduling primitives by allowing execution context to be handed off to another thread. This can eliminate scheduling overhead and queuing within the kernel. In the fast path best case, the sender thread executes within the context of the receiver thread.

The Lightweight Remote Procedure Call (LRPC) [BALL89] mechanism also involves kernel intervention to pass messages between client and server, but takes advantage of architectural details of the DEC SRC Firefly. Execution progresses through the kernel and into the remote address space using a special purpose stack structure that is used by both the sender and receiver. Frequently used parameters are cached in processor registers.

As with the thread scheduling implementations discussed above, recent interprocess communication design have been removed from the kernel and implemented at the user level. This allows users to directly invoke IPC primitives without the added costs of crossing user/kernel boundaries. A level of indirection is removed because instead of invoking the kernel, the user now directly communicates with the remote process.

The URPC technique [BALL90] relies on pair-wise shared memory between the client and server processes to pass message data. The message delivery system is controlled by low priority threads that poll the message queues looking for work. The threads only poll while the system is idle. While this polling mechanism can produce low latency message transfers, this best case scenario only occurs when there is no other work for the system. Therefore, this model is not appropriate for systems with constant workloads. The Raven kernel is intended to be used in applications where good IPC performance is required under load.

The split level scheduling technique and memory mapped streams implemented in the continuous media system [GA91] describes one way of using shared memory and scheduling techniques to reduce communication bottlenecks. Split level scheduling allows scheduling decisions to be made at both the kernel and user level. A shared data structure between the user/kernel level facilitates the communication of thread scheduling information. This sharing of information is similar to the Raven kernel, but in Raven, the amount of information shared is much less. In order to properly honour thread priorities and real-time events in remote address spaces, much more scheduling detail must be shared between the user and kernel.

The *address-valued signal* mechanism introduced in [CK93] describes a hardware assisted low level signalling mechanism. This is a hardware solution to the Raven kernel's task signalling facility. The hardware maintains a FIFO queue of signal interrupts, making it especially easy and efficient for one address space to send a low level synchronization message to another address space. Virtual addresses are used to direct the signal to any particular address space, and the hardware handles the rest. The remote processor is interrupted and presented the next signal on the hardware FIFO queue. Higher level communication protocols, such as RPC, can be built with this low level mechanism.

Chapter 7

Conclusion

7.1 Summary

This report presented a new multiprocessor operating system, known as the Raven kernel. The Raven kernel provides a multitasking, time sliced, environment for user level programs to execute in. The system provides the notion of tasks, virtual memory, threads, and interprocess communication. However, unlike traditional microkernel architectures, the Raven kernel implements many of these services completely in user space. The motivation behind this design was to improve system performance by reducing the number of user/kernel boundary changes.

An overview of the runtime environment used for the Raven kernel was provided. The hardware platform used is the Motorola 88100 four processor Hypermodule, with 32MB of shared memory. A special kernel debugger based on gdb allows kernel code to be interactively debugged and tested.

The implementation of the kernel services was then described. The kernel consists of several modules that provide three main services to user level programs:

- Task management (allocation and scheduling).
- Virtual memory management (memory allocation and mapping).
- Low level upcall dispatching.

The description of the user level kernel followed. The following operating system services were implemented completely at the user space:

- Preemptive thread scheduling. Threads migrate from processor to processor in an effort to balance the load.

- Device interrupt handlers. Hardware interrupts are efficiently funnelled up from the kernel to user level registered handlers.
- Semaphores, used to synchronize events between threads in remote address spaces.
- Interprocess communication. A synchronous and asynchronous port based messaging scheme was implemented using shared memory queues and low level synchronization routines.
- A nameserver database for fast lookup of global names.

A set of primitive low level event signalling routines made interprocess communication and scheduling possible without kernel intervention:

- The `intr_remote_cpu()` processor interruption facility delivers hardware interrupts to remote processors.
- The `task_signal()` signalling facility sends asynchronous event message to remote tasks.

The performance results show that reduced kernel intervention and improved performance is possible using these techniques. However, the trade-off between performance is stability. To reduce communication bottlenecks between address spaces, extensive use of shared memory regions is employed. These shared memory regions are left exposed to malicious processes or errant program behaviour. Therefore, the system is suited towards dedicated environments where programs are trusted.

7.2 Future Work

The Raven kernel is intended to provide the basis for an efficient and lightweight parallel programming environment for high-speed parallel applications. Work will be continued in this area to improve performance and add functionality.

The performance results showed that scheduling and lock contention overhead contributed to most of the interprocess communication bottleneck. The current task and thread scheduler makes scheduling decisions based solely on the round-robin fairness scheme. A better task and thread scheduler could be designed which would identify communicating threads, and try to schedule them together to reduce context switching bottlenecks.

Appendix A

Kernel system call interface

This appendix summarizes the supervisor level system call interface that is available for general purpose user programs. The first section presents the system calls intended by use for user level kernels. The second section presents the interface intended for user application programs.

A.1 System calls provided to the user level kernel

This section summarizes the system calls provided to user level kernels. User application programs should not call these routines directly.

A.1.1 Task management system calls

```
int task_timer_event( int wakeup_time );
int task_request_cpu();
int task_relinquish_cpu();
int task_intr_relinquish_cpu();
int task_cleanup( int task_id );
```

A.1.2 Interrupt management system calls

These system calls register and enable interrupts to the user level.

```
int intr_register_user( int intr_vec );
int intr_deregister_user( int intr_vec );
```

A.1.3 Exception management system calls

These system calls register and enable exceptions to the user level.

```
int excp_register_user( int intr_vec );
int excp_deregister_user( int intr_vec );
```

A.1.4 Global semaphore management

```
int kernel_sem_enqueue(int sem_id);
int kernel_sem_dequeue(int sem_id);
```

A.2 System calls provided for application programs

This section summarizes the system calls provided by the supervisor kernel for application programs.

A.2.1 Task management system calls

```
int task_suspend( int task_id );
int task_resume( int task_id );
int task_signal( int task_id, int signal, int user_data );
int task_create( int *task_id, int priority, TASK_EXEC_HDR *hdr,
                 int code_region, int data_region, TASK_ARGS *args );
int task_destroy( int task_id );
int task_info( int task_id );
```

A.2.2 Virtual memory management system calls

```
int vm_alloc(int *region_id, int task_id, void **addr, int size, int attrb );
int vm_free(int region_id);
int vm_move(int region_id, int dest_task_id, void **dest_addr, int dest_attrb);
int vm_share( int src_region_id, int *dest_region_id, int dest_task_id,
              void **dest_addr, int dest_attrb );
int vm_map_device( int *region_id, void *phys_addr, void **addr, int size );
int vm_unmap_device( int region_id );
```

A.2.3 Console input/output

```
void kprint(char *str);
```

```
int kgetstr(char *str, int len);
```

A.2.4 Program loader

```
int read_exec_hdr(char *filespec, TASK_EXEC_HDR *hdr);
int read_exec( int code_region, int data_region );
```

Appendix B

User kernel library call interface

This appendix summarizes the user level kernel library call interface that is available for general purpose user programs. All of these calls are prototyped in the `<user/threads.h>` header file.

B.1 Thread management

```
int thread_me();
int thread_sleep( unsigned long sleep_time );
int thread_suspend();
int thread_resume( int id );
void thread_sched();
int thread_create( int *thread_id, void (func)(), char *name, int priority,
                   int stacksize, int ready, int num_args, ...);
int thread_destroy( int id );
```

B.2 Synchronization primitives

```
/* Spin lock routines */
void lock_wait(int *lock);
void lock_free(int *lock);

/* semaphore routines */
int sem_wait( int sem_id, int no_block );
int sem_signal( int sem_id );
int sem_reset( int sem_id, int count );
int sem_count( int sem_id );
int sem_create( int *sem_id, int count, int global );
int sem_destroy( int sem_id );
```

B.3 Asynchronous Send/Receive port IPC

```
int port_send( int port_id, char *msg, int msg_size, int no_block );
int port_recv( int port_id, char *msg, int *msg_size, int no_block );
```

```
int port_create( int *port_id, int max_msg_size, int queue_size, int global );
int port_destroy( int port_id );

/* for global ports only -- these do memory mappings of port queues */
int port_reference(int port_id);
int port_dereference(int port_id);
```

B.4 Synchronous Send/Receive/Reply port IPC

```
int rpc_port_send( int port_id, char *send_data, int send_len,
                   char *reply_data, int *reply_len );
int rpc_port_recv( int port_id, char **recv_data, int *recv_len,
                   char **reply_data );
int rpc_port_reply( int port_id, int msg_id, int reply_len );
int rpc_port_create(int *port_id);
int rpc_port_destroy(int port_id);

/* special calls for global rpc_ports */
int grpc_port_create(int *port_id, int max_data_len, int num_msg_bufs);
int rpc_port_dereference(int port_id);
```

B.5 Nameserver

```
int nameserver_find(char *str, int *data);
int nameserver_register(char *str, int data);
int nameserver_deregister(char *str);
```

B.6 User level memory management

```
/* Zone memory allocator routines */
void *zone_alloc( int zone_id );
int zone_free( void *buf );
int zone_create( int *zone_id, int size, int alloc_size );
int zone_destroy( int zone_id );

void *malloc(int size);
#define free(buf) (zone_free(buf))
```

B.7 Interrupt and exception management

```
int intr_register( int intr_vec, void *handler );
int intr_deregister( int intr_vec )
```

```
int excp_register( int excp_vec, void *routine, int global );
int excp_deregister( int excp_vec );
```

Appendix C

Unix version

A Unix version of the user level interface was implemented to aid in the development and testing of user level programs. The Unix version implements a non-preemptive thread scheduler, properly integrated with Unix filesystem I/O using a `select()` wrapper.

The following function prototypes document the Unix version interface:

```
int thread_me();
int thread_sleep( long sleep_time );
int thread_suspend( int id );
int thread_resume( int id );
void thread_sched();
int thread_create( int *thread_id, void (func)(), char *name, int priority,
                   int stacksize, int num_args, ...);
int thread_destroy( int id );

int sem_wait( int sem_id, int no_block );
int sem_signal( int sem_id );
int sem_reset( int sem_id, int count );
int sem_count( int sem_id );
int sem_create( int *sem_id, int count, int global );
int sem_destroy( int sem_id );

int port_send( int port_id, int *msg, int msg_size );
```

```
int port_recv( int port_id, void **msg, int *msg_size );
int port_create( int *port_id, int msg_size, int queue_size, int attrib );
int port_destroy( int port_id );

/* for Unix I/O */
int Read( int fd, char *buf, int nbytes );
int ReadN( int fd, char *buf, int nbytes );
int Write( int fd, char *buf, int nbytes );
int WriteN( int fd, char *buf, int nbytes );
int Open( char *filespec, int flags, int mode );
int Close(int fd);
int Socket( int domain, int type, int protocol );
int Accept(int fd, struct sockaddr *addr, int *addrlen);
int Connect(int fd, struct sockaddr *name, int namelen);
```

Bibliography

[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.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10:53–79, February 1992.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[And89] Thomas E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *International Conference on Parallel Processing*, pages II–170–II–174, 1989.

[BALL89] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 102–113, Litchfield Park, AZ, 3–6 December 1989. Published as Operating Systems Review, volume 23, number 5.

[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.

[Bed90] Robert Bedichek. Some efficient architecture simulation techniques. *Winter 1990 USENIX Conference*, January 1990.

[Ber91] Brian N. Bershad. Practical considerations for lock-free concurrent objects. Cmu-cs-91-183, Carnegie-Mellon University, September 1991.

[BRG+88] David L. Black, Richard F. Rashid, David G. Golub, Charles R. Hill, and Robert V. Baron. Translation lookaside buffer consistency: A software approach. Cmu-cs-88-201, Carnegie-Mellon University, December 1988.

[CD90] Eric C. Cooper and Richard P. Draves. C threads. Technical report, Department of Computer Science, Carnegie Mellon University, September 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.

[Com84] Douglas Comer. *Operating system design, the Xinu approach.* Prentice Hall, 1984.

[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.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, September 1965.

[Doe87] Thomas W. Doeppner. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Department of Computer Science Brown University, Providence, RI 02912, June 1987.

[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.

[Gol86] Murray W. Goldberg. Pthreads. Technical report, Department of Computer Science, Univeristy of British Columbia, 1986.

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

[GT90] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, June 1990.

[Her90] Maurice Herlihy. A methodology for implementing highly concurrent data structures. *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, January 1991.

[JAG86] M.D. Janssens, J.K. Annot, and A.J. Van De Goor. Adapting unix for a multiprocessor environment. *Communications of the ACM*, September 1986.

[KLMO91] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. 13th SOSP.*, pages 41–55, Asilomar, Pacific Grove, CA, 13 Oct. 1991. Published as ACM. SIGOPS.

[Knu66] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, May 1966.

[MB76] R. Metcalfe and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, July 1976.

[MCS91a] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):21–65, February 1991. Earlier version published as TR 342, URCSD, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice UNIV, May 1990.

[MCS91b] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *PROC of the Fourth ASPLOS*, pages 269–278, Santa Clara, CA, 8-11 April 1991. In *CAN 19*:2, *OSR 25* (special issue), and *ACM SIGPLAN Notices 26*:4.

[Mot88a] Motorola. *MC88100 User's Manual.* Motorola, 1988.

[Mot88b] Motorola. *MC88200 User's Manual.* Motorola, 1988.

[Mot88c] Motorola. *MVME188BUG 188Bug Debugging Package User's Manual.* Motorola, 1988.

[Mot88d] Motorola. *MVME6000 VMEbus Interface User's Manaual.* Motorola, 1988.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 191–201, Litchfield Park, AZ, 3–6 December 1989. Published as Operating Systems Review, volume 23, number 5.

[MP91] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, February 1991.

[MSLM91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *PROC of the Thirteenth SOSP*, pages 110–121, Pacific Grove, CA, 14-16 October 1991. In *OSR 25*:5.

[PKB+91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. *Proceedings of the Usenix 1991 Winter Conference*, 1991.

[SLM89] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. A multi-user, multi-language open operating system. In *PROC of the Second Workshop on Workstation Operating Systems*, pages 125–129, Pacific Grove, CA, 27-29 September 1989.

[SM90] Inc. Sun Microsystems. Lightweight processes. *SunOS Programming Utilities and Libraries*, March 1990.

[SS92] D. Stein and D. Shah. Implementing lightweight threads. *Proceedings of the Usenix 1992 Summer Conference*, 1992.

[Sta89] Richard M. Stallman. *The GNU gdb debugger.* The Free Software Foundation, 1989.

[TR87] A. Tevanian and R.F. Rashid. MACH: A basis for future UNIX development. Cmu-cs-87-139, Carnegie-Mellon University, June 1987.