Beyond Multiprocessing ... Multithreading the SunOS Kernel

J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams − SunSoft, Inc.

ABSTRACT

Preparing the SunOS/SVR4 kernel for today's challenges: symmetric multiprocessing, multi-threaded applications, real-time, and multimedia, led to the incorporation of several innovative techniques. In particular, the kernel was re-structured around threads. Threads are used for most asynchronous processing, including interrupts. The resulting kernel is fully preemptible and capable of real-time response. The combination provides a robust base for highly concurrent, responsive operation.

Introduction

When we started to investigate enhancements to the SunOS kernel to support multiprocessors, we realized that we wanted to go further than merely adding locks to the kernel and keeping the user process model unchanged. It was important for the kernel to be capable of a high degree of concurrency on tightly coupled symmetric multiprocessors, but it was also a goal to support more than one thread of control within a user process. These threads must be capable of executing system calls and handling page faults independently. On multiprocessor systems, these threads of control must be capable of running concurrently on different processors. [Powell 1991] described the user visible thread architecture.

We also wanted the kernel to be capable of bounded dispatch latency for real-time threads [Khanna 1992]. Real-time response requires absolute control over scheduling, requiring preemption at almost any point in the kernel, and elimination of unbounded priority inversions wherever possible.

The kernel itself is a very complex multi-threaded program. Threads can be used by user applications as a structuring technique to manage multiple asynchronous activities; the kernel benefits from a thread facility that is essentially the same.

The resulting SunOS 5.0 kernel, the central operating system component of Solaris 2.0, is fully preemptible, has real-time scheduling, symmetrically supports multiprocessors, and supports user-level multithreading. Several of the locking strategies used in this kernel were described in [Kleiman 1992]. In this paper we'll describe some of the implementation features that make this kernel unique.

Overview of the Kernel Architecture

A kernel thread is the fundamental entity that is scheduled and dispatched onto one of the CPUs of the system. A kernel thread is very lightweight, having only a small data structure and a stack. Switching between kernel threads does not require a change of virtual memory address space information, so it is relatively inexpensive. Kernel threads are fully preemptible and may be scheduled by any of the scheduling classes in the system, including the real-time (fixed priority) class. Since all other execution entities are built using kernel threads, they represent a fully preemptible, real-time ''nucleus'' within the kernel.

Kernel threads use synchronization primitives that support protocols for preventing priority inversion, so a thread's priority is determined by which activities it is impeding by holding locks as well as by the service it is performing [Khanna 1992].

SunOS uses kernel threads to provide asynchronous kernel activity, such as asynchronous writes to disk, servicing STREAMS queues, and callouts. This removes various diversions in the idle loop and trap code and replaces them with independently scheduled threads. Not only does this increase potential concurrency (these activities can be handled by other CPUs), but it also gives each asynchronous activity a priority so that it can be appropriately scheduled.

Even interrupts are handled by kernel threads. The kernel synchronizes with interrupt handlers via normal thread synchronization primitives. If an interrupt thread encounters a locked synchronization variable, it blocks and allows the critical section to clear.

A major feature of the new kernel is its support of multiple kernel-supported threads of control, called lightweight processes (LWPs), in any user process, sharing the address space of the process and other resources, such as open files. The kernel supports the execution of user LWPs by associating a kernel thread with each LWP, as shown in Figure 1. While all LWPs have a kernel thread, not all kernel threads have an LWP.

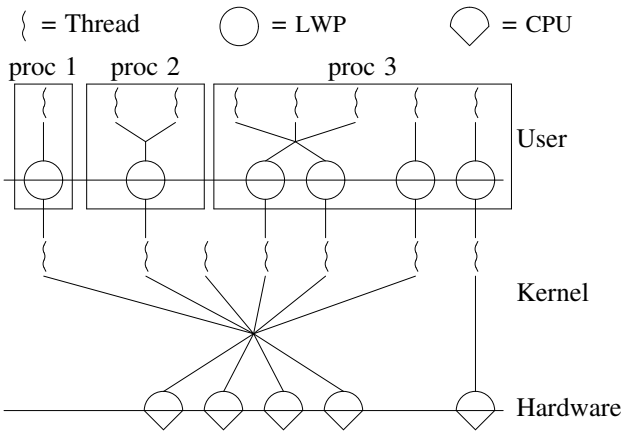

Figure 1: Multi-thread architecture examples

A user-level library uses LWPs to implement user-level threads [Stein 1992]. These threads are scheduled at user-level and switched by the library to any of the LWPs belonging to the process. User threads can also be bound to a particular LWP. Separating user-level threads from the LWP allows the user thread library to quickly switch between user threads without entering the kernel. In addition, it allows a user process to have thousands of threads, without overwhelming kernel resources.

Data Structures

In the traditional kernel, the `user` and `proc` structures contained all kernel data for the process. Processor data was held in global variables and data structures. The per-process data was divided between non-swappable data in the `proc` structure, and swappable data in the `user` structure. The kernel stack of the process, which is also swappable, was allocated with the `user` structure in the user area, usually one or two pages long.

The restructured kernel must separate this data into data associated with each LWP and its kernel thread, the data associated with each process, and the data associated with each processor. Figure 2 shows the relationship of these data structures in the restructured kernel.

The per-process data is contained in the `proc` structure. It contains a list of kernel threads associated with the process, a pointer to the process address space, user credentials, and the list of signal handlers. The `proc` structure also contains the vestigial `user` structure, which is now much smaller than a page, and is no longer practical to swap.

The LWP structure contains the per-LWP data such as the process-control-block (pcb) for storing user-level processor registers, system call arguments, signal handling masks, resource usage information, and profiling pointers. It also contains pointers to the associated kernel thread and process structures. The kernel stack of the thread is allocated with the LWP structure inside a swappable area.

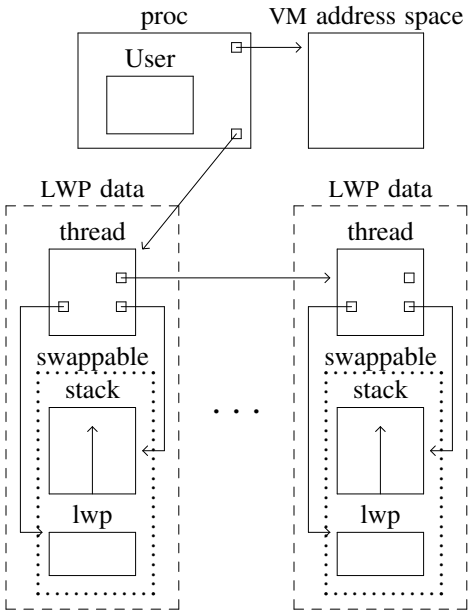

Figure 2: MT Data Structures for a Process

The kernel thread structure contains the kernel registers, scheduling class, dispatch queue links, and pointers to the stack and the associated LWP, process, and CPU structures. The thread structure is not swapped, so it also contains some data associated with the LWP that is needed even when the LWP structure is swapped out. Thread structures are linked on a list of threads for the process, and also on a list of all existing threads in the system.

Per-processor data is kept in the `cpu` structure, which has pointers to the currently executing thread, the idle thread for that CPU, and current dispatching and interrupt handling information. There is a substructure of the `cpu` structure that can be architecture dependent, but the main body is intended to be applicable to most multiprocessing architectures.

To speed access to the thread, LWP, process, and CPU structures, the SPARC implementation uses a global register, %g7, to point to the current thread structure. A C-preprocessor macro, `curthread`, allows access to fields in the current thread structure with a single instruction. The current LWP, process, and CPU structures are quickly accessible through pointers in the thread structure. In the future we may dedicate additional global registers for other frequently accessed structures.

Kernel Thread Scheduling

SunOS 5.0 provides several scheduling classes. A scheduling class determines the relative priority of processes within the class, and converts that priority to a global priority. With the addition of multithreading, the scheduling classes and dispatcher operate on threads instead of processes. The

scheduling classes currently supported are system, timesharing, and real-time (fixed-priority).

The dispatcher chooses the thread with the greatest global priority to run on the CPU. If more than one thread has the same priority, they are dispatched in round-robin order.

The kernel has been made preemptible to better support the real-time class and interrupt threads. Preemption is disabled only in a small number of bounded sections of code. This means that a runnable thread runs as soon as is practical after its priority becomes high enough. For example, when thread A releases a lock on which higher priority thread B is sleeping, the running thread A immediately puts itself back on the run queue and allows the CPU to run thread B. On a multiprocessor, if thread A has better priority than thread B, but thread B has better priority than the current thread on another CPU, that CPU is directed to preempt its current thread and choose the best thread to run. In addition, user code run by an underlying kernel thread of sufficient priority (e.g., real-time threads) will execute even though other lower priority kernel threads wait for execution resources. Further details can be found in [Khanna 1992].

System Threads

System threads can be created for short or long-term activities. They are scheduled like any other thread, but usually belong to the system scheduling class. These threads have no need for LWP structures, so the thread structure and stack for these threads can be allocated together in a non-swappable area, as shown in Figure 3.

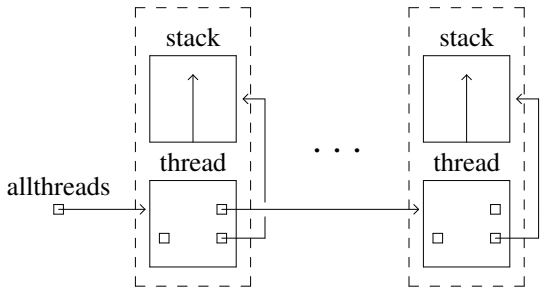

Figure 3: System Threads

A new segment driver, `seg_kp`, handles stack allocations. It handles virtual memory allocations for the kernel that can be paged or swapped out; it also provides ''red zones'' to protect against stack overflow. System threads use `seg_kp` for the stack and the thread structure, in a non-swappable region. LWPs use it to allocate the LWP structure and kernel stack in a swappable region.

Synchronization Architecture

The kernel implements the same synchronization objects for internal use as are provided by the user-level libraries for use in multithreaded application programs [Powell 1991]. These are mutual exclusion locks (*mutexes*), condition variables, semaphores, and multiple readers, single writer (readers/writer) locks. The interfaces are shown in Figure 4[1].

```
/* Mutual exclusion locks */
void  mutex_enter(kmutex_t *lp);
void  mutex_exit(kmutex_t *lp);

void  mutex_init(kmutex_t *lp, char *name,
        kmutex_type_t type, void *arg);
void  mutex_destroy(kmutex_t *lp);
int   mutex_tryenter(kmutex_t *lp);

/* condition variables */
void  cv_wait(kcondvar_t *cp, kmutex_t *lp);
int   cv_wait_sig(kcondvar_t *cp,
          kmutex_t *lp);
int   cv_timedwait(kcondvar_t *cvp,
          kmutex_t *lp, long tim);
void  cv_signal(kcondvar_t *cp);
void  cv_broadcast(kcondvar_t *cp);

/* multiple reader, single writer locks */
void  rw_init(krwlock_t *lp, char *name,
          krw_type_t type, void *arg);
void  rw_destroy(krwlock_t *lp);
void  rw_enter(krwlock_t *lp, krw_t rw);
int   rw_tryenter(krwlock_t *lp, krw_t rw);
void  rw_exit(krwlock_t *lp);
void  rw_downgrade(krwlock_t *lp);
int   rw_tryupgrade(krwlock_t *lp);

/* counting semaphores */
void  sema_init(ksema_t *sp,
          unsigned int val, char *name,
          ksema_type_t type, void *arg);
void  sema_destroy(ksema_t *sp);
void  sema_p(ksema_t *sp);
int   sema_p_sig(ksema_t *sp);
int   sema_tryp(ksema_t *sp);
void  sema_v(ksema_t *sp);
```

Figure 4: Kernel Thread Synchronization Interfaces

These are all implemented such that the behavior of the synchronization object is specified when it is initialized. Synchronization operations, such as acquiring a mutex lock, take a pointer to the object as an argument and may behave somewhat differently depending on the type and optional type-specific argument specified when the object was initialized.

Most of the synchronization objects have types that enable collecting statistics such as blocking counts or times. A patchable kernel variable can

[1]Note that kernel synchronization primitives must use a different type name than user synchronization primitives so that the types are not confused in applications that read internal kernel data structures.

also set the default types to enable statistics gathering. This allows the selection of statistics gathering on particular synchronization objects or on the kernel as a whole.

The semantics of most of the synchronization primitives cause the calling thread to be prevented from progressing past the primitive until some condition is satisfied. The way in which further progress is impeded (e.g., sleep, spin, or other) is a function of the initialization. By default, the kernel thread synchronization primitives that can logically block, can *potentially* sleep.

Some of the synchronization primitives are strictly bracketing (e.g., the thread that *locks* a mutex must be the thread that *unlocks* it) and a single owner can be determined (i.e., mutexes and writer locks). In these cases, the synchronization primitives support the priority inheritance protocol, as described in [Khanna 1992].

Some synchronization primitives are intended for situations where they may block for long or indeterminate periods. Variants of some of the primitives are provided (e.g., `cv_wait_sig()` and `sema_p_sig()`) that allow blocking to be interrupted by a reception of a signal. There is no non-local jump to the head of the system call, as there was in the traditional `sleep` routine. When a signal is pending, the primitive returns with a value indicating the blocking was interrupted by a signal and the caller must release any resources and return.

Mutual Exclusion Lock Implementation.

Mutual exclusion locks (mutexes) prevent more than one thread from proceeding when the lock is acquired. They prevent races on access to shared data and are by far the most heavily used primitive.

Mutexes are usually held for short intervals. For example, it would not be good to hold a critical system mutex while waiting for disk I/O to complete. Mutexes are not recursive; the owner of the lock cannot again call *mutex_enter()* for the same lock. If a thread holds a mutex, the same thread must be the one to release the mutex. These rules are enforced to promote good programming practice and to avoid deadlocks.

If *mutex_enter* cannot set the lock (because it is already set), the blocking action taken depends on the mutex type that was passed to *mutex_init*, and stored in the mutex. The default blocking policy for mutexes, called *adaptive* (type `MUTEX_DEFAULT`), spins while the owner of the lock (recorded when the lock is acquired) remains running on a processor. This is done by polling the owner's status in the spin wait loop[2]. If the owner ceases to run, the caller stops spinning and sleeps[3]. This gives fast response and low overhead for simple contention.

Spin mutexes are available as type `MUTEX_SPIN`, which takes as its type-specific argument the interrupt level to be disabled while the mutex is held. It is rarely used, as adaptive mutexes are more efficient, in general.

Device drivers are restricted to using type `MUTEX_DRIVER`, which takes a Sun-DDI-defined opaque value as an argument. This argument is basically an interrupt priority in the current implementation, and determines whether the blocking policy is adaptive or spin, based on whether the interrupt priority is above the ''thread level'' (see below).

A simple trick speeds up `mutex_enter()` for adaptive mutexes. Non-adaptive mutexes use a separate primitive lock field in the mutex data structure, with the lock field used by the adaptive type always in the locked state. This is so that `mutex_enter()` can always attempt to apply an adaptive lock first, and only if that fails, consider the possibility that the mutex might be another type.

Turnstiles vs Queues in Synchronization Objects

Each synchronization object requires a way of finding threads that are suspended waiting for that object. It is important to keep the storage cost of synchronization objects small, because many system structures contain synchronization objects, so the queue header is not directly in the object. Instead, two bytes in the synchronization object are used to find a *turnstile* structure containing the sleep queue header and priority inheritance information [Khanna 1992]. Turnstiles are preallocated such that there are always more turnstiles than the number of threads active.

One alternative method would be to select the sleep queue from an array using a hash function on the address of the synchronization object. This is essentially the approach used by `sleep()` in the traditional kernel. The turnstile approach is favored for more predictable real-time behavior, since they are never shared by other locks, as hashed sleep queues sometimes are.

Interrupts as Threads

Many implementations [Hamilton 1988] [Peacock 1992] have a variety of synchronization primitives that have similar semantics (e.g., mutual exclusion) yet explicitly sleep or spin for blocking. For mutexes, the spin primitives must hold interrupt priority high enough while the lock is held to

[2]In order to avoid locking while inspecting the owner's status during the spin, the state is determined indirectly. The algorithm spins while the current thread pointer of any CPU points to the owning thread, indicating it is running.

[3]On uniprocessors, this turns into always sleeping, since the owner cannot be running.

prevent any interrupt handlers that may also use the synchronization object from interrupting while the object is locked, causing deadlock. The interrupt level must be raised before the lock is acquired and then lowered after the lock is released.

This has several drawbacks. First, the raising and lowering of interrupt priority can be an expensive operation, especially on architectures that require external interrupt controllers (remember that mutexes are heavily used). Secondly, in a modular kernel, such as SunOS, many subsystems are interdependent. In several cases (e.g., mapping in kernel memory or memory allocation) these requests can come from interrupt handlers and can involve many kernel subsystems. This in turn, means that the mutexes used in many kernel subsystems must protect themselves at a relatively high priority from the *possibility* that they may be required by an interrupt handler. This tends to keep interrupt priority high for relatively long periods and the cost of raising and lowering interrupt priority must be paid for every mutex acquisition and release. Lastly, interrupt handlers must live in a constrained environment that avoids any use of kernel functions that can potentially sleep, even for short periods.

To avoid these drawbacks, the SunOS 5.0 kernel treats most interrupts as asynchronously created and dispatched high-priority threads. This enables these interrupt handlers to sleep, if required, and to use the standard synchronization primitives.

On most architectures putting threads to sleep must be done in software. This must be protected from interrupts if interrupts are to sleep themselves or wakeup other threads. The restructured kernel uses a primitive spin lock protected by raised priority to implement this. This is one of a few bounded sections of code where interrupts are locked out.

Traditional UNIX kernel implementations [Leffler 1989] [Bach 1986] also protect the dispatcher by locking out interrupts, usually all interrupts. The restructured kernel has a modifiable level (the ''thread level'') above which interrupts are no longer handled as threads and are treated more like non-portable ''firmware'' (e.g., simulating DMA via programmed I/O). These interrupt handlers can only synchronize using the spin variants of mutex locks and software interrupts. If the ''thread level'' is set to the maximum priority, then all interrupts are locked out during dispatching. For implementations where the ''firmware'' cannot tolerate even the relatively small dispatcher lockout time, the ''thread level'' can be lowered. Typically this is lowered to the interrupt level at which the scheduling clock runs.

Implementing Interrupts as Threads

Previous versions of SunOS have treated interrupts in the traditional UNIX way. When an interrupt occurs the interrupted process is held captive (*pinned*) until the interrupt returns. Typically, interrupts are handled on the kernel stack of the interrupted process or on a separate interrupt stack. The interrupt handler must complete execution and get off the stack before anything else is allowed to run on that processor. In these systems the kernel synchronizes with interrupt handlers by blocking out interrupts while in critical sections.

In SunOS 5.0 interrupts behave like asynchronously created threads. Interrupts must be efficient, so a full thread creation for each interrupt is impractical. Instead, we preallocate interrupt threads, already partly initialized. When an interrupt occurs, we do the minimum amount of work to move onto the stack of an interrupt thread, and set it as the current thread. At this point, the interrupt thread and the interrupted thread are not completely separated. The interrupt thread is not yet a full-fledged thread (it cannot be descheduled) and the interrupted thread is *pinned* until the interrupt thread returns or blocks, and cannot proceed on another CPU. When the interrupt returns, we restore the state of the interrupted thread and return.

Interrupts may nest. An interrupt thread may itself be interrupted and be pinned by another interrupt thread.

If an interrupt thread blocks on a synchronization variable (e.g., mutex or condition variable), it saves state (*passivates*) to make it a full-fledged thread, capable of being run by any CPU, and then returns to the pinned thread. Thus most of the overhead of creating a full thread is only done when the interrupt must block, due to contention[4].

While an interrupt thread is in progress, the interrupt level it is handling, and all lower-priority interrupts, must be blocked. This is handled by the normal interrupt priority mechanism unless the thread blocks. If it blocks, these interrupts must remain disabled in case the interrupt handler is not reenterable at the point that it blocked or it is still doing high-priority processing (i.e., should not be interrupted by lower-priority work). While it is blocked, the interrupt thread is bound to the processor it started on as an implementation convenience and to guarantee that there will always be an interrupt thread available when an interrupt occurs (though this may change in the future). A flag is set in the `cpu` structure indicating that an interrupt at that level has blocked, and the minimum interrupt level is noted. Whenever the interrupt level changes,

[4]On SPARC this overhead involves flushing the entire register file. This is only done if the interrupt handler sleeps, not during interrupt handling without contention.

the CPU's base interrupt level is checked, and the actual interrupt priority level is never allowed to be below that.

There is also an interface which allows an interrupt thread to continue as a normal, high-priority thread. When `release_interrupt()` is called, it saves the state of the the pinned thread and clears the indication that the interrupt thread has blocked, allowing the CPU to lower the interrupt priority level.

An alternative approach to this is to use bounded first-level interrupt handlers to capture device state and then wake up an interrupt thread that is waiting to do the remainder of the servicing [Barnett 1992]. This approach has the disadvantages of requiring device drivers to be restructured and of always requiring a full context switch to the second level thread. The approach used in SunOS 5.0 allows full thread behavior without restructured drivers and with very little additional cost in the no-contention case.

Interrupt Thread Cost

The additional overhead in taking an interrupt is about 40 SPARC instructions. The savings in the mutex enter/exit path is about 12 instructions. However, mutex operations are much more frequent than interrupts, so there is a net gain in time cost, as long as interrupts don't block too frequently. The work to convert an interrupt into a "real" thread is performed only when there is lock contention.

There is a cost in terms of memory usage also. Currently an interrupt thread is preallocated for each potentially active interrupt level below the thread level for each CPU[5]. An additional interrupt thread is preallocated for the clock (one per system). Since each thread requires a stack and a data structure, perhaps 8K bytes or so, the memory cost can be high.

However, it is unlikely that all interrupt levels are active at any one time, so it is possible to have a smaller pool of interrupt threads on each CPU and block all subsequent interrupts below the thread level when the pool is empty, essentially limiting how many interrupts may be simultaneously active.

Clock Interrupt

The clock interrupt[6] is handled specially. There is only one clock interrupt thread in the system (not one per CPU), and the clock interrupt handler invokes the clock thread only if it is not already active.

[5]There are nine interrupt levels on the Sun SPARC implementation that can potentially use threads.

[6]This occurs 100 times a second on current Sun SPARC implementations.

The clock thread could possibly be delayed for more than one clock tick by blocking on a mutex or by higher-level interrupts. When a clock tick occurs and the clock thread is already active, the interrupt is cleared and a counter is incremented. If the clock thread finds the counter non-zero before it returns, it will decrement the counter and repeat the clock processing. This occurs very rarely in practice. When it occurs, it is usually due to heavy activity at higher interrupt levels. It can also occur while debugging.

Kernel Locking Strategy

The locking approach used almost exclusively in the kernel to ensure data consistency is data-based locking. That is, the mutex and readers/writer locks each protect a set of shared data, as opposed to protecting routines (monitors). Every piece of shared data is protected by a synchronization object.

Some aspects of locking in the virtual memory, file system, STREAMS, and device drivers have already been discussed in [Kleiman 1992]. Here we'll elaborate a bit on device driver issues, as they are closely related to interrupt threads.

Non-MT Driver Support

Some drivers haven't been modified to protect themselves against concurrency in a multithreaded environment. These drivers are called *MT-unsafe*, because they don't provide their own locking.

In order to provide some interim support for *MT-unsafe* drivers, we provided wrappers that acquire a single global mutex, `unsafe_driver`. These wrappers insure that only one such driver will be active at any one time. This wrapper is illustrated by Figure 5.

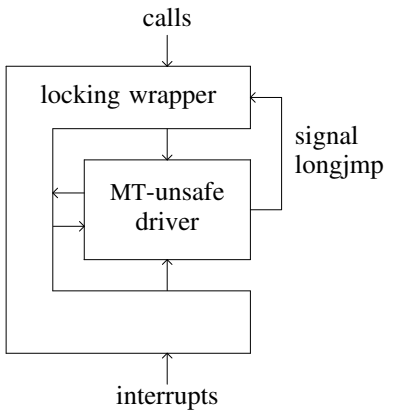

Figure 5: Unsafe Driver Wrapper

There are several ways a driver may be entered, from the explicit driver entry points, interrupts, and call-backs. Each of these entries must acquire the `unsafe_driver` mutex if the driver isn't safe. For example, if an MT-unsafe driver uses `timeout()` to request a function call at a later

time, the `callout` structure is marked so that the `unsafe_driver` mutex will be held during the function call.

MT-unsafe drivers can also use the old `sleep/wakeup` mechanism. `Sleep()` safely releases the `unsafe_driver` mutex after the thread is asleep, and reacquires it before returning.

The `longjmp()` feature of `sleep()` is maintained as well. When a thread is signalled in `sleep()`, if it specified a dispatch value greater than PZERO, a `longjmp()` takes the thread to a `setjmp()` that was performed in the unsafe driver entry wrapper, which returns EINTR to the caller of the driver.

`Sleep()` checks to make sure it is called by an MT-unsafe driver, and panics if it isn't. It isn't safe to use `sleep()` from a driver which does its own locking.

It is fairly easy to provide at least simple locking for a driver, so almost all drivers in the system have some of their own locking. These drivers are called *MT-safe*, regardless of how fine-grained their locking is. Some developers have used the term *MT-hot* to indicate that a driver does fine-grained locking.

SVR4/MP DKI Locking Primitives

As we implemented our driver interfaces, UNIX International and USL were defining the SVR4 Multiprocessor Device Driver Interface and Driver-Kernel Interface (DDI/DKI), with a different set of locking primitives, based around the traditional UNIX interrupt-blocking model.

SunOS 5.0 implements those interfaces to the extent defined so far, using our locking primitives and ignoring any spin semantics. This allows drivers using those interfaces to be more easily ported. SunOS drivers typically use the the SunOS synchronization primitives.

Implementation Technology

Some interesting techniques made it easier to get this all working.

Kernel Time Slicing

Since the kernel is fully preemptible we were able to make kernel threads time-slice. We simply added code to the clock interrupt handler to preempt whatever thread was interrupted. This allows even a uniprocessor to have almost arbitrary code interleavings. Increasing the clock interrupt rate made this even more valuable in finding windows where data was unprotected. By causing kernel threads to preempt each other as often as possible we were able to find locking problems using uniprocessor hardware before multiprocessor hardware was available. Even when working multiprocessor hardware arrived, there were far more uniprocessors available than multiprocessors. We intend this only as a debugging feature, since it does have some adverse performance impact, however slight.

Lock Hierarchy Violation Detection

Instead of establishing a system lock hierarchy *a priori*, we developed a static analysis tool that would check for lock ordering violations in the system. This lint-like tool, called *locknest*, reads C source code, constructs call graphs and reports on locking cycles. We feel it helped during early implementation debugging, and probably reduced the amount of time spent debugging deadlocks. A similar tool is described in [Korty 1989].

Deadlock Detection

A side-benefit of the priority inheritance mechanism [Khanna 1992], is that deadlocks caused by hierarchy violations are usually detected at run time as well. It does a good job on mutexes and readers/writer locks held for write, but since there isn't a complete list of threads holding a read lock, it can't always find deadlocks involving readers/writer locks. There are other deadlocks possible with condition variables; these aren't detected.

Summary

SunOS 5.0 is a multithreaded and symmetric multiprocessor version of the SVR4 kernel. The primary features are:

- Fully preemptible, real-time kernel
- High degree of concurrency on symmetric multiprocessors
- Support for user threads
- Interrupts handled as independent threads
- Adaptive mutual-exclusion locks

The thread models inside the kernel and at user level are almost identical. The scheduling of kernel threads onto CPUs is analogous to the way the threads library schedules user-level threads onto LWPs. The use of threads for structuring the kernel has mostly good effects though they can be overused. Threads do have a cost. The stacks are large, and must be allocated on separate pages if protection for potential stack overrun is needed. Also, context switching is still expensive. Some things are still better implemented by callouts and other ''zero-weight'' processes, but threads provide a nice structuring paradigm for the kernel.

References

[Bach 1986] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[Hamilton 1988] Graham Hamilton and Daniel S. Conde, *An Experimental Symmetric Multiprocessor Ultrix Kernel*, USENIX, Winter 1988,

Dallas, Texas.

[Kleiman 1992] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner, *Symmetric Multiprocessing in Solaris 2.0*, COMPCON Spring 1992, p181, San Francisco, California.

[Khanna 1992] Sandeep Khanna, Michael Sebrée, John Zolnowsky, *Realtime Scheduling in SunOS 5.0*, USENIX, Winter 1992, San Francisco, California. This describes the real-time features and considerations in this kernel.

[Korty 1989] Joe Korty, *Sema: a Lint-Like Tool for Analyzing Semaphore Usage in a Multithreaded Unix Kernel,* USENIX Winter 1989, San Diego, California. This describes a tool for doing static lock hierarchy analysis.

[Leffler 1989] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[Barnett 1992] David Barnett, *Kernel Threads and their Performance Benefits*, Real Time, Vol. 4, No. 1, Lynx Real-Time Systems, Inc., Los Gatos, CA., 1992.

[Peacock 1992] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang and Wilfred Yu, *Experiences from Multithreading System V Release 4*, Symposium on Experiences with Distributed & Multiprocessor Systems (SEDMS) III, March 1992, Newport Beach, California.

[Powell 1991] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, *SunOS Multi-thread Architecture,* USENIX Winter 1991, Dallas, Texas. This describes the architecture for user-level multi-threading.

[Stein 1992] D. Stein, D. Shah, *Implementing Lightweight Threads*, USENIX Summer 1992, San Antonio, Texas. This describes the implementation of the user-level threads package.

Author Information

Joseph Eykholt is a Senior Staff Engineer and technical leader in the OS-Multithreading group at SunSoft. He received an MSEE from Purdue University in 1978, and a BSEE from Purdue in 1977. Prior to coming to Sun, he was one of the leading developers of multiprocessing features for the Amdahl UTS system, and a logic designer for the Amdahl 580 CPU. His address is SunSoft, Inc., M/S MTV5-40, 2550 Garcia Avenue, Mountain View, CA, 94043. His E-mail address is jre@Eng.Sun.COM. By phone: (415) 336-1849.

Steve Kleiman is a Distinguished Engineer in the Operating Systems Technology Department of SunSoft. He is currently architect of Multi-threading in SunOS. He received an M.S. in Electrical Engineering from Stanford University in 1978 and a B.S. in Electrical Engineering and Computer Science from M.I.T in 1977. He has been involved with the design and development of UNIX and workstation architecture since 1977; first at Bell Telephone Laboratories and then at Sun. He was one of the developers of NFS, Vnodes, and of the original port of SunOS to SPARC. His E-mail address is srk@Eng.Sun.COM. By phone: (415) 336-7295.

Steve Barton graduated from the University of California, Santa Cruz in 1982 with a BA in Computer and Information Sciences. Since then he has worked at Zilog Inc., Parallel Computers, Counter-Point Computers, and Telestream Corp. He is currently a Member of Technical Staff at Sunsoft. He's been with Sun for the last four years. Reach him electronically at steve.barton@Eng.Sun.COM.

Roger Faulkner is a Senior Staff Engineer in the OS-Multithreading group at SunSoft. He received a B.S. in Physics from N. C. State University in 1963 and a Ph.D. in Physics from Princeton University in 1967, then joined Bell Laboratories, where he was seduced by computers. He has been actively involved in the inner workings of the UNIX kernel since 1976 and has done compiler and debugger development along the way. He is one of the principals involved in the development of the /proc file system for SVR4. His E-mail address is raf@Eng.Sun.COM. By phone: (415) 336-1115.

Anil Shivalingiah is a Staff Engineer in the OS-Virtual Memory group at SunSoft. He received an M.S. in Computer Science from University of Texas, Arlington in 1983 and a B.S. in Electronics Engineering from UVCE, India in 1981. He's been with Sun for the last three years. His E-mail address is ans@Eng.Sun.COM.

Mark Smith graduated with a B.S. in Computer Science from the University of California, Santa Barbara in 1986. He is currently a Member of Technical Staff at SunSoft in the OS-Multithreading group. Prior to coming to Sun he worked in the Design Automation department of Amdahl Corp. He can be reached by E-mail at mds@Eng.Sun.COM.

Dan Stein is currently a Member of Technical Staff at Sunsoft where he is one of the developers of the SunOS Multi-thread Architecture. He graduated from the University of Wisconsin in 1981 with a BS in Computer Science.

Jim Voll works in the OS-Multithreading group at SunSoft. He received his B.S. from the University of California, Santa Barbara in 1981. Prior to working at Sun he has worked at Amdahl and Cygnet Systems. He routinely destroys his home directory. His E-mail address is jjv@Eng.Sun.COM.

Mary Weeks has been a member of technical staff at Sun Microsystem since 1986. Prior to Sun, she worked at Xerox. She received her B.A. in computer science from the University of California at

Berkeley in 1984.

Dock Williams is a Staff Engineer in the OS-Multithreading group at SunSoft. He received a S.B. in Electrical Engineering and Computer Science from M.I.T. in 1980. He has been with Sun for over six years. Prior to joining Sun, he worked at American Information Systems, ONYX Systems, Tri-Comp Systems, and Hughes Aircraft Radar Systems. His E-mail address is dock@Eng.Sun.COM, phone: (415) 336-1246.