The Skidoo Real–Time Operating System

by

Thomas Jack Trebisky

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

Master of Science

In the Graduate College

The University of Arizona

2 0 0 2

Statement by Author

Signed: ____________________________

Approval by Thesis Director

This thesis has been approved on the date shown below:

Dr. Gregory R. Andrews Professor	Date

Acknowledgments

First of all I would like to thank my wife Ingrid and my sons Alexander and Paul. They willingly paid a price and did without the attentions of husband and father during the many hours I spent with the computer instead of with them.

I would particularly like to thank my advisor, Greg Andrews. I very well might not have begun taking graduate classes at all without his timely encouragement. I certainly would not have tackled this project without his support, and I am grateful for his investment of time and energy.

I appreciate the willingness of John Hartman and Mikael Degermark to serve on my thesis committee, and to make time in their busy schedules.

I thank my friend and colleague, Alan Koski, for many stimulating conversations, and for being a enthusiastic fan of the project. He kindly read an early draft of the thesis and asked me for more!

I thank my friend Steve West for much help and assistance over the years, but particularly for encouraging me when I was in the midst of the decision of whether or not to take classes and work toward yet another degree.

I thank Craig Foltz, director of the MMT Observatory, for his enlightened attitude toward my education and the inevitable conflicts and pressures it placed upon my job responsibilities.

Table of Contents

Table of Contents—*Continued*

List of Tables

List of Figures

ABSTRACT

Embedded systems have needs that are not adequately met by conventional operating systems. Skidoo is a new operating system especially tailored to support embedded systems. Independently scheduled threads are provided that synchronize using semaphores and condition variables. Threads share a common address space and communicate using shared variables. Fully preemptive scheduling meets the needs of hard real–time applications.

Chapter 1

INTRODUCTION

Embedded systems play an increasingly important role in modern society. They occur within automobiles, appliances, disk–drives, internet routers, weapons systems, and myriad other applications. Two things characterize embedded systems. First, their software is static and tailored to a specific mission. Apart from bug fixes and field upgrades, the software in an embedded system never changes. Second, the hardware in an embedded system is likely to be very restrictive. Many embedded systems are components in high–volume, cost–sensitive applications. Considerations of space, reliability, and power consumption often preclude the use of rotating disk drives.

Many embedded systems contain software in which strict time deadlines must be met in order to ensure proper operation; such systems are called real–time systems. A distinction is sometimes made between "soft" and "hard" real–time systems. A soft real–time system runs correctly if some statistical portion of time deadlines are met. A hard real–time system runs correctly only if every time deadline is met.

Skidoo[1] is a new operating system that can be used to build embedded systems. Skidoo provides threads, semaphores, and fully preemptive scheduling. At any time, the highest priority runnable thread is running, or a low latency transfer is in progress to set it running. Additionally, Skidoo offers timer facilities, convenient interrupt handling facilities, and a library of support routines and device drivers. Skidoo runs in protected mode on the x86 architecture and has very small memory requirements. It is adequate for building many real–time embedded applications.

Skidoo is intended to be a toolkit that is to be used to build a custom operating system. Each deployment of Skidoo is in fact an operating system that is tailored to the task at hand. By contrast, conventional operating systems such as Unix [3, 14] are designed to support general purpose computing. They support timesharing computing and provide facilities such as filesystems, virtual memory, and memory protection that are unnecessary in an embedded application. While it is possible to adapt a general purpose operating system such as Unix for use in real–time applications, a system that specifically addresses the requirements of real–time embedded applications can be smaller, simpler, and faster.

[1] The name "Skidoo" comes from a ghost town [15] on the west side of Death Valley National Park, in California.

1.1 Related work

RT-Linux [4] is an example of a conventional general–purpose operating system that has been enhanced to support real–time requirements. It does this by creating a real–time scheduling regime within Linux. The regular Linux kernel is then run as a low priority task under control of the real–time scheduler. This arrangement is adequate to support hard real–time requirements, but for some embedded systems, the software would be too large in terms of hardware requirements. It does have the advantage that software may be developed on the same system on which it will run.

Both Linux [21] and Solaris [22] offer POSIX 4 [7] real–time scheduling extensions. This makes it possible to implement real–time applications within the usual Unix timesharing environment. These are both still full–scale Unix systems however, so they would be inappropriate for hardware–restricted embedded applications.

It is worth noting that work is being done on the traditional Linux kernel to make it more suitable for some classes of real–time programming. In particular, preemption points are being provided within the kernel to limit the length of the code path within the kernel before an opportunity exists to switch context [2].

VxWorks [19] is a proprietary system without a Unix heritage. It provides a proprietary interface, but offers a large set of POSIX library facilities to aid porting Unix software. VxWorks expects that a Unix host system is used to develop software that then runs on a target system with distinct hardware and software. This requires a dedicated development system but allows the target system to be very spartan in terms of both hardware and software.

VxWorks has been a significant source of inspiration for Skidoo. Like VxWorks, Skidoo supports a set of threads running in a single address space and avoids utilizing address mapping and protection hardware. In contrast Skidoo is much simpler, yet it provides additional facilities such as continuations and race–free condition variables.

1.2 Outline of the Thesis

Skidoo was developed mainly as an excellent learning exercise. However, it is useful for getting real work done, as will be shown. It can be used as the basis for further research in embedded systems, as well as for the construction of specific applications.

Chapter 2 presents Skidoo and outlines the services that Skidoo makes available. Chapter 3 describes the implementation of Skidoo, discussing the steps taken and the decisions that were made. Chapter 4 discusses a test suite that was developed to exercise Skidoo, as well as experiments and applications that demonstrate its utility. Chapter 5 gives a summary of what was accomplished, what could have been done differently, and what has been left undone. The Appendix gives details of the programming interface to Skidoo.

Chapter 2

SERVICES

This chapter gives a survey of the services provided by Skidoo. The two central features of the Skidoo kernel are threads and binary semaphores. The most important feature of threads is the ability to block and thus to be independently scheduled. Semaphores provide an entity that can be used for synchronization. An incredible amount of work can be accomplished given just threads and semaphores. The rest of the Skidoo kernel provides condition variables, continuations, and access to essential hardware via timers, interrupts, and basic device drivers. The actual kernel routines are described in Appendix A.

2.1 Threads

A thread is an independently scheduled flow of execution. Each thread has a private stack, and a small amount of state, which consists of a set of flags and register values needed to resume the thread after it has blocked. Notably, a thread does not have a private address space (other than a stack). All threads share a common global address space and communicate using shared variables.

A thread in the Skidoo kernel is either ready or blocked. A thread only becomes blocked when it blocks itself. Once blocked, a thread is not eligible to be run until it is unblocked. A thread may be unblocked directly by another thread, or more commonly via a semaphore. With the exception of interrupt handlers, all code runs on behalf of some thread. Each thread is assigned a unique priority. For clarity, the terms "more urgent" and "less urgent" are used rather than "higher" and "lower" priority. In fact, priorities with larger numerical values are less urgent.[1] At all times, the current thread is always the ready thread with the most urgent priority.

Special care was taken to handle the case where a thread is unblocked from interrupt code. The difficulty arises in the case where a thread more urgent than the one currently running is marked ready to run. When this happens, the thread that was running when the interrupt occurred is left suspended, and the kernel resumes the more urgent thread when interrupt processing is finished.

2.2 Preemptive scheduling and priorities

In the Skidoo kernel, each thread must be assigned a unique priority when it is created. At all times, the thread with the most urgent priority that is ready to run is

[1] This choice is entirely arbitrary and mimics the ordering used in VxWorks.

running, or is in the process of being made to run. This scheduling policy is explicitly unfair: if the currently running thread never blocks, no thread of lower priority will ever run. This policy is called strictly preemptive scheduling. It is an error for two threads to be assigned the same priority. The system could have been designed to have some special defined behavior in this case (such as time slicing), but this has not been done. Providing fairness in this special case is not necessary in embedded systems and would only serve to complicate the kernel.

2.3 Semaphores

Threads together with some synchronization facility [18] provide a sufficient basis to build significant applications. Skidoo provides simple binary semaphores [1] as its fundamental synchronization primitive.

Semaphores may be created with an initial value of zero or one. Typically, semaphores with an initial value of zero are used for signaling, and semaphores with an initial value of one are used for mutual exclusion. The P operation is herein denoted "blocking" on a semaphore. When the value of a semaphore is zero, the "blocking" operation in fact blocks and places the current thread on a list associated with the semaphore. When the value of a semaphore is one, the "blocking" operation changes the value to zero and keeps running. The V operation is herein denoted "unblocking" a semaphore. When the value of a semaphore is one, the "unblocking" operation does nothing.[2] When the value is zero and the list of waiting threads is non–empty, one of the waiting threads is unblocked; otherwise the value of the semaphore is changed to one.

Signaling semaphores are commonly associated with a single thread as a private semaphore. Under this convention, only one thread uses the semaphore to await a signal. A mutual exclusion semaphore is not associated with any thread in particular, but rather with some resource that requires locking.

2.4 Condition variables

Condition variables can be viewed as a toolkit for building monitors [9]. In Skidoo they are provided as a higher level synchronization facility than semaphores. In essence a condition variable is a mutex semaphore and a signaling semaphore handled together as a unit as in POSIX 4 [7]. To allow synchronization in device drivers between interrupt handlers and thread context running the driver, a special form of condition variable is provided that disables processor interrupts as a form of mutual exclusion. To use a condition variable, a thread will acquire the mutex semaphore, check the resource locked by the mutex, and upon finding that it must wait for the

[2]An alternative would be for the unblocking operation to delay if a semaphore is set to one, but this would produce grave difficulties if the unblock was being done from within an interrupt routine.

condition of the resource to change will perform a “wait” on the condition variable. The “wait” operation will block the thread and release the mutex in an atomic operation, preventing race conditions. To unblock the thread when the condition of the resource changes, another thread acquires the mutex and uses the “signal” operation. In the case of an interrupt routine, acquisition of the mutex is implicit in that interrupts are blocked while in the interrupt handler.

2.5 Continuations

Continuations are provided as a more efficient alternative to the usual blocking semantics. The idea of a continuation is taken from the Mach Operating System [6]. A continuation provides a streamlined way for a thread to block and specify a point of resumption. When a thread blocks with a continuation, it abandons its context and processor state (including register values, and the stack). When the thread unblocks, it comes alive in the continuation function, as if it were starting up anew. Most Skidoo facilities which may block provide the option to block with a continuation, as well as the conventional return from the blocking call.

When a thread blocks with a continuation, it specifies a function to begin executing in when it is unblocked. When the thread does unblock, rather than returning in the usual way from the block call, it resumes by calling the specified function.

Using continuations requires some reorganization of code. A function that was designed to use a traditional blocking call will need to be partitioned into a function which performs some setup and then blocks, and a function which handles the event as a continuation. In many cases the event handling function will end by making a blocking call and again specifying itself as a continuation. This sort of tail recursion is a natural way to code functions that should be activated periodically after a fixed delay.

Figure 2.1 contains an example of a function using tail recursion to achieve periodic activation. A new thread is created using the `thr_new()` call. This thread prints a message, then specifies itself as a continuation after a delay of 25 clock ticks. The thread is blocked until the number of ticks elapses, and then runs again printing the message. This goes on forever, or until the thread is destroyed.

The advantage of continuations is that they are very lightweight. When a thread blocks specifying a continuation, its context can be abandoned—no registers need to be saved. The mechanism used to resume using a continuation is the same one that is used to launch new threads. In essence a thread is being launched anew each time it resumes with a continuation. It would be possible to abandon the stack and allocate a new one (as is done in Mach), but it is actually more efficient, although less frugal with memory, to retain the stack.

```
void
ticker_init ( void )
{
        (void) thr_new ( "tick", tick_fn, (void *)25, PRI_TICK, 0 );
}

void
tick_fn ( int delay )
{
        printf ( "Kilroy was here!\n" );
        thr_delay_c ( delay, tick_fn, delay );
}
```

FIGURE 2.1. Tail recursion using a continuation

2.6 Timer facilities

A timer is a hardware device that provides interrupts at a programmer–defined interval. A set of timer facilities provides convenient access to the available timing signals. At this time, the timer ticks at a nominal rate of 100 Hz.

A routine to handle timer interrupts is a standard part of Skidoo. This default routine keeps track of time and supports a delay service. The delay service allows the current thread be blocked until a specified number of timer "ticks" have passed. The default routine may be augmented by a user supplied C function that will also be called each time the timer "ticks". This routine is called at interrupt level and should be short and carefully coded.

A common use of the ability to connect a user supplied routine to the timer interrupt is to produce periodic thread activations. By subdividing the basic clock and using semaphores to unblock waiting threads from interrupt level, accurate periodic activations may be accomplished. Applications that require extremely accurate timing (such as waveform generation), may perform crucial processing in such a clock function.

2.7 Interrupt facilities

It is very useful to be able to connect arbitrary C functions to interrupt sources. Providing a convenient facility for doing this relieves individual device drivers from the machine dependent complexities of manipulating interrupt hardware. This is a real benefit for embedded systems which typically include unique hardware devices that require simple device drivers. If this facility is abstracted appropriately, it is also an aid to portability. This facility—dubbed *interrupt channeling*—is used by existing drivers for the keyboard, serial port, and timer. It is also used to connect handlers to hardware traps such as divide by zero.

A related facility, already alluded to, is the ability to unblock a thread from within an interrupt routine. Interrupts are handled on the stack of whatever thread happens to be running when the interrupt arrives. During processing of the interrupt (which may be entirely unrelated to the thread currently running), a previously blocked thread may need to be unblocked. The thread state is marked ready to run, and then the thread priority is compared with the priority of the current thread. If the priority is less urgent than the current thread, nothing more needs to be done. If the priority is more urgent than the current thread, a flag is set so that when the return from interrupt is about to happen, a context switch to the new thread takes place.

The case also should be mentioned where there is no ready thread and an interrupt occurs. In this case, a tight loop is being run in the context of whatever thread last blocked itself. This loop is watching to see if that thread again becomes ready. If this thread does get marked ready, the loop will terminate and the thread will return from the blocking function. If some other thread gets marked ready, a switch must be made immediately to run that thread in the manner described above.

2.8 Memory allocation

The Skidoo kernel has an extremely simple memory allocation scheme. A table of available memory regions is maintained, and blocks are allocated from the first region that contains sufficient space. A call to free a memory block is provided, but it is ignored at present in this simple allocator. This scheme is entirely adequate for most embedded applications that will allocate all resources that are required at boot time. Facilities that may want to reuse allocated objects (such as semaphores) should keep them on a private free list.

As an alternative to this simple scheme, the Solaris slab allocator [22, p. 392] has been made to work with the Skidoo kernel. However, the slab allocator requires nearly as much code memory as Skidoo itself (see Figure 3.3).

2.9 Device drivers

Drivers for the keyboard and console were essential to develop and debug the kernel. The serial port driver was as much a demonstration as an essential part of the kernel. In retrospect, the serial port could have been used in lieu of the console for development. The console driver has the advantage of being independent of interrupts, and thus is useful in cases where the serial driver would have failed. The keyboard driver typically uses interrupts, but can be configured to work without them.

2.10 Boot services

Bootstrapping is a low–level hardware–dependent issue. This is particularly so on the x86, where the machine starts running in a backward compatibility mode (x86 "real mode"). The goal of bootstrapping is to be able to run C code in x86 protected mode.

At this time, Skidoo is able to boot from floppy disk, as well as over the network using BOOTP and TFTP . It would be fairly straightforward to allow booting from a hard drive or CDROM.

Once Skidoo has been initialized, it creates one initial thread at priority 0, which executes the C function `user_init()`. Typically this function will launch the set of user threads and then exit.

CHAPTER 3

IMPLEMENTATION

This chapter describes the implementation of Skidoo. The description is given as a step by step chronology. As each major subsystem is presented the algorithms used to implement it are discussed. In almost every case the simplest possible algorithm has been chosen. Correctness has been placed ahead of efficiency, yet with the expectation that more complex and sophisticated algorithms will be introduced as the project matures. In some cases (notably memory allocation) this has already occurred.

3.1 Hardware Platform

The decision to make Skidoo run on the Intel x86 [10] processor was an easy one. X86 hardware is cheap, ubiquitous, and amazingly effective. This choice made it straightforward to use the Gnu C compiler hosted on a Linux system for development. Other potential targets were the Sparc and the Motorola 680x0 processors. Although these architectures are in many ways more attractive, the hardware is less common, more expensive, and slower. A desktop personal computer was obtained for use as a target machine. The machine used had a 200 Mhz Pentium MMX processor, 64 MB of memory, video card, and network card.

3.2 Diskless Booting

Part of the challenge of the project was that Skidoo had to boot and run on bare hardware. It was expected (and rightly so) that development would involve many cycles of experimental rebooting and that a scheme that made this as efficient as possible would be the best choice. It is possible to write images onto floppy disks on the development machine and boot them by transferring them to the target machine. However, this rapidly becomes tedious, and floppy disks are remarkably unreliable. This was, however, a useful method when a pair of laptop computers were being used during a mobile development session.

Network booting is far superior to using removable media. A new image can be built on the development machine, and when the target machine reboots, this image is transferred into target memory and executed. Network booting uses the BOOTP and TFTP protocols and is facilitated by a public domain package called `netboot` [11]. Some time was lost discovering that some features of this package were broken or mis–documented. In particular, `netboot` exhibited aberrant behavior for an image containing 512 or fewer bytes. Ultimately `netboot` successfully loaded and ran a small assembly language program that used BIOS[1] routines to print a short message.

3.3 Processor Initialization

After reset, the Intel x86 processor is running in "real mode," which is a compatibility mode that runs software written for the oldest members of the x86 processor family. The `netboot` package expects the processor to be running in "real mode," as does the BIOS software. The Gnu C compiler generates code for an x86 processor running in protected mode. Protected mode supports 32 bit registers and a simple linear address space. The purposes of processor initialization are to relocate the Skidoo image into low memory and to perform the transition from real to protected mode. Once this is done, the processor can run code generated by the Gnu C compiler.

Debugging the processor initialization code was difficult and frustrating. Most of the processor initialization code had to be written in assembly language. In addition, it was not possible to use the real mode BIOS console routines for debugging while the initialization was in progress. A useful debugging tool was a cable with a single LED[2] that was connected to the parallel printer port. A simple routine that looped while blinking this LED was useful as a sentinel to mark progress through the initialization code. Once the processor was properly initialized, a newly coded LED loop, written in C and compiled by the Gnu C compiler, was successful in making the LED blink.

3.4 Console Output

Once it was possible to write code in C and run it in protected mode, progress became much faster. The next thing to do was to print messages on the console. Because the PC architecture supports simple memory mapped console output, it was easy and actually fairly enjoyable to write routines to output messages. No interrupts were involved and there were no complicated timing or synchronization issues.

Once the console output was working, and a simple `printf()` function was available, a keyboard driver became almost essential. Although the keyboard can generate interrupts, the driver was initially written to use polling loops to monitor the keyboard status register. Once both keyboard and console were working, some diagnostic routines were written, including routines to display regions of memory.

An alternative to developing drivers for the console and keyboard would have been to develop a driver for the serial port. This would have had some advantages in that a serial connection via a cable to the development system could have been used for debugging. Having such a facility at the earliest stages of development would

[1] The BIOS refers to the software in read–only memory. On a typical personal computer it contains bootstrap software, along with a rudimentary set of device drivers.

[2] LED: light emitting diode

have been very useful and this should be considered if Skidoo is ported to other architectures.

3.5 Threads

The most important aspect of Skidoo was the ability to do concurrent programming using threads along with semaphores for synchronization. In Skidoo, a thread consists of a control structure and a stack. The control structure contains a pointer to the stack, space to store saved registers, a small amount of status information, and a pointer to the function where that thread should start executing. A thread state variable indicates whether the thread is ready to run or blocked for some reason.

All threads are kept on a single linked list. Initially there is only a single thread which starts in the function `user_init()`. Switching between threads is accomplished by saving registers, switching stacks, and restoring registers so that execution resumes in a different thread. Although it is obscurely documented, the Gnu C compiler expects 6 registers to be preserved between function invocations (`ebx`, `edx`, `esp`, `ebp`, `esi`, and `edi`), and 2 registers (`eax` and `ecx`) may be freely destroyed. Assembly language code was written to perform context switching.

The pair of routines `thr_block()` and `thr_unblock()` are the heart of the thread system. A thread calls `thr_block` to mark itself not ready; another thread is then selected to be run. Calling `thr_unblock` allows a blocked thread to be marked ready once again. At this early stage of the system there were no interrupts (in particular no clock interrupt) and no preemptive scheduling. When a thread performed a blocking call, it would save its registers and switch to an idle thread, which ran a scheduler. The scheduler would search the thread list for some other thread to run. If there was no ready thread, the system could do nothing but halt.

It turns out that there is no reason to have a separate idle thread. (In fact having one introduces a needless thread switch.) This code was revised so that each thread runs the scheduling loop when it needs to find another thread to pass control to. A call `thr_yield()` was needed in the early system; it ran the scheduling loop, even when a thread did not wish to block. It has been eliminated in the final system, but it was necessary before preemptive scheduling and priorities were operational. Without it, new threads were created but never ran.

3.6 Priorities

A policy was needed to decide which thread to run in the event that more than one thread was in the ready state. By assigning each thread a priority and insisting that all priorities be unique, it is easy to define a simple, unambiguous policy: the thread in the ready state with the most urgent priority should run. It is certainly possible to define other policies, especially if there are multiple threads with identical priority.

Adding priorities involved adding the priority to the thread control structure and adding policy code to the scheduling loop. At first, the entire linked list of threads was searched and the most urgent ready thread selected and run. It soon became obvious that a good optimization is to keep the list in order, with the most urgent threads first, and search the list only from the current thread to the end. Currently, a thread is created with a specified priority and that priority remains fixed. It would be easy to allow a thread to change priority (or to have its priority changed). Once this is done, it will be necessary to consider the need to immediately schedule a thread whose priority has been elevated.

3.7 Semaphores

The blocking and unblocking calls provide a clumsy form of synchronization. The utility of semaphores is well known [1, 5]. Given the thread blocking and unblocking facility, they are simple to implement using a state variable and a list of blocked threads. Skidoo implements binary semaphores with an initial value of one for mutual exclusion. Binary semaphores with an initial value of zero are used for private signaling semaphores.

3.8 Timers and Interrupts

The ability to perform accurate time delays is essential to embedded real time programming. The desire for timing signals was the impetus to provide facilities in Skidoo for dealing with interrupts. A substantial amount of assembly language code was written to save registers and call a specified C language interrupt handler. A function was provided to allow an arbitrary interrupt source to be routed to a specified C function and this facility was used to implement a handler for timer interrupts. The default timer handler keeps time by counting ticks, performs a callback to a user timer function (if one has been registered), and handles a list of threads with pending delays. This list is kept in order of increasing delay with the shortest delay at the front of this list. As entries are added, they are placed in proper order and the delay count adjusted so that it is relative to the entry preceding it. Once this is done, only the front entry needs to be decremented at each tick [13].

3.9 Preemption

When an interrupt occurs, some thread is suspended while the interrupt handler runs. Normally, when the handler is finished, the same thread is again resumed. Once an interrupt had the potential to modify the state of a thread, it was necessary to consider the possibility of resuming a different thread upon completion of the interrupt. In particular, it is desirable to resume a different thread when that thread is more urgent than the currently running thread, and has been marked ready by the interrupt handler. As a specific example, expiration of a delay interval could cause a new thread to be marked ready within an interrupt routine. If this thread was of more urgent priority than the current thread, it would be necessary to resume it immediately rather than resume the thread that was running when the interrupt occurred. In this case, the currently running thread is marked as "suspended during interrupt" and the more urgent thread is resumed instead when the interrupt is finished. This results in there being two possible ways that a thread can be suspended. One is the synchronous case where the thread itself performed a call to `thr_block`. The second is the asynchronous case where the thread was left suspended after an interrupt routine handed control to some other thread. In retrospect it would be possible to make these two states identical by simply saving some additional registers in the synchronous case, but this has not been done.

The situations that may result in a change in the currently running thread are enumerated in Figure 3.1. Notice that `thr_yield()` is not listed, as it has been eliminated in the final system.

Thread is created.
Thread exits.
Thread blocks.
Thread is unblocked.
Thread changes priority.

FIGURE 3.1. Thread preemption points

The ability to unblock threads and transfer to them immediately from interrupt code is vital in real–time systems. In Figure 3.1 the blocking and unblocking points include semaphores, timer delays, and other synchronization primitives yet to be described. Whenever a thread enters or leaves the system (or is blocked or unblocked), it is necessary to reevaluate which thread should be running. The ability for a thread to change priority was originally not part of the design, but it is necessary to provide correct behavior in certain situations. Notable among these is *priority inversion*, where a less urgent thread holds a mutex that a more urgent thread is blocked waiting to acquire.

3.10 Continuations

Normally a thread calls `thr_block()` to block, and returns again from this function when it unblocks. Continuations [6] were implemented as an experimental alternative. A thread which blocks with a continuation specifies the function it should execute when it unblocks. It never returns from the blocking call but instead executes the specified function. This makes it possible to save only the function pointer and an argument rather than the full set of registers. Continuations were easy to implement by adding additional thread state to support a new resumption mechanism. It turns out that this mechanism is entirely appropriate for launching a new thread: specify a continuation function and add the new thread to the thread list.

Continuations suggest an additional optimization that has not yet been exploited. When the scheduling loop finds no thread ready, and the current thread expects to resume with a continuation, the system could set a flag indicating that no state needs to be saved when departing from this thread. It would remain in this state until an interrupt occurs, and no registers would need to be saved to transition from this state into the interrupt routine and from there to whatever thread should be resumed. It has been pointed out [5] that most operating systems have a "homing position" where the system resides when it is at rest. Threads "accept a task" and leave the homing position, returning again when the task has been performed. By designing a system to consist of threads that each handle a specific task using a continuation, very fast response times could be achieved.

3.11 Condition Variables

Once a clean interrupt system was developed and tested, the keyboard driver was redesigned to use interrupts. An interrupt–driven keyboard could support "hot keys" that would display debug information, and even reboot the system in cases where the system had hung. (This did in fact prove immensely useful.) The keyboard interrupt handler would, with interrupts blocked, add characters to a queue. It is necessary both to lock the queue during access by a thread consuming the characters, and to signal such a thread once it had blocked upon finding the queue empty. Locking was done by disabling interrupts, but the business of blocking while releasing the lock needed to be handled carefully to avoid race conditions. This is exactly the sort of problem that monitors and condition variables were intended to solve [1, 9].

A condition variable facility was developed to couple together a mutex lock with a signaling semaphore. Two forms of condition variables are provided. The first couples together a mutex semaphore with a signaling semaphore and is intended for signaling between threads. The second couples together the interrupt lock with a signaling semaphore and is intended for signaling between threads and interrupt routines. The latter form has been used in both the keyboard and serial drivers.

3.12 Drivers

A driver for the serial ports was developed next. It was gratifying that Skidoo was mature enough that this driver could be developed in a straightforward way, entirely in C, using existing facilities. Interrupt routines were connected to hardware using interrupt channeling. Condition variables were used for signaling between interrupt routines and waiting threads—both to indicate available characters on reception and additional buffer space on transmission. The character queue library designed for the keyboard driver was used for both transmitted and received characters in the serial port driver. Although this driver did not require new mechanisms in Skidoo, it did give the system a good workout. With the serial port active, interrupts from the keyboard, timer, and serial port were all active and some new bugs were exposed and fixed.

A driver for an ISA–bus data acquisition card was also developed (the "DAS-16" marketed by Keithley–Metrabyte and others). The driver uses interrupts, was coded entirely in C, and required no new methodology in Skidoo. Writing a driver like this is an excellent test, since every embedded system seems to involve developing drivers for new hardware, and a good test of the system is how easy it makes this process. Having immediate access to the address space containing the hardware and not needing to install assembly language interrupt routines made the process easy.

description	lines
header files	713
assembly language	1307
console.c	1108
delay.c	166
main.c	107
prf.c	644
random.c	112
serial.c	523
sklib.c	390
thread.c	1550
trap.c	958
version.c	7
das16.c	407
user.c	140
tests.c	2096
server.c	125
total	10353

FIGURE 3.2. Source code summary

3.13 Code size

The source code for Skidoo is freely available from `http://kofa.mmto.org/skidoo`. (This thesis describes version 0.4.1 of Skidoo.) Skidoo consists of about 10000 lines of C source code, as shown in Figure 3.2. Considering that about 2000 lines of this are test cases, the core of Skidoo is approximately 8000 lines of code. About 1300 lines of this are assembly language code. The processor initialization, thread switching, and basic interrupt handling could only be expressed in assembly language. On a processor other than the x86, the processor initialization code would be much smaller, since other processors do not have multiple modes like the x86. Figure 3.2 does not include the slab allocator or any of the network code.

Figure 3.3 shows the compiled size of the Skidoo kernel in various configurations. The basic kernel is compact enough to fit into read–only memory as part of a compact application.

basic kernel	22k
kernel and tests	36k
kernel and slab allocator	55k
kernel and network	320k

FIGURE 3.3. Executable code size

Chapter 4
APPLICATIONS AND TESTS

This chapter discusses what has been done to test and exercise Skidoo. An extensive test suite was developed case by case as features were added to Skidoo. An interactive "shell" was developed to invoke the tests and to inspect the internal state of Skidoo. Two applications – a serial terminal and a data acquisition facility – were developed to demonstrate the utility of Skidoo.

4.1 Test suite

Each time a new feature was added to Skidoo, a test case was written to exercise that feature. All of the test cases have been retained, even old, seemingly historical and trivial ones. This test suite has grown to over 2000 lines of code. Assembling this set has proved to be a prudent course of action. After the specific test case for a new feature has run successfully, the entire suite of previous cases is run. Once that is successful, the entire suite is placed in a loop and run multiple times, sometimes through the night. Both of these last procedures have turned up unexpected bugs. Figure 4.1 shows the set of tests.

1	Start a thread
2	Setjmp and longjmp
3	Timer hookup
4	Thread delay
5	Create multiple threads
6	Create reentrant threads
7	Signal with semaphores
8	Pass arguments to thread functions
9	Many threads block on single semaphore
10	Unblock semaphore from an interrupt routine
11	Signal thread using condition variable
12	Thread join and exit
13	Mutex semaphore
14	Keyboard diagnostic
15	Serial port diagnostics

Figure 4.1. Test suite

4.2 Debugging tools

In addition to the test suite, a set of diagnostic tools were developed to exercise Skidoo. A simple command line interface allows individual tests to be run once or any number of times. Additionally the entire suite may be run as many times as desired. Retaining every test, and repeating them as new features were added has proved invaluable. Often the addition of a new feature introduces bugs in an old one. When changes are made to basic algorithms, it is essential to be able to verify that all features still work properly.

In addition to running the tests, the "shell" allows dumping of memory, inspecting the stack, and inspecting important data structures. An example of the display of the thread list is shown in Figure 4.2.

This display shows a typical thread status during testing. The asterisk next to the thread at priority 55 indicates the currently running thread. The state column shows that the two threads with more urgent priorities are blocked on a semaphore and a timer delay. A single character shows the resumption mode of each thread: `J` for the usual "jump" mode, `C` for a continuation, `I` for a postponed interrupt.

```
  Thread:  name (   &tp    )  state     esp     pri
  Thread: sserv (0000C4EC)     SEM J 00072E8C 50
  Thread:   tf2 (0000C3F4)   DELAY C 00074FF4 52
* Thread:   tf1 (0000C470)   READY I 00073FF4 55
  Thread:  user (0000C568)   READY J 00071F80 899
  Thread:   sys (0000C5E4)   READY J 00070FAC 950
  Thread  Cur : (0000C470)  (INT)
```

FIGURE 4.2. Typical thread display

4.3 Timing

A number of experiments were performed to measure the time necessary to respond to interrupts, and to transfer control to a previously blocked thread. In all cases the timer was used to generate interrupts to supply the triggering event. The Pentium timestamp counter was used to measure intervals.[1] The system is running a thread at a non-urgent priority that is looping reading the timestamp counter and writing

[1] The Pentium timestamp counter is a constantly incrementing 64 bit timer that counts at the system clock rate, in this case 200 Mhz. It can be read using a special processor instruction into a pair of 32 bit registers.

it into a memory location. When the timer interrupt happens, the last timestamp value remains in the designated memory location. Once the more urgent thread is activated, it reads the timestamp counter again and can calculate the transfer time. These experiments were performed on a processor running at 200 Mhz so that resolution of 5 nanoseconds was obtained. Figure 4.3 shows the results obtained.

experiment	latency
thread to thread, normal	2.32 microseconds
thread to thread, continuation	2.05 microseconds
thread to interrupt	0.855 microseconds

FIGURE 4.3. Control transfer timings

4.4 Serial terminal

Given the console, keyboard, and serial port driver, a very simple application is a serial terminal. A pair of threads are created. One blocks waiting for keyboard input; the other blocks waiting for serial port input. When characters arrive from the serial port, they are output to the console. When they arrive from the keyboard, they are output to the serial port.

Even though this is a trivial application, it serves as a good diagnostic for multiple interrupt sources (keyboard, timer, and serial port). This test has no real–time requirements apart from the need to move input characters before the small (128 byte) input buffers overflow.

The code for this application is shown in Figure 4.4. The function `user_init` runs as a high priority thread, initializes serial port parameters, starts two new threads, and is done. The first thread runs in the function `t_in` and is usually blocked waiting for characters to arrive on the serial port, which it then copies to the console. The second thread runs in the function `t_out` and is usually blocked waiting for characters to be typed at the keyboard. When characters are typed, they are copied to the serial port. Although both threads run at the same priority, neither is CPU bound and will yield the processor to the other when it blocks.

```
void
user_init ( void ) /* initialize port & create 2 threads */
{
        sio_baud  ( PORT, 9600 );
        sio_crmod ( PORT, 0 );

        (void) thr_new ( "te_i",  t_in, (void *)PORT, PRI_TERM, 0 );
        (void) thr_new ( "te_o", t_out, (void *)PORT, PRI_TERM, 0 );
}

static void
t_in ( int port ) /* copy from serial port to console */
{
        int c;

        for ( ;; ) {
            c = sio_getc ( port );
            if ( c == '\r' )
                    c = '\n';
            vga_putc ( c );
        }
}

static void
t_out ( int port ) /* copy from keyboard to serial port */
{
        int c;

        for ( ;; ) {
            c = kb_read ();
            sio_putc ( port, c );
            if ( c == '\r' )
                sio_putc ( port, '\n' );
        }
}
```

FIGURE 4.4. Serial terminal application

4.5 Data Acquisition Server

A data acquisition server was written as a demonstration application. This server allows an analog data acquisition device to be controlled remotely using a serial port. Single samples, periodic data, and burst data may be collected from any of 16 inputs.

This application makes use of a special piece of computer hardware. A "DAS-16" analog data acquisition board was installed in the target system and a device driver written to access this device under Skidoo. The "DAS-16" board is typical of the sort of hardware used in embedded control projects. It has a 12–bit analog to digital converter and an input multiplexer that selects one of 16 channels for conversion. It also has a pair of 12–bit digital to analog converters and a programmable timer. Analog to digital conversions may be triggered by software or by the timer. When a conversion is complete, an interrupt is generated.

Most of the work developing this application consisted of writing the device driver for the "DAS-16." The driver for this device consists of about 400 lines of C, and was straightforward to develop. Once the driver was written, a simple protocol was designed to make the drivers capabilities accessible from the serial port.

The server provides access to the hardware in three different modes. A single sample may be obtained from any channel at any time. A burst captures a precisely timed sequence of 1000 points at 1000 Hz. Periodic sampling may be scheduled at a more leisurely rate. The server uses two threads. One thread waits for commands from the serial port and handles them as they arrive. The second thread handles periodic sampling and waits for timer events. A mutual exclusion semaphore is used to arbitrate access to the conversion hardware by the two threads. The lock is held for the duration of a burst (an entire second). A sample of the code to support the burst mode is shown in Figure 4.5.

The interrupt routine that supports the burst mode follows in Figure 4.6. It accumulates the required number of samples into a buffer, then unblocks the waiting thread using a semaphore. The thread that supports periodic sampling is activated by passing a semaphore from a clock interrupt routine, as shown in Figure 4.7.

This is a simple application, but it illustrates many of the features of Skidoo. This kind of application is used for monitoring temperatures and forces in a remote location with telemetry being obtained over a serial connection, perhaps using optical fibers. The set of facilities provided by Skidoo are adequate to allow it to replace commercial operating systems in many applications now in service at large telescopes in Southern Arizona.

```
/* create semaphores */
das_sem = sem_signal_new ( SEM_FIFO );
das_mutex = sem_mutex_new ( SEM_PRIO );

short *
das_burst ( int chan, int num )
{
        /* enter critical region to access hardware */
        sem_block ( das_mutex );

        das_scan ( chan, chan );
        das_rate ( 100, 100 );
        outb ( CTL_IRQ_5, base + CTL );
        outb ( CTL_IE | CTL_IRQ_5 | CTL_TT, base + CTL );

        /* connect handler to interrupt */
        irq_5_hookup ( das_int );

        /* clear the interrupt flag and enable the clock.  */
        outb ( 0x00, base + STATUS );
        outb ( CLK_GATE, base + CLOCK );

        count = 0;
        want = num;
        next = buffer;
        state = RUN;

        /* wait for signal that data has accumulated */
        sem_block ( das_sem );

        /* exit critical region */
        sem_unblock ( das_mutex );
        return buffer;
}
```

FIGURE 4.5. Example: routine implementing burst mode

```
static void
das_int ( void )
{
        int scan, data;
        unsigned long ticks;

        /* clear interrupt request.  */
        outb ( 0xff, base + STATUS );

        /* let this request be a no-op */
        if ( state != RUN )
            return;

        /* read sampled value from hardware */
        scan = inb ( base + DATA_LO );
        data = inb ( base + DATA_HI ) << 4;
        *next++ = data | (scan & 0xf0) >> 4;

        /* signal when requisite number is obtained */
        if ( ++count >= want ) {
            state = HOLD;
            sem_unblock ( das_sem );
        }
}
```

FIGURE 4.6. Example: interrupt routine

```
static short per_buf[240];
static short *next_per = per_buf;
static int per_count = 0;
static struct sem *per_sem;

void my_timer ( void )
{
        /* subdivide clock, activate thread every 10 seconds */
        if ( (per_count++ % 1000) == 0 )
            sem_unblock ( per_sem );
}

static void
periodic ( int xx )
{
        for ( ;; ) {
            /* wait for timer activation */
            sem_block ( per_sem );
            /* place new reading in buffer */
            if ( next_per < &per_buf[240] ) {
               *next_per++ = das_adc ( 3 );
            }
        }
}

void
user_init ( int xx )
{
        /* new signaling semaphore */
        per_sem = sem_signal_new ( SEM_FIFO );

        /* initialize timer and connect timer handler */
        tmr_rate_set ( 100 );
        tmr_hookup ( my_timer );

        /* launch new thread to be activated by timer */
        (void) thr_new ( "das", periodic, (void *) 0, 49, 0 );
}
```

FIGURE 4.7. Example: timer activated thread

Chapter 5

CONCLUSIONS AND FUTURE WORK

This thesis has described the design and implementation of Skidoo, a compact real–time operating system. Skidoo offers a complete set of threading services at the kernel level, along with a versatile set of synchronization primitives. The premise that a useful system could be built simply from threads and semaphores has been demonstrated to be valid. Additional services such as timers, interrupt channeling, and device drivers have been implemented to enrich the facilities provided.

Skidoo incorporates a number of features that are unusual and interesting, if not new. A clean high level facility for interrupt handling makes the system especially convenient to work with. The availability of continuations offers a new and useful way to control threads.

There is no question that this project has been a worthwhile learning exercise. Indeed it has gone significantly beyond that to become a useful tool for myself, and hopefully for others. It has already been used to support a number of small projects. However, more can be done to provide library routines and a more complete set of device drivers. The following sections outline some major extensions of functionality that are sensible next steps to take with this project.

5.1 Networking

Without question, providing a network stack would be the single thing that would add the most utility to Skidoo. This work has been started and development is ongoing.

The goal of adding a TCP/IP network stack is to have TCP and UDP sockets available within Skidoo. Rather than do this work from scratch, the plan is to incorporate the network code from an existing open source system. BSD 4.4 and Linux are both reasonable candidates, but Linux has been chosen because it supports the greatest diversity of hardware. In order to use Linux source code with little or no change, it will be necessary to construct a limited Linux emulator within Skidoo. The advantage of keeping the Linux code pristine is that it should be easier to migrate to newer versions of the Linux code as they become available.

The network facility (and Linux emulation layer) would be optional modules that could be omitted from Skidoo to reduce the memory requirements for those applications where they are not required.

5.2 Memory protection

Skidoo makes no use of the memory protection hardware that is available on the x86 processor. It would be possible to make some beneficial uses of the memory protection hardware without changing the single shared address space that is a central feature of Skidoo.

The memory pages containing the Skidoo kernel could be marked as read/execute or execute only, so that processor traps would occur if application threads made invalid references to that part of the address space. This could be a great benefit to debugging of new code and should make the system more robust.

The pages containing the stack for each thread could be mapped into fixed virtual addresses for all threads, and a page at the end of the stack could be set up as a "red zone" to catch stack overflows. This would also be a significant aid to debugging.

5.3 Debug facilities

Skidoo was developed with almost no planning and forethought given to debugging. The process would almost certainly have been more efficient if some sort of debug facility had been built in as early as possible. It is possible to run the Gnu debugger in remote mode across a serial link, and this would have been a great help. It would be a worthwhile facility to incorporate if Skidoo is developed further, and particularly if it is ported to new architectures.

5.4 Incremental module loading

At present, to add code to Skidoo, new modules must be compiled and linked with the Skidoo core. The resulting image is then loaded by rebooting the system.

Once a network facility is available, it would be very attractive to have a feature whereby modules could be incrementally loaded into a running system. Primarily this would be of benefit for code development, because many iterations of testing could be done without the necessity of rebooting the system. This would also make it possible to boot the Skidoo core from a read-only medium such as CDROM, and then incrementally load modules to obtain customized behavior or to facilitate development and testing.

To do incremental module loading would require maintaining a symbol table to perform lookups of already loaded symbols. Existing object file formats provide relocation information that could be used to modify address references as a module was loaded. The software to do the relocation would have to be written. An attractive option would be to do the symbol table management and relocation outside of Skidoo as part of a more sophisticated development system than ran on the development host. This would be especially attractive for a target host with minimal memory.

5.5 Porting to new hardware

It would be worthwhile to port Skidoo to non–x86 architectures as well as to multiprocessor machines (whether x86 or some other architecture). Many projects exist using older hardware such as Multibus and VMEbus computers. These projects have a substantial investment in hardware other than the processor itself and most commonly use non–x86 processors such as the Motorola 680x0 or the Sparc.

Equally interesting would be the task of making Skidoo run on one of the increasingly common and inexpensive x86 SMP machines. The change to the scheduling policy in this case is straightforward: at any time, run the two runnable threads with most urgent priorities! The fact that Skidoo is already fully preemptable will be a tremendous aid to making it run on a multiprocessor.

5.6 Other suggestions

A message passing facility would be a useful addition for both synchronization and communication. This would be essentially a data carrying semaphore. An example of where this would be useful would be an I/O operation that blocks waiting for data, but also is set up to unblock on a timeout. A message would provide a convenient way to indicate which event unblocked the thread.

An interactive shell would also be a valuable addition to Skidoo. A very useful mode of testing has been to exploit the serial driver in Skidoo and to set up a simple RPC facility across the serial link. This would be substantially more useful when network sockets are available. As it is, it is incredibly productive to manipulate Skidoo using an interpreted language such as Perl or Ruby. Since an interactive shell would be a tool for development and debugging, it makes a lot of sense to let it run on the development host and to communicate with Skidoo using an RPC stub. In any event, it should be arranged as an optional module so that both it and networking could be omitted to produce a more compact image if desired.

Appendix A

KERNEL INTERFACE SPECIFICATION

A.1 Hardware requirements

Skidoo runs on the x86 processor. Development was done using a 200 MHz Intel Pentium-MMX with 64 MB of memory. Skidoo has been tested on processors ranging from the 486 through the Pentium-III. It should run on the 386 processor as well, but this has not been tested. Certain processor enhancements such as the 64 bit time stamp counter will not be available on the 386, but this is not essential to run the kernel.

The most convenient way to run Skidoo is to use the `netboot` package and arrange for diskless booting from a server. If this is done, the only hardware required besides the processor, motherboard and memory are a video and network card. If `netboot` is not present in the ROM on the network card, a floppy disk or CDROM will be required for booting. If the motherboard supports it as a boot device, the CDROM is the most reliable and convenient.

`thr_new`	Create a new thread.
`thr_exit`	Terminate current thread.
`thr_self`	Identify current thread.
`thr_kill`	Terminate some thread.
`thr_join`	Await thread termination.
`thr_block`	Block current thread.
`thr_block_c`	Block current thread with continuation.
`thr_block_q`	Block current thread, reusing continuation.
`thr_unblock`	Unblock some thread.

Table A.1. Thread calls

A.2 Threads

```
struct thread * thr_new ( char *name, tfptr func, void *arg,
int prio, int flags )
```

This is used to create a thread. A string may be given to identify the thread on status listings. A function is specified, to which a single argument may be passed. (If it is desired to pass multiple values to a thread, they should be loaded into a structure, and a pointer to the structure should be passed to the thread function.) The thread is assigned a priority, and flags are provided to specify unusual behavior. The present collection of flags are `TF_FPU` (the thread uses floating point) and `TF_BLOCK` (the thread should start up blocked). Most threads do not use floating point, and floating point registers are not saved and restored during a context switch unless the appropriate flag is specified.

Priorities are stored as 32 bit signed integers; larger positive numerical values are less urgent. Behavior for negative priorities is not defined and these values should be avoided. Each thread must be created with a unique priority so that there is no ambiguity about which thread should run at any given time.

```
void thr_exit ( void )
```

This may be used by a thread to destroy itself. If a thread just "falls off the end" by returning from the thread function, a call to `thr_exit` is made transparently. In the majority of cases, threads either run forever or fall off the end, so this function is rarely used directly.

```
struct thread * thr_self ( void )
```

This allows a thread to get a pointer to itself. This can be more convenient than saving the pointer returned by `thr_new`.

```
void thr_kill ( struct thread * )
```

This terminates a running thread. If called on the current thread, it is the same as calling `thr_exit`. If called on another thread, it arranges for it to resume in `thr_exit` and marks it ready to run. The next time it is scheduled, it will exit.

```
void thr_join ( struct thread * )
```

This blocks the current thread until the specified thread calls `thr_exit`.

```
void thr_block ( enum thread_state why )
void thr_unblock ( struct thread * )
```

These functions are the most fundamental synchronization primitives in the Skidoo kernel. Calling `thr_block` blocks the current thread and posts a state other than `READY` to indicate why. Calling `thr_unblock` unblocks the specified thread. They are rarely accessed directly; semaphores or condition variables are used instead.

```
void thr_block_c ( enum thread_state why, tfptr func, void *arg )
```

This is identical to `thr_block` except that a continuation is specified. When the thread is unblocked it will resume in the continuation function, whereas with `thr_block` the thread resumes by returning from the `thr_block` call.

```
void thr_block_q ( enum thread_state why )
```

This is a common optimization after a call to `thr_block_c` has been previously made. It should be noted that at all times every thread has a continuation function set. If one has never been set explicitly, it is implicitly the function specified to `thr_new` when the thread was created. A call to `thr_block_q` blocks the thread and sets a flag so that it will resume in whatever continuation function has already been specified. It is a slight optimization over calling `thr_block_c` repeatedly with the same continuation function.

`sem_mutex_new`	Create a new mutex semaphore.
`sem_signal_new`	Create a signaling semaphore.
`sem_destroy`	Destroy a semaphore.
`sem_block`	Block on a semaphore.
`sem_unblock`	Unblock a semaphore.
`sem_block_try`	Test and block on a semaphore.
`sem_block_c`	Block on a semaphore with continuation.
`sem_block_q`	Block on a semaphore, reusing continuation.

TABLE A.2. Semaphore calls

A.3 Semaphores

```
struct sem * sem_mutex_new ( int flags )
struct sem * sem_signal_new ( int flags )
```

Calling `sem_mutex_new` creates a new mutual exclusion semaphore. A call to `sem_signal_new` creates a new signaling semaphore. The flags variable may be used to indicate alternate scheduling policies (`SEM_FIFO` versus `SEM_PRIO`). The default is FIFO scheduling.

```
void sem_destroy ( struct sem * )
```

This call destroys a semaphore so that resources associated with it can be reused. This should only be done when there is no possibility of further activity on the semaphore.

`void sem_block ( struct sem * )`

This is effectively the `P` operation from the classical semaphore literature. If the semaphore is set (1), this call will clear it and keep executing. If the semaphore is clear (0), this call will block the current thread and place it on a list associated with the semaphore.

`void sem_unblock ( struct sem * )`

This is the `V` operation. This routine never blocks, but it may cause another thread to be unblocked. It does nothing if the semaphore is already set (1). If the semaphore is clear (0) and the semaphore queue is empty, the semaphore is set. If the semaphore is clear and the semaphore queue is non–empty, one thread in the queue is unblocked and the semaphore value remains cleared. Which queue entry gets unblocked depends on a policy flag set when the semaphore was created. In the usual case the policy is `SEM_FIFO`, and the entry at the front of the queue is unblocked. If the policy is `SEM_PRIO`, the entry with the most urgent priority is removed from the queue and unblocked.

`int sem_block_try ( struct sem * )`

This is a version of `sem_block` that attempts to acquire a semaphore (typically a mutex) but that will never block. If the semaphore is set, it clears the semaphore and returns 1. If the semaphore is already clear, it returns 0 rather than blocking as `sem_block` would do.

`void sem_block_c ( struct sem *sem, tfptr func, void *arg )`

This is identical to `sem_block` except that it resumes via a continuation.

`void sem_block_q ( struct sem *sem )`

This is identical to `sem_block` except that it resumes using a previously established continuation.

`cv_new`	Create a new condition variable.
`cv_destroy`	Destroy a condition variable.
`cv_wait`	Block and wait for a condition.
`cv_signal`	Signal a condition.
`cpu_enter`	Enter interrupt locked region.
`cpu_leave`	Leave interrupt locked region.
`cpu_new`	Create a new CPU condition variable.
`cpu_wait`	Wait for a condition under a CPU lock.
`cpu_signal`	Signal a condition under a CPU lock.

Table A.3. Condition variable calls

A.4 Condition Variables

Condition variables are a coupling of a mutex semaphore and a signaling semaphore. One mutex semaphore may be involved with several signaling semaphores, each expressing a different predicate. For this reason the mutex semaphore must be created first, and then coupled to each predicate in turn. Once this is done, the condition variable is a single unit that can be used in the wait call.

```
struct cv * cv_new ( struct sem *mutex )
```

This constructs a new condition variable that binds together the indicated mutex and a newly generated signaling semaphore.

```
void cv_destroy ( struct cv * )
```

This destroys a condition variable, and releases its resources.

```
void cv_wait ( struct cv * )
```

This blocks and waits for a signal on a condition variable. The caller must already hold the mutex semaphore.

```
void cv_signal ( struct cv * )
```

This unblocks a thread waiting on a condition variable.

```
void cpu_enter ( void )
void cpu_leave ( void )
```

This pair of routines obtain and release a cpu lock – by disabling and re–enabling all interrupts – in order to enter and leave an interrupt sensitive critical region. They provide an interrupt safe mutex.

```
struct sem *cpu_new ( void )
void cpu_wait ( struct sem * )
void cpu_signal ( struct sem * )
```

These routines are used in conjunction with `cpu_enter` and `cpu_leave` to implement a cpu locked condition variable. The routine `cpu_wait` is used to block and wait for a signal while holding a cpu lock. `cpu_signal` is used to unblock a thread waiting for the signal, typically from an interrupt handler where the cpu lock is implicitly held.

`tmr_rate_set`	Set clock interrupt rate.
`tmr_rate_get`	Get clock interrupt rate.
`tmr_hookup`	Connect a function to the timer.
`tmr_delay`	Block current thread for an interval.
`tmr_delay_c`	Block and delay using continuation.
`tmr_delay_q`	Block and delay reusing last continuation.

Table A.4. Timer calls

A.5 Timer facilities

A programmable hardware timer exists which produces interrupts at a 100 Hz rate. On the x86, the actual rate is 100.0067052 Hz. The timer is accessed by the following functions:

```
void tmr_rate_set ( int hz )
```

This sets the rate at which interrupts are produced by the timer. If this function is never called, timer interrupts are produced at 100 Hz.

```
int tmr_rate_get ( void )
```

This discovers the rate at which interrupts are produced by the timer.

```
void timer_hookup ( fptr func )
```

This specifies a C function that is called each time the timer interrupts. Only one callback of this sort is allowed; subsequent calls replace the previously established function. A null argument may be specified to disconnect the function.

```
void thr_delay ( int nticks )
```

This is a convenient (although somewhat imprecise) way to obtain timing delays. After this call the thread is blocked until the specified number of timer ticks have elapsed. At this time the thread will be made ready, and will run immediately if it is the runnable thread of most urgent priority. Otherwise, it will run only after more urgent runnable threads have blocked.

```
void thr_delay_c ( int nticks, tfptr func, void *arg )
```

This is identical to `thr_delay` except that it resumes in a continuation function.

```
void thr_delay_q ( int nticks )
```

This function delays, resuming in a previously established continuation function. Usually the continuation will have been specified in a `thr_delay_c` call, but the continuation given in `thr_new` could be used as well. The `thr_delay_q` function is convenient for constructing periodic loops using tail-recursion.

`vga_putc`	Put a character on the screen.
`vga_puts`	Put a string on the screen.
`vga_screen`	Select an alternate screen.
`getchar`	Await and read a keystroke.
`getchare`	Read keystroke and echo to screen.
`sio_getc`	Read character from serial port.
`sio_putc`	Send character to serial port.
`sio_gets`	Read string from serial port.
`sio_puts`	Send string to serial port.
`sio_baud`	Set serial port baud rate.

Table A.5. Device driver calls

A.6 Device drivers

This section summarizes the most important device driver access routines.

```
void vga_putc ( int c )
void vga_puts ( char *s )
void vga_screen ( int n )
```

The console driver outputs characters to a VGA compatible video card supporting a 25 line by 80 column console. A call to `vga_putc` places a single character on the screen. Calling `vga_puts` places all characters in a null terminated string on the screen. A call to `vga_screen` selects one of 8 virtual screens for display. Additional functions manipulate the cursor and are described in the source code.[1]

```
int getchar ( void )
int getchare ( void )
```

The keyboard driver reads from the standard PC keyboard. Calling `getchar` reads a character from the keyboard without attempting to echo the character. Calling `getchare` reads a character and echos it to the console, as would normally be expected.

```
int sio_getc ( int port )
void sio_putc ( int port, int c )
void sio_gets ( int port, char *s )
void sio_puts ( int port, char *s )
void sio_baud ( int port, int rate )
```

The serial driver reads from and writes to either serial port 0 or 1. Calls to `sio_getc` and `sio_putc` read and write a single character from the specified port. Calls to `sio_gets` and `sio_puts` read and write a string from the specified port. A call to `sio_baud` sets the baud rate on the specified port. Rates from 300 to 38400 are supported exactly. A rate of 56000 baud is only possible with a 3 percent error given the standard 1.8432 Mhz crystal, but this seems to work just fine.

A.7 Interrupt facilities

```
void vector_hookup ( int vector, fptr func )
```

This call arranges that the specified C function is called whenever the indicated interrupt occurs.

A.8 Booting and initialization

After bootstrap, the kernel relocates itself to the lowest part of memory. It then makes the rest of memory available for dynamic allocation. After all subsystems are initialized, the first thread is started in the function `user_init`. This is expected to be supplied by the user and will typically allocate resources and start other threads necessary to run the intended application. This first thread runs at the most urgent possible priority (priority 0), so that no other threads run until it exits.

[1] The source code may be obtained from `http://kofa.mmto.org/skidoo`. This thesis describes version 0.4.1 of Skidoo.

References

[1] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading, Massachusetts, 2000.

[2] Alexander Horstkotte Arnd C. Heursch and Helmut Rzehak. Preemption concepts, rhealstone benchmark and scheduler analysis of linux 2.4. In *Proceedings of the Real Time and Embedded Computing Conference*, Milan, November 2001.

[3] Maurice. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Upper Saddle River, New Jersey, 1986.

[4] Michael Barabanov. A linux–based real–time operating system. Master's thesis, New Mexico Institute of Mining and Technology, Socorro, New Mexico, 1997.

[5] Edsger W. Dijkstra. The structure of the THE –multiprogramming system. *Communications of the ACM*, 11(5):345–346, May 1968.

[6] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, 1991.

[7] Bill O. Gallmeister. *Programming for the Real World, POSIX.4*. O'Reilly and Associates, Cambridge, Massachusetts, 1995.

[8] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Second Symposium on Operating Systems Design and Implementation*, pages 123–136. USENIX, Seattle, October 1996.

[9] Richard. C. Holt. *Concurrent Euclid, the UNIX System, and Tunis*. Addison-Wesley, Reading, Massachusetts, 1983.

[10] Intel. *Intel386EX Embedded Microprocessor User's Manual*. Intel Corporation, Santa Clara, California, 1996. Order Number 272485-002.

[11] Gero Kuhlmann. http://netboot.sourceforge.net, 2002.

[12] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, April 1979.

[13] J. Lions. *A Commentary on the Unix Operating System*. Department of Computer Science, University of New South Wales, 1977.

[14] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, Massachusetts, 1996.

[15] Remi Nadeau. *Ghost Towns and Mining Camps of California*. The Ward Ritchie Press, Los Angeles, California, 1972.

[16] Gary Nutt. *Operating systems, A modern perspective*. Addison-Wesley, Reading, Massachusetts, 2000.

[17] Alessandro Rubini. *Linux Device Drivers*. O'Reilly and Associates, Sebastopol, California, 1998.

[18] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, Reading, Massachusetts, 1994.

[19] Wind River Systems. *VxWorks Programmers Guide*. Emeryville, California, 1989.

[20] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Upper Saddle River, New Jersey, 1992.

[21] Linus Torvalds. Linux: a portable operating system. Master's thesis, University of Helsinki, Finland, 1997.

[22] Uresh Vahalia. *UNIX Internals, the New Frontiers*. Prentice-Hall, Upper Saddle River, New Jersey, 1996.