Realtime Scheduling in SunOS 5.0

Sandeep Khanna
Michael Sebrée
John Zolnowsky

SunSoft Incorporated

Abstract

We describe the fundamental mechanisms in SunOS 5.0 to provide realtime scheduling functionality. Our primary goal was to provide bounded behavior for dispatching or blocking threads. To achieve this goal we have modified the kernel to be *fully preemptive*, guaranteeing dispatch after both synchronous and asynchronous wakeups. We have also worked toward controlling *priority inversion* in the kernel. The result is a kernel capable of delivering realtime scheduling and bounded response to a large class of user level applications.

1.0 Introduction

A realtime operating system may be defined as one that has the ability to provide a required level of service in a bounded response time. To achieve a bounded response time, time-critical applications require control over their scheduling behavior. Increasing numbers of new, interesting applications for the desktop possess time-critical aspects. Some examples that come immediately to mind are in the area of user interfaces, multimedia, and virtual reality. These applications are typically "mixed-mode," that is, they are partitionable into schedulable entities, some but not all of which require realtime response. It seems desirable to provide a standards-conformant, full-featured environment like SunOS for the tasks without realtime requirements. But to support these applications, SunOS must be able to provide some realtime capability to those tasks that are time-critical. From our desires to provide realtime capability and to use SunOS as the basis of our work, we derived the set of requirements listed below.

- The scheduling of tasks in the kernel should be deterministic. By deterministic scheduling, we mean the kernel should provide priority-based scheduling for user tasks, so that the time-critical application developer has control of the scheduling behavior of the system; the kernel should provide bounded dispatch latency, so that time-critical user tasks are not subjected to unexpected and undesirable delays; the kernel should be free from unbounded priority inversions.

- No draconian demands should be placed on application behavior in order to obtain realtime response. This is important for mixed-mode applications including non-realtime components which require the general services of a UNIX environment.

- The resultant operating system should be appropriate for multiprocessor machines.

- The resultant operating system should present a standard interface to the programmer and user. In particular, the interface that we must support is that described in the System V Interface Definition [AT&T 1989].

Historical implementations of UNIX have not provided bounded dispatch latency. The principal failure of these systems was that a process executing in the kernel was not preemptible. A low-priority process would retain control of the processor until the process either blocked or attempted to return to user state.

One possible solution was the use of preemption points whereby, at various points throughout the kernel, code was inserted to check if the current process should be preempted and, if so, force the preemption [AT&T 1990]. These preemption points could be inserted, however, only where the process could recover from the effects of the preemption. For instance, an exiting process releasing its memory resources could not be preempted if its process state was not safe for scheduling. Hence, while a kernel with preemption points might be able to provide bounded dispatch latency, it might be guaranteed only while no process used certain kernel facilities.

SunOS 5.0 has a fully preemptible kernel, based on fully synchronized access by kernel code to kernel storage and resources. The elimination of the pervasive use of elevated processor interrupt levels to mask interrupts leaves a small set of non-preemption intervals. This permits immediate preemption when a higher priority task becomes runnable. Another relevant feature of SunOS 5.0 is the use of dynamically loaded kernel modules to enhance kernel extensibility.

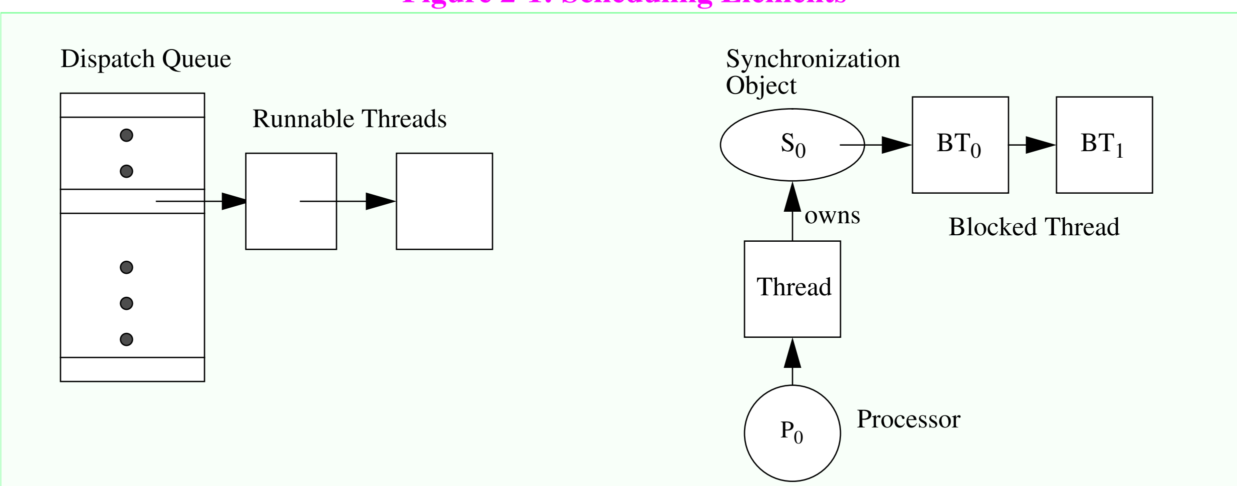
The rest of this paper consists of the following sections: Details of Implementation, which describes the objects and methods involved in implementing realtime scheduling in the kernel; Lessons Learned, where we discuss some of the problems of our implementation; Performance, where we describe the results of our work and the measurement methods used; and Futures, where we discuss areas for possible improvement and enhancement.

2.0 Details of Implementation

Several elements compose the realtime scheduling features of SunOS 5.0. The foremost of these is that the internal architecture of SunOS is based on threads [Powell 1991]. Within the kernel, threads interact through the use of shared memory and synchronization objects.

The user and programmer interfaces for realtime scheduling are those provided by SVR4. Runnable threads are queued in a system-wide dispatch queue array, and the scheduler determines when and which threads are to be dispatched for execution on the system processors. We describe below the scheduler activities in kernel preemption, multiprocessor scheduling, and the techniques we used to lower dispatch latency.

Figure 2-1: Scheduling Elements



2.1 Elements

Figure 2-1 illustrates the scheduling elements of SunOS 5.0. The fundamental unit of scheduling in this system is the thread. A thread is a single flow-of-control. Each thread possesses a register state and a stack. The system associates with each thread extra state information relating to its schedulability. These include the thread dispatch priority and processor affinity that determines on which processors a thread can execute. In SunOS 5.0, processor affinities are not user accessible; the default is to execute on all processors.

The primary example of a thread is an LWP executing within a process [Powell 1991]. Such a thread has extra associated information for accessing, for example, file descriptors, user credentials, and signal context. Kernel daemon threads are responsible for pageout, swapping, the background servicing of STREAMS. An idle thread is selected for execution whenever no other thread is runnable, and switches whenever another thread becomes runnable.

Threads interact using synchronization objects. A *mutex* allows a single thread access to the information protected by the mutex. A *condition variable* allows threads to suspend execution, waiting for some change in the condition which the thread requires. A *readers/writer lock* allows a single thread to modify or multiple threads to examine some shared information. A *counting semaphore* permits synchronization from a thread that does not wish to block. Operations on synchronization objects include acquisition, which may involve blocking the calling thread, and release, which may involve unblocking other threads.

Interrupt processing is performed by interrupt threads, created at interrupt time and exiting when interrupt processing is complete [SunSoft MT, to appear]. One particular interrupt thread, the clock thread, is dispatched upon clock interrupt, and is responsible for time-based scheduling activities. Currently, interrupt threads have an affinity for the processor which took the interrupt, and so cannot migrate to another processor. Interrupt and foreground threads use synchronization objects to interact, and hence the pervasive use of elevated interrupt level in the kernel is eliminated. Asynchronous wakeup refers to the occasion when an interrupt thread releases a synchronization object and unblocks another thread; we use the term synchronous wakeup when this is done by a foreground thread.

2.2 Priority Model

We associate with each thread a number of priority values: dispatch priority, global priority, inherited priority, and typically, a user priority. The user priority, together with other application parameters, determines the thread's global priority. The inherited priority, derived through priority inheritance as described below, and the global priority determine the dispatch priority, the actual value used in queuing and selecting a thread for execution.

The dispatcher, illustrated in Figure 2-1, uses an array of dispatch queues, indexed by dispatch priority. When a thread is made runnable, it is placed on a dispatch queue, typically at the end, corresponding to its dispatch priority. When a processor switches to a new thread, it always selects the thread at the beginning of the highest priority nonempty dispatch queue. Threads may not change dispatch priority while on a dispatch queue; the thread must be first removed, its dispatch priority adjusted, and then the thread may be placed on a different dispatch queue.

When a thread needs to wait on a synchronization object, it is placed on a sleep queue associated with the synchronization object. The sleep queue is maintained in dispatch priority order, so that when the synchronization object is released, the highest priority thread waiting for the object is at the head of the sleep queue.

2.3 Scheduling Attributes

In SunOS 5.0 threads are divided into of the scheduling classes. Each class chooses the attributes for the priority of threads in that class. These attributes are determined in the class dependent functions supplied by each scheduling class. Scheduling class dependent code must abide by certain rules expected by the class independent code, such as that the higher the priority value, the higher the priority of the thread. However, scheduling class dependent functions have the flexibility to decide the range of priority values for threads belonging to the class, and the class dependent functions also determine when (if ever) a thread's priority value changes.

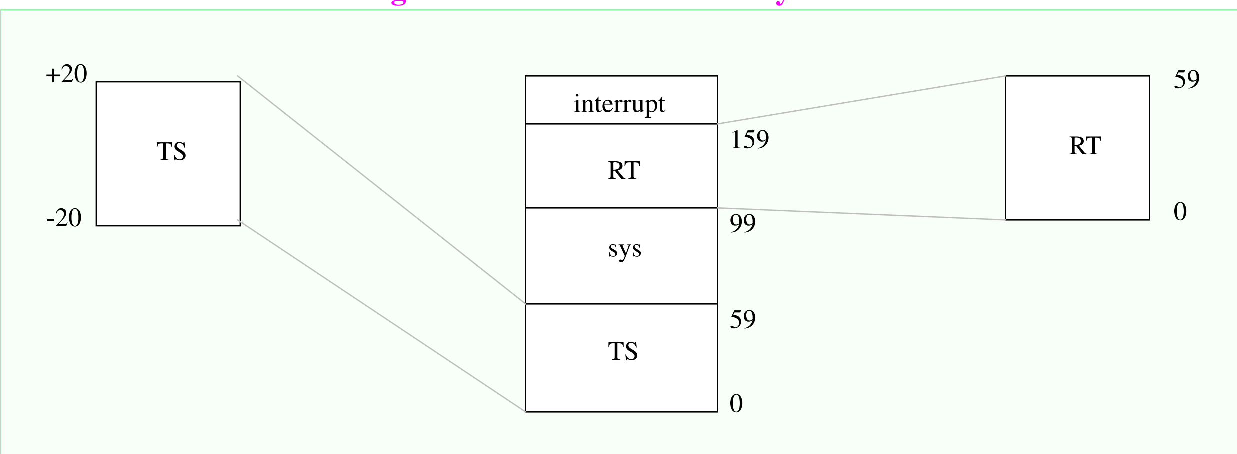
A new thread inherits the scheduling class of its parent. Associated with each thread is a class id, *t_cid* and a pointer to the class specific data, *t_cldata*. Within the class specific data is the class specific priority, the time quantum associated with the thread and the other class related data. A thread may change its scheduling class by using the **priocntl**(2) system call. The **priocntl**(2) system call may also be used to change other parameters associated with thread processor usage.

SunOS 5.0 by default supports three scheduling classes. The **time-sharing** class supports a time slicing technique for threads using the processor. Time-sharing class threads are scheduled dynamically, with a few hundred milliseconds per time slice. The time-sharing scheduler switches context in round-robin fashion often enough to give every thread an equal opportunity to run. The **sys** class exists for the purpose of scheduling the execution of special system threads and interrupt threads. Threads in the **sys** class have fixed priorities established by the kernel when they are started. It is not possible for a user thread to change its class to **sys** class. The **realtime** class supports a fixed priority technique of processor access. Realtime threads are scheduled strictly on the basis of their priority and the time quantum associated with them. A realtime thread with infinite time quantum runs until it terminates, blocks or is preempted. Figure 2-2 shows the default configuration of scheduling classes in SunOS 5.0.

Interrupt thread priorities are computed such that they are always the highest priority threads in the system. If a scheduling class is dynamically loaded, the priorities of the interrupt threads are recomputed to ensure that they remain the highest priority threads in the system.

Each scheduling class has a unique scheduling policy for dispatching threads within its class and a set of priority levels which apply to threads in that class as illustrated in Figure 2-2. A class-specific mapping translates these priorities into a set of global priorities. The user can configure the ranges as well as the global mapping associated with each class.

Figure 2-2: The Global Priority Model



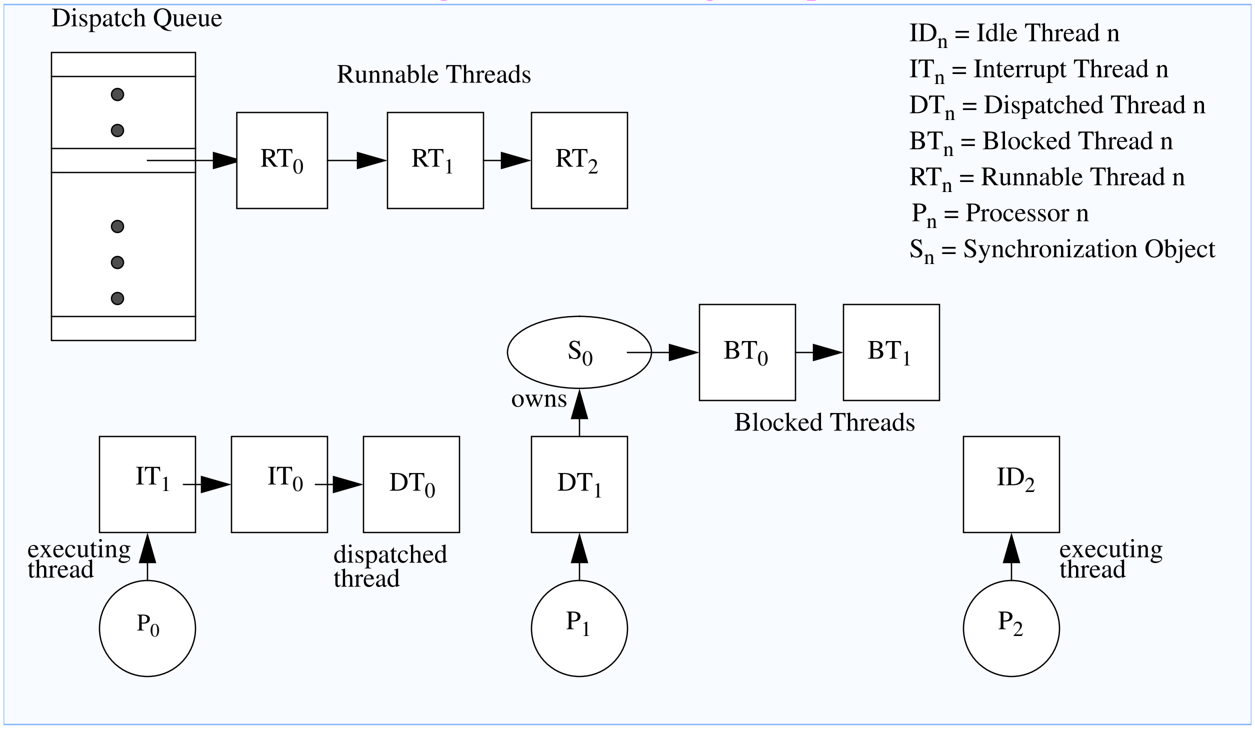
2.4 Scheduling

SunOS 5.0 is designed to run on a shared-memory multiprocessor system. The set of threads, threads' data, and synchronization objects are shared by all processors, and the system has a single dispatch queue for all processors. We assume that each processor can send an interrupt to any other processor. Except for specially configured threads bound to a single processor, threads may be selected for dispatch on any processor.

As far as scheduling is concerned, threads can be in one of three states: blocked, runnable, or executing. Figure 2-3 shows examples of each of these cases. Blocked threads are those waiting on some synchronization object, such as BT_n. When the object is released, the highest priority or all waiting threads in the sleep queue are made runnable. Unblocked threads are placed at the end of the dispatch queue for their dispatch priority, such as RT_n. The thread enters the executing state when a processor selects it for execution, such as DT_n. An executing thread may become a blocked thread by waiting on a synchronization object, or may be preempted by a higher priority thread and placed back on the dispatch queue as a runnable thread. When an executing thread blocks, the system dispatches another thread.

In a uniprocessor system, realtime scheduling is defined to mean that the highest priority thread is dispatched, within a bounded time of its becoming runnable. The obvious extension to a n-processor environment is that the n highest priority processes should be dispatched. Unfortunately, this state is not achievable in a system in which some threads are restricted to or from certain processors. We have made a more local generalization, emphasizing scheduling operations in terms of a single thread or single processor at a time. We chose the following design objective: a runnable thread will be dispatched if it has higher priority than some thread currently executing on a processor for which this thread has affinity. Stated from the processor point of view: every processor is executing a thread with at least as high dispatch priority as the highest among the runnable threads having affinity for this processor. This objective can guarantee the dispatch latency for a thread only if it remains the highest priority thread on the dispatch queue until dispatched. Each arrival of a higher priority thread while a thread is awaiting dispatch restarts its dispatch latency interval.

Figure 2-3: Scheduling Example



Currently, a single spin lock, **schedlock**, protects all scheduling operations. In particular, whenever the release of a synchronization object makes some thread runnable, **schedlock** is held while placing the thread on the dispatch queue. The function **sched_lock**() obtains this lock, and **sched_unlock**() releases the lock. If the lock is currently held, the processor will spin on the lock, waiting for access. To prevent interference and delays from interrupt routines, the holder of **schedlock** runs at an elevated processor interrupt level.

Associated with each processor is a set of scheduling variables: *cpu_thread, cpu_dispthread*, *cpu_idle*, *cpu_runrun*, *cpu_kprunrun*, and *cpu_chosen_level*. The *cpu_thread* value refers to the thread currently executing on the processor, and is changed whenever execution of a different thread begins. The *cpu_dispthread* value records the identity of the thread last selected for dispatch on that processor. The *cpu_idle* value refers to a special idle thread allocated for this processor, having a priority lower than any dispatch priority, and never appearing in the dispatch queue. The *cpu_runrun* and *cpu_kprunrun* values record requests for preemption of the current thread. The *cpu_chosen_level* records the priority of the thread which is slated to preempt the thread currently executing on that processor.

Threads are placed on the dispatch queue by one of two functions: **setfrontdq**() or **setbackdq**(). The **setfrontdq**() function is used primarily when a thread is preempted, to place it at the head of its dispatch queue so that when a thread is next chosen for dispatch from its level, it will be the first selected. The **setbackdq**() function places a thread at the tail of its dispatch queue. Thus a newly-awakened thread will be dispatched only after all other threads at its dispatch priority have been dispatched.

After **setfrontdq**() or **setbackdq**() has put a thread on the dispatch queue, the **cpu_choose**() function is called to find a processor on which the runnable thread might be dispatched. This function finds the processor with the lowest priority dispatched thread. If this thread has a lower priority than the newly runnable thread, that processor is marked for preemption, that processor's *cpu_chosen_level* is set to the new thread's priority, and if necessary, the interprocessor interrupt is sent. The *cpu_chosen_level* is used by later calls to **cpu_choose**() as an indication that a higher priority thread is intended for this processor.

Table 1: Scheduling Interfaces

Function	Description
setfrontdq	Put a thread onto the front of the dispatch queue.
setbackdq	Put a thread onto the back of the dispatch queue.
cpu_choose	Determine a processor to execute a thread.
cpu_surrender	Have a thread give up its processor.
disp	Select a thread for execution from the dispatch queue.
swtch	Select the next thread to execute.
kpreempt	Attempt to preempt the kernel.
kpreempt_disable	Disable preemption for a critical interval.
kpreempt_enable	Reenable preemption.

When a thread which is currently executing on a processor has its dispatch priority lowered below that of the highest priority runnable thread, either through the effect of a **priocntl**() call or by loss of inheritance (§2.5.2), it is appropriate for that thread to be preempted by the higher priority thread. The **cpu_surrender**() function accomplishes this, finding the processor on which the target thread is executing and marking it for preemption, and sending the interprocessor interrupt, if necessary. The level of preemption is determined by comparison with a system parameter, *kpreemptpri*, with lower levels causing user preemption and higher levels causing kernel preemption.

Before we can describe the dispatch operation, some details concerning interrupt thread creation and termination are required. For efficiency, the dispatch of an interrupt thread is not a complete dispatch operation. Instead, the thread executing at the time of the interrupt is pushed onto a LIFO list, and execution of the interrupt thread begins immediately, with cpu_thread set to refer to the interrupt thread. Since the pushed thread cannot be dispatched until the interrupt thread terminates or blocks, the interrupted thread is called "pinned." If the interrupt thread terminates without blocking, as would typically be the case, the head of the interrupt list is popped off for execution [SunSoft MT, to appear].

The **swtch**() function provides the fundamental operation of scheduling. If the calling thread is an interrupt thread with a pinned thread, the pinned thread is unpinned and execution is switched to that thread. Otherwise, the function **disp**() is called to select the highest priority thread eligible for execution on this processor. If no runnable thread exists for this processor, **disp**() returns the idle thread for this processor (ID_2 in Figure 2-3).The **disp**() function automatically resets the *cpu_chosen_level* and the *cpu_kprunrun* and *cpu_runrun* preemption flags, and sets *cpu_dispthread* to reflect the newly dispatched thread.

SunOS 5.0 has two modes of preemption, user and kernel. User level preemption refers not to a user level request, but a "lazy" preemption which is deferred until the thread last dispatched attempts to return to user mode; this corresponds to the historical notion of *runrun* [Leffler 1989]. User level preemption forces a **swtch**() call before resuming user state execution of a process; this level of preemption is requested by setting the *cpu_runrun* flag associated with the processor. User preemption requests are recognized at the end of trap or system call processing.

Kernel preemption requests an immediate **swtch**() call, and is requested by setting the *cpu_kprunrun* flag. These requests can occur because of scheduling operations on either the same or another processor. The requests on the same processor may occur while executing the thread to be preempted, or may occur while executing an interrupt thread pinning the thread to be preempted. Requests occurring on another processor result in a interprocessor interrupt, and thus can be processed like preemption requests occurring during local interrupts.

Thus, kernel preemption requests need only be checked at two places: in the **sched_unlock**() routine, and at the end of interrupt processing. In both instances, if the *cpu_kprunrun* flag is set, the **kpreempt**() routine is called. This routine determines whether circumstances are appropriate for preemption, or should be deferred. Examples of circumstances under which preemption is deferred include when the executing thread is already slated to call **swtch**(), the calling thread is the idle thread or an interrupt thread, or preemption is disabled. The latter condition arises when some processor state, such as floating point or memory management context is in flux. These conditions are bracketed by calls to **kpreempt_disable**() / **kpreempt_enable**(), with the latter function checking for a deferred preemption.

The thread time quantum is enforced by class-specific code, which marks the class specific data to indicate an expired thread, and requests preemption. When the actual preemption occurs, the class-specific code checks the expiration mark, and if so, calls **setbackdq**() rather than **setfrontdq**() when placing the current thread on the dispatch queues, thus providing round-robin scheduling.

2.5 Priority Inversion

Priority inversion is the condition that occurs when the execution of a high priority thread is blocked by a lower priority thread. If the duration of priority inversion in a system is unbounded, it is said to be uncontrolled. Uncontrolled priority inversion can cause unbounded delays during blocking, resulting in missed deadlines even under very low levels of processor utilization. During the design phase, we identified two types of priority inversion as particular areas of concern. These areas are *hidden scheduling* and the priority inversion problem associated with the use of synchronization objects, usually called simply the *priority inversion problem*.

2.5.1 Hidden Scheduling

We define hidden scheduling as that work done asynchronously in the kernel on behalf of threads without regard to their priority. One example is the traditional model of streams processing. In this traditional model, whenever a process is about to return from the kernel to user mode, the kernel checks to see if there are any requests pending in the streams queues, and if so, these requests are processed before the thread returns to user mode. In effect, these requests are being handled at the wrong priority.

Another example is the processing of the callout-queue, a mechanism for scheduling delayed processing of specified functions. The **timeout**() function puts requests on the callout-queue; its interface is specified in Table 2. Regardless of whether a request for delayed processing was issued by a timesharing thread, a realtime thread or a system thread, callout processing is done at the lowest interrupt level. It is possible that the time for a request issued by a time-sharing thread arrives while a realtime thread is running. This will cause the realtime thread to be interrupted, resulting in priority inversion. Since neither the number nor the duration of the functions on the callout-queue are predictable, the duration of priority inversion is non-deterministic, and hence is uncontrolled. Measurements taken under SunOS 4.0.3 and SunOS 4.1 have shown that it could take longer than 5 milliseconds to process the callout-queue. Such unscheduled delays are unacceptable in an operating system that purports to offer realtime response.

Table 2: Timeout Interfaces

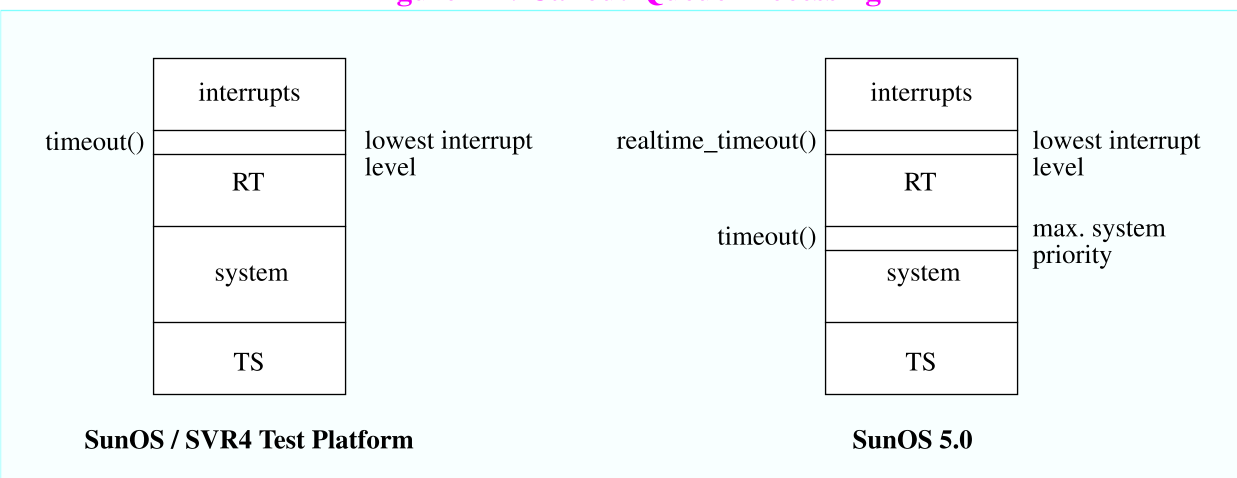
Interface	Description
timeout (*func*, *arg*, *t*)	Schedule the function *func* to be called at time *t* from now. Processed at the highest priority for system threads.
realtime_timeout (*func*, *arg*, *t*)	Schedule the function *func* to be called at time *t* from now. Processed at the lowest software interrupt level.

In the ideal case, all work done in the kernel would be done at the priority of the thread that requested the work. To do this for the callout mechanism would entail associating a priority with each request and creating a thread at that priority to do the requested work. The obvious drawbacks of this approach are the creation of many threads, more context switches, and the major restructuring required of the existing callout-queue framework. Another possible alternative is to process only the smaller requests at interrupt level and process the remainder at a lower priority. This approach requires complete knowledge of the entire system. With dynamically loadable modules this is not a feasible solution.

To address the problem of hidden scheduling, we have whenever possible followed a policy of moving kernel processing of this sort into kernel threads, so that the work does not preempt or run at the expense of realtime work on the system. The processing of the streams queues has been dealt with in this fashion [SunSoft MT, to appear]. Similarly, the delayed processing scheduled by the **timeout**() function is done by a kernel thread, **callout_thread**, running at the highest **sys** class priority. In the default configuration, this thread runs at a priority immediately below the realtime class priorities. As before, at every clock tick, the **callout_thread** checks to see if any timeout processing needs to be done. The difference is that, instead of being called at interrupt level, now it simply sleeps on a condition variable periodically signalled by the clock interrupt. The **callout_thread** runs only after all the realtime threads run, thereby avoiding priority inversion due to delayed processing.

Yet realtime threads may need to set a timer/alarm so that they can be awakened in realtime. This problem has been addressed by adding a function called **realtime_timeout**(). The requests made via **realtime_timeout**() are run at the lowest interrupt level and are kept on a separate heap. Figure 2-4 shows the priority space model before and after the changes for **realtime_timeout**(). Only time-critical interfaces use this function. For example, the interval timer and its interface, **setitimer**() are based on **realtime_timeout**(). The interfaces for **timeout**() and **realtime_timeout**() are identical as shown in Table 2.

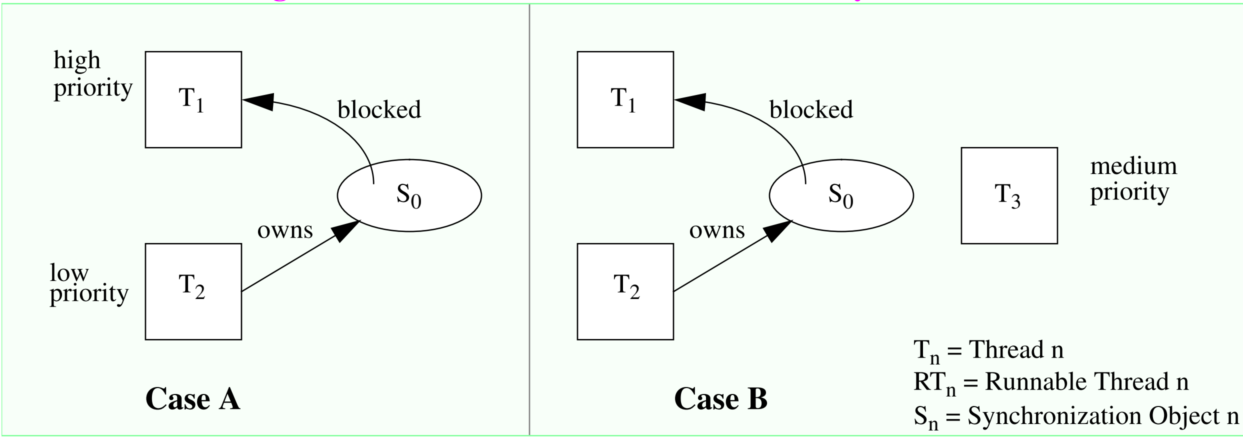
Figure 2-4: Callout-Queue Processing



2.5.2 The Priority Inversion Problem

The priority inversion problem was first identified by Lampson and Redell in their discussion on the use of monitors in Mesa [Lampson 1980]. It has received a moderate amount of attention lately in the literature [Rajkumar 1988] [Sha 1990]. What follows is a brief description of the problem as it pertains to SunOS. SunOS 5.0 is a multithreading kernel that uses synchronization objects such as *mutexes* and *readers/writer locks* to enforce thread synchronization. The use of such synchronization objects can lead to uncontrolled priority inversion. By way of illustration, consider the case where a high priority thread that uses a *mutex* gets blocked during the periods of time when a lower-priority thread that owns the *mutex* is preempted by intermediate-priority threads. These periods of time are potentially very long—in fact, they are unbounded, since the amount of time a high priority thread must wait for a *mutex* to become unlocked may depend not only on the duration of some critical sections, but on the duration of the complete execution of some threads. Figure 2-5 below illustrates these situations. Case A shows simple priority inversion: low priority thread T_2 blocks high priority thread T_1 because T_2 holds synchronization object S_0; Case B is identical, save that medium priority thread T_3 preempts the execution of T_2, with the result that T_1's blocking time depends on the duration of T_3's execution.

Figure 2-5: Bounded and Unbounded Priority Inversion



To solve the priority inversion problem, we have chosen to implement the *basic priority inheritance protocol*. A complete and detailed discussion of this protocol is beyond the scope of this paper, but its essentials are easily described. A more elaborate description may be found in [Sha 1990]. The basic priority inheritance protocol attempts to limit the duration of priority inversion during blocking by having a blocked high priority thread propagate (will) its priority to all lower priority threads that block it. When lower priority threads cease to block a high priority thread, the lower priority threads revert to their original priority.

The basic priority inheritance protocol concerns itself with two general abstract data types. These abstract data types are the schedulable entity, in our case the thread, and synchronization objects, in our implementation mutexes and readers/writer locks. Let us consider what happens in the general case when threads acquire and release synchronization objects. Before a thread T_1 enters a critical section, it attempts to acquire ownership of the synchronization object S guarding the critical section. If the synchronization object S is already owned by thread T_2, the attempt by thread T_1 to acquire synchronization object S fails, and thread T_1 blocks. In this scenario, thread T_1 is said to be blocked on synchronization object S and blocked by thread T_2. If the synchronization object S is not already owned by another thread, thread T_1 will acquire ownership of synchronization object S and enter the critical section that it guards. When thread T_1 exits this critical section, it releases synchronization object S and awakens the highest priority thread blocked on S by T_1.

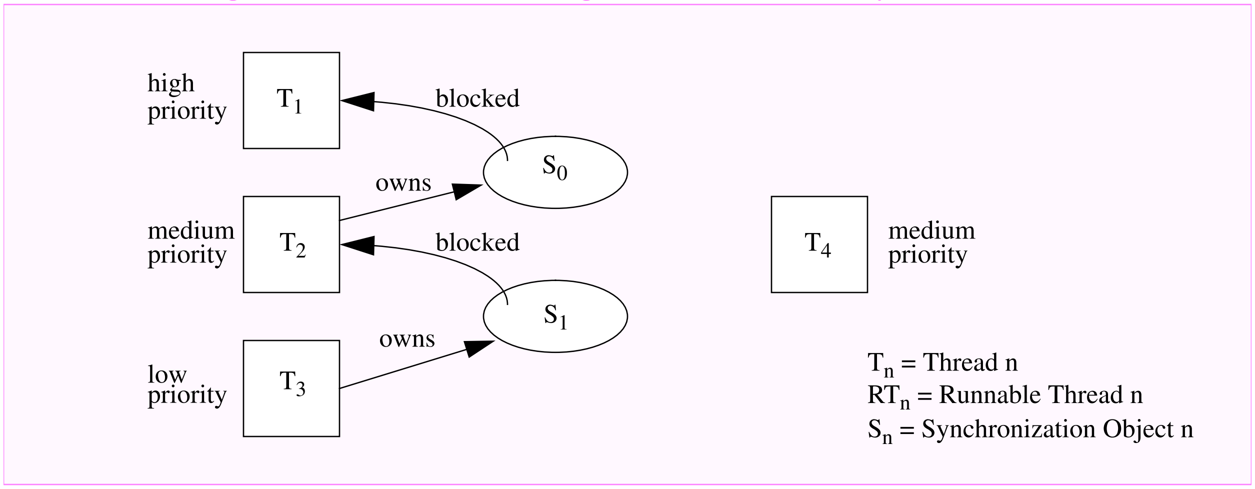
Priority inheritance determines at what priority threads block and are dispatched. Priority inheritance makes a distinction between the global priority and the *inherited priority* of a thread. The inherited priority is the priority a thread obtains via priority inheritance by blocking higher priority threads. The *dispatch priority* is computed as the maximum of the global and inherited priorities of a thread. A thread T executes at its global priority unless it is in a critical section and is blocking higher priority threads. If thread T blocks higher priority threads, T inherits a priority equal to the maximum dispatch priority of the threads it blocks. When thread T exits a critical section and releases the associated synchronization object, it relinquishes the inheritance it obtained by holding the synchronization object.

To ensure that the duration of priority inversion is bounded, the basic priority inheritance protocol requires that priority inheritance be transitive. That is, if there are three threads, T_1, T_2, and T_3, such that the dispatch priorities are ordered thus:

$$\text{Priority}(T_1) > \text{Priority}(T_2) > \text{Priority}(T_3)$$

and if thread T_3 blocks thread T_2, and thread T_2 blocks thread T_1, then thread T_3 inherits the priority of thread T_1 via thread T_2. Figure 2-6 below illustrates a case where transitive priority inheritance prevents unbounded blocking by the highest priority thread: if runnable thread T_3 does not inherit indirectly from blocked thread T_1, then the medium priority runnable thread T_4 will preempt the execution of T_3, resulting in unbounded blocking for T_1.

Figure 2-6: Indirect Blocking & Transitive Priority Inheritance



As described in the literature [Rajkumar 1988], the basic priority inheritance protocol imposes bounds on the duration of blocking. Briefly, if there are *m* lower priority threads, and these threads access *k* distinct synchronization objects in common with a higher priority thread, the higher priority thread can be blocked by at most a number of critical sections equal to the minimum of *m* and *k* .

2.5.2.1 Priority Inheritance Primitives

Table 3 summarizes the operations of priority inheritance. Priority inheritance for both mutexes and readers/writer locks is implemented using these primitives. The **pi_willto**() function is used when a thread attempts to acquire a synchronization object and discovers that it is blocked by another thread. The **pi_waive**() function is used when a thread releases ownership of a synchronization object and must surrender the inheritance that it received via the synchronization object. These two functions are the primary operations in our priority inheritance implementations. Note that the function **pi_willto**() contains no direct reference to the synchronization object that the argument thread is blocked on. This is because **pi_willto**() is called only after the argument thread has been put on the sleep queue for the synchronization object. Doing so causes information to be saved within the thread structure itself, allowing the synchronization object to be found via the thread.

Table 3: Priority Inheritance Primitives

Function Name	Operation
void pi_willto(*thread*)	Will the priority of the argument thread to all threads directly or indirectly blocking it.
void pi_waive(*thread, sync_object*)	Release the priority inheritance that the argument thread obtained via a particular synchronization object.

2.5.2.2 Priority Inheritance Implementation

Priority inheritance comes into play only when threads block on a synchronization object. To implement transitive priority inheritance, when starting from an arbitrary synchronization object, priority inheritance must be able to find the owning thread and each successive synchronization object and thread in the blocking chain. In order to make this information quick and easy to obtain, each synchronization object maintains a pointer to its owning thread; likewise, each blocked thread keeps a pointer to the synchronization object it is blocked on and a tag field to identify the synchronization object's type. Using this information, the priority inheritance mechanism in SunOS 5.0 can follow a blocking chain to the end. The end of the blocking chain, as far as priority inheritance is concerned, arrives when it finds a thread that is not blocked, or a synchronization object that is not priority inverted.

The distinction that priority inheritance makes between global priority and inherited priority is reflected in the implementation. The thread structure contains a field for each. Ordinarily, the field that represents the thread's inherited priority is zero. If the thread inherits a priority, the thread is marked to indicate this condition, and the inherited priority field is set accordingly. The code that enqueues and dequeues threads makes note of whether the thread possesses an inherited priority and uses it instead of the global priority when appropriate. Related to this issue, when priority inheritance encounters a thread that is in a sleep queue or dispatch queue, it must first dequeue the thread, will the new, inherited priority to the thread, then re-enqueue the thread at its new priority. Currently, in order to maintain the consistency of global data structures, all these operations must take place while holding **schedlock**, the global scheduler lock (§2.4).

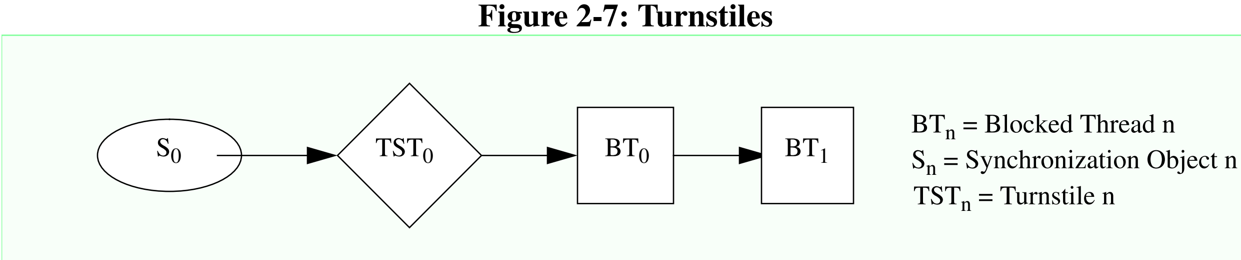
Note further that it is possible for a thread to hold several synchronization objects at once, each of which could be priority inverted at potentially different priorities. When priority inheritance wills a priority to a priority inverting thread, it saves a record of the synchronization object and the priority of the highest priority thread blocked on it in a circular linked-list in the thread structure of the priority inverting thread. When the priority inverting thread releases a synchronization object, the record of its inheritance from this synchronization object is removed from the thread's linked-list, and the linked-list is traversed to compute the thread's (possibly new) dispatch priority.

The sleep queue and all the priority inheritance information associated with a synchronization object are encapsulated in an abstract data type called a *turnstile*. Figure 2-7 below displays this relationship. Because there are many, many synchronization objects in the system, we allocate turnstiles dynamically when needed. When a thread blocks on a synchronization object that previously had no threads blocked on it, the newly blocked thread allocates a turnstile from a pool of turnstiles and attaches the turnstile to the synchronization object. The pool itself grows with the number of allocated threads in the kernel. The existence of a turnstile attached to a synchronization object is an indication that there are threads blocked on a synchronization object. When a thread releases a synchronization object that it owns and in so doing discovers that it has awakened the last of the threads in the turnstile's sleep queue, the releasing thread returns the turnstile to the free pool.

The traditional UNIX implementation of sleep queues uses a hashing scheme based on a *wait channel*, which is usually the address of a desired resource or some offset therefrom; aliasing of hash buckets is implicit in such a scheme [Leffler 1989]. High priority processes sleeping on unrelated wait channels may hash to the same sleep queue bucket; thus to wake up processes sleeping on a particular resource, it is necessary to hash to the appropriate sleep queue bucket, then traverse the sleep queue looking for those processes with matching wait channels. The result is that the behavior of insertion and release operations on sleep queues is not bounded by the number of processes competing for the resource—instead, the bound is the number of processes in the system.

Rather than using hashing schemes, turnstiles provide a per-synchronization object sleep queue, so aliasing is avoided. Insertion and release operations on turnstiles are bounded only by the number of threads competing for the associated synchronization object. Since threads waiting in a turnstile are queued in priority order, higher priority threads have a fixed bound for behavior. Turnstiles have potential to improve the performance of sleep and wakeup operations on large systems with lots of threads.

Figure 2-7: Turnstiles



2.5.2.3 Priority Inheritance for Readers/Writer Locks

The case of readers/writer locks deviates somewhat from the idealized picture we have presented for synchronization objects. When a synchronization object of this type is acquired by a thread with the intention of updating the data it protects, i.e., when a writer lock is in effect, there is no deviation: there is a single thread that controls the synchronization object and is the beneficiary of all inheritance applied via the synchronization object. The deviation comes in the case of readers locks. Readers locks can have a potentially large number of "owners." It is not practical in terms of space to keep a pointer in the readers lock to every thread that currently "owns" it. To simplify the implementation, for readers locks we have implemented what we term the *owner-of-record*. When a readers/writer lock is acquired for reading, the first thread that obtains the readers lock is assigned "ownership" of the synchronization object. As such, it is the beneficiary of all inheritance that passes through the readers lock until it releases the lock. When the owner-of-record thread releases a readers lock, it is possible that there are still other threads that own it. These owners are in a sense anonymous, since they cannot be identified by inspecting the readers lock, nor can they inherit from it while they own it. Since it is not an uncommon condition for a readers lock to have a single owner, our measurements indicate that there is still value in providing this limited form of priority inheritance for readers locks.

3.0 Lessons Learned

In the implementation of the realtime scheduling features, we encountered two problems: the requirements on mutex implementation and the generalization of priority inheritance.

3.1 Mutex Entry/Exit Implementation

Priority inheritance requires the owner of a synchronization object to be known. Our basic mutex entry code appears like:

```
ldstub   [MUTEX_PTR + M_LOCK], LOCK_REG
tst      LOCK_REG
bnz      lock_already_held
nop
st       OWNER_ID_REG, [MUTEX_PTR + M_OWNER]
```

An interrupt occurring between the `ldstub` and `st` instructions can leave the mutex held by an unknown owner.

One possible solution is to raise the processor interrupt level to prevent interruption during the above sequence. Since the management of the processor interrupt level is nontrivial, efficiency requires that uncontested entry/exit should not require raising the processor interrupt level. Another solution might be to use an atomic operation which could set the lock and owner fields simultaneously, but in the SPARC architecture [SPARC 1991], the sole atomic update available is the `ldstub` instruction. A third solution might be to have a blocking thread arrange for deferred inheritance, which the acquiring thread would have to check for and assume after return from interrupt. Again, this solution was rejected for its performance cost.

We chose to solve this issue by constraining the mutex lock operation to a stylized behavior which could be recognized by the interrupt handler. The use of a fixed register as LOCK_REG, with the convention that a zero value in this register indicated the successful acquisition of a mutex, allows the interrupt handler to safely set the mutex owner before processing the interrupt. Not only does this work in the uniprocessor case, where no other thread ever sees the lock held with an unknown owner, but it works in the multi-processor case because the interval between setting the lock and setting the owner is bounded, and a thread attempting to acquire the mutex from another processor can spin until the owner is known.

In order to prevent erroneous inheritance, possibly to a thread that is no longer extant, we chose to invalidate the owner field of a mutex at exit. This invalidation could not take place after the lock was released due to possibility of a race with a thread acquiring the mutex from another processor. It could not take place before the lock was released, due to the possibility of taking an interrupt between clearing the owner and the lock, and being unable to determine unambiguously whether the lock was held by the interrupted thread or some other. Hence we were forced to encode the owner and lock fields within the same word, and clear both with a single instruction.

3.2 Limitations in Providing Priority Inheritance

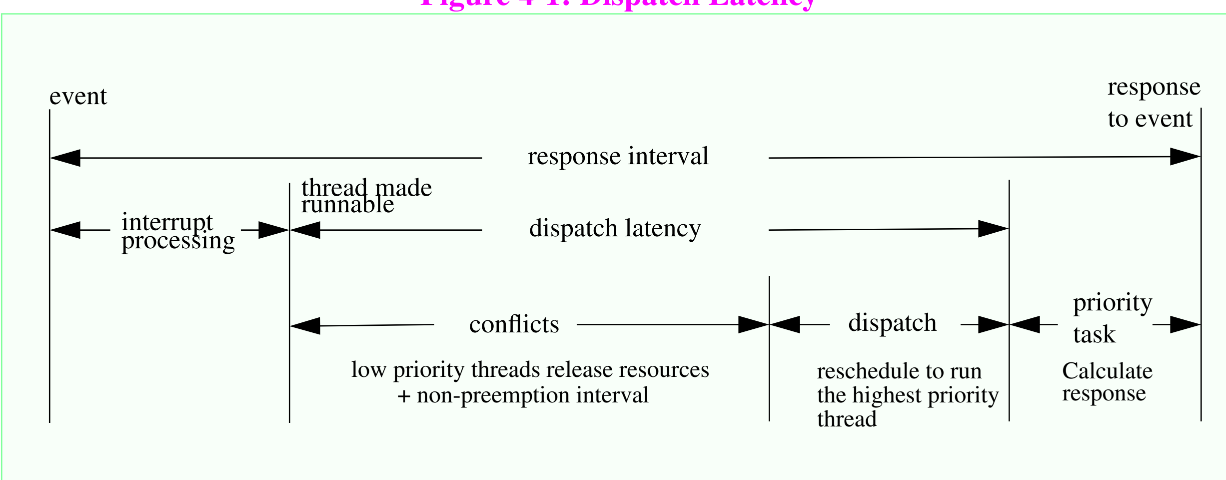
Priority inheritance is appropriate only when it is known which thread or threads must release the synchronization object in order for the blocking thread to proceed. We faced this issue for readers/writer lock, due to the cost of maintaining the list of reader access holders.

For condition variables and semaphores, priority inheritance is not possible. The protocol for these synchronization objects does not determine which thread(s) will release these objects. This is particularly unfortunate for condition variables, which can be used with mutexes to build higher order synchronization operations. Even if the protocol for these synchronization objects determines which threads(s) will be releasing them, the inability to inherit through condition variables precludes the provision of inheritance.

4.0 Performance Measurement

Scheduling performance from the realtime perspective can be defined in terms of the dispatch latency. We define dispatch latency as the amount of time it takes to begin execution of a high-priority runnable thread. As illustrated in Figure 4-1, this time includes the remainder of the non-preemption interval for the currently running thread, the time to resolve resource conflicts represented by acquiring held synchronization objects, and the context switch time. We do not include the processing time associated with interrupts nor any of the time within the application used in responding to the event which made the thread runnable. The total application response time will of course include both interrupt and application processing time.

Figure 4-1: Dispatch Latency



We examined two different ways of measuring dispatch latency. The first way is to measure all possible paths in the kernel to see if there exists a path which exceeds the response time we wish to guarantee. This requires a complete static analysis of the kernel, and the use of dynamically loaded modules again makes this solution infeasible. The second way is to repeatedly sample the dispatch latency of the kernel under specified loads. With a sufficiently large number of samples, this second way should have the same result. To measure the dispatch latency of SunOS 5.0, we have chosen the sampling approach.

Our sampling dispatch latency test (SDLT) is composed of a sampling process, using a sample driver and recording violations for further analysis.The sampling driver provides two **ioctl**() commands to control sampling. The first command computes and records the time for a wakeup event, blocks the sampling process, and scheduling the wakeup via realtime_timeout. When the wakeup callout function is called, it unblocks the sampling process, enables kernel execution path recording, and records the time of kernel wakeup. When the thread resumes execution, it records the time of kernel dispatch, and the first **ioctl**() then returns the recorded times to the sampling process. The sampling process then uses the second **ioctl**() command to disable kernel execution path recording and attain a timestamp for the return to user mode. The execution path recording has been extended to include timestamps to enable detailed analysis. When the sampling process detects a dispatch latency violating our projected guaranteed value, it captures the violation time parameters and execution path trace to a file.

We have measured our kernel with preemption both disabled and enabled. With preemption disabled, we have observed dispatch latencies in excess of 100 milliseconds. Our preliminary results with preemption enabled showed a dispatch latency of about 2 milliseconds on the SPARCstation 1, with a small number of larger values. Our analysis of these larger values pointed to long non-preemption intervals associated with the memory management unit hardware layer [Moran 1988]. We are in the process of reducing these intervals. A similar problem can arise when the context of the realtime process is stolen by another process. The operations associated with reloading the realtime process's context can take over 4 milliseconds on a SPARCstation 1, much of which is non-preemptible. We discuss a possible solution to this problem in § 5.0.

5.0 Future Work

Although we feel that we have achieved much in bounding the dispatch latency of SunOS, there remain a number of areas where we could improve the performance of the system, or increase its utility as a base for realtime applications. In this section, we discuss some of the areas we are interested in improving.

• Realtime I/O

In SunOS 5.0, much of the I/O processing is done through streams. The streams processing is done at the systems priority level and thus executes below any active realtime thread. It is impossible to guarantee realtime I/O for streams without drastically changing the handling of streams processing. Other I/O activities require additional changes in request queuing to gain realtime priority-based behavior.

• Dispatch Latency

We intend to continue trying to reduce the dispatch latency of the system. Work is going on to improve the granularity of the locking and to reduce the length of non-preemption intervals. It is our goal to bring the dispatch latency on the SPARCstation 2 down to 1 millisecond.

While we have been able to bound dispatch latency, applications are interested in the overall response time. One of the components of response time is the time required to process interrupts. Arrival of multiple interrupts may delay dispatch, thus making the duration of interrupt processing unbounded. We intend to provide the user a mechanism to block certain interrupts temporarily when running in a multiprocessor environment. Such a capability provides a means of shielding realtime threads from the delays due to interrupts.

• Locking Contexts

Since realtime threads are often event-driven, they typically do not run very often. On machines such as the SPARCstation 1 that implement the older, large kernel-managed TLB-based MMU, it is possible for a realtime thread to have its context stolen, even though its pages may be locked into primary memory. In order to resume the realtime thread, it may be necessary to steal a context from another thread and reload the realtime thread's context. To avoid the long non-preemption points associated with this activity, we are contemplating implementing a facility to lock a context into association with a thread, providing a facility analogous to that for locking pages into memory.

6.0 Summary

In SunOS 5.0, we have provided the following realtime functionality:

- Static priority and fixed quantum scheduling for realtime threads.
- A fully preemptive kernel, providing bounded dispatch latency.
- Hidden scheduling in the kernel has been greatly reduced, eliminating much priority inversion.
- Priority inversion arising from the use of synchronization objects has been controlled by implementing the basic priority inheritance protocol.

7.0 Bibliography

[AT&T 1989] *System V Interface Definition, 3rd Edition*, AT&T 1989.

[AT&T 1990] *UNIX System V Release 4 Internals Student Guide*, AT&T 1990.

[Lampson 1980] B. W. Lampson and D. D. Redell, "Experiences with processes and monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105-117, February 1980.

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

[Moran 1988] Joseph P. Moran, "SunOS Virtual Memory Implementation," *Proceedings for the Spring 1988 EUUG Conference*, EUUG, London, England, Spring 1988.

[Rajkumar 1988] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the Real-Time Systems Symposium*, December 6-8, 1988, Huntsville, Alabama.

[Powell 1991] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, "SunOS Multi-thread Architecture," *Proceedings 1991 USENIX Winter Conference*.

[Sha 1990] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

[SPARC 1991] *The SPARC Architecture Manual, Version 8*, Prentice-Hall 1991.

[SunSoft MT, to appear] "Multithreading Techniques Used in the SunOS 5.0 Kernel," to be submitted to a future conference.

8.0 Acknowledgments

The authors are indebted to the multithreading group of the Operating Systems Technology Group of SunSoft, both for providing the platform multi-thread implementation and for their assistance in development of the realtime scheduling mechanisms. We also wish to acknowledge the assistance of Deepak Dhamdhere for his contributions to the Performance section of this paper. He built many of the measurement tools and collected the data that provided a basis for that section.

9.0 Correspondence

Correspondence regarding this paper should be sent to the address below.

Michael Sebrée
SunSoft Incorporated, M/S MTV5-44
2550 Garcia Avenue
Mountain View, CA 94043
Michael.Sebree@Eng.Sun.COM

10.0 Biographies

Sandeep Khanna is a software engineer in the Operating Systems Technology Group at SunSoft. He graduated with an M.S. in Computer Science from the University of Mississippi in 1987. He received his B.E. Electrical and Electronics Engineering from Birla Institute of Technology and Science, Pilani, India in 1984. He is currently involved with the design and implementation of realtime extensions to SunOS 5.0.

Michael Sebrée is a software engineer in the Operating Systems Technology Group at SunSoft. He is rumored to have obtained some knowledge of operating systems in general, and the UNIX operating system in particular, at Vanderbilt University and at UCLA.

John Zolnowsky is a software engineer in the Operating Systems Technology Group at SunSoft. He has a Ph.D. in Computer Science from Stanford University, was instruction set architect for the 68000 processor family, and now serves as technical editor for the POSIX 1003.4 Realtime Extensions.