Adding Scheduler Activations to Mach 3.0 *

Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

Technical Report 92-08-03
August 1992
Revised March 1993

Abstract

When user-level threads are built on top of traditional kernel threads, they can exhibit poor performance or even incorrect behavior in the face of blocking kernel operations such as I/O, page faults, and processor preemption. This problem can be solved by building user-level threads on top of a new kernel entity, the *scheduler activation*. The goal of the effort described in this paper was to implement scheduler activations in the Mach 3.0 operating system. We describe the design decisions made, the kernel modifications required, and our additions to the CThreads thread library to take advantage of the new kernel structure. We also isolate the performance costs incurred due to scheduler activations support, and empirically demonstrate that these costs are outweighed by the benefits of this approach.

1 Introduction

User-level threads built on top of traditional kernel threads offer excellent performance for common operations such as creation, termination, and synchronization. Unfortunately, though, they exhibit poor performance or even incorrect behavior in the face of blocking kernel operations such as I/O, page faults, and processor preemption. This has presented the application programmer with a dilemma: use either kernel threads, which are well integrated with system services but have expensive common-case operations, or user-level threads, which have efficient common-case operations but are poorly integrated with system services.

To resolve this dilemma, Anderson et al. [ABLL92] designed a new kernel entity, the *scheduler activation*, which provides proper support for user-level thread management. User-level threads built on top of scheduler activations combine the functionality of kernel threads with the performance and flexibility of user-level threads. In the past few years a consensus has emerged that scheduler activations are the right kernel mechanism for supporting the user-level management of parallelism.

The goal of the effort described in this paper was to implement scheduler activations in the Mach 3.0 operating system [RBF+89]. This paper places emphasis on the implementation rather than the concepts involved. We describe the design decisions made, the kernel modifications required, and our additions to the CThreads thread library to take advantage of the new kernel structure.

*This work was supported in part by the National Science Foundation under Grants No. CCR-9200832, CDA-9123308, CCR-8907666, CCR-8703049, and CCR-8619663, by Digital Equipment Corporation, and by the Washington Technology Center. The authors' email addresses are {pauld, dylan, raj, lazowska}@cs.washington.edu

1.1 User-Level Threads

When expressing parallelism, the less expensive the unit of parallelism, the finer the grain of parallel work that can be supported with tolerable overhead. It has long been accepted that kernel processes (e.g., as in Unix) are too expensive to support any but the most coarse grain parallel applications. Multiprocessor operating systems such as Mach and Topaz [TSS88] attempted to address this problem by providing kernel threads. But kernel threads have also proven to be too expensive, and Mach and Topaz now provide user-level thread systems built on top of their kernel threads. Anderson et al. [ABLL92] argue that the cost of kernel threads relative to user-level threads is not an artifact of existing implementations, but rather is inherent, arising from two factors:

- *The cost of kernel services.* Trapping into the kernel is more expensive than a simple procedure call. Architectural trends are making kernel traps relatively more expensive [ALBL91]. In addition, to protect itself from misbehaving user-level programs, the kernel must check each argument for bad values that could cause the system to crash. A user-level thread package uses procedure calls to provide thread management operations, avoiding the overhead of kernel traps. Furthermore, the package doesn't necessarily have to bulletproof itself since a crash will only affect the misbehaving program.

- *The cost of generality.* Kernel threads must be all things to all people. Even if a particular program has no need for some specific facility (e.g., round-robin scheduling), it still must pay the price for the existence of that facility. A user-level thread package can be customized and optimized for each application's needs [BLL88].

User-level threads have proven useful for expressing relatively fine-grain parallelism. Unfortunately, implementing user-level threads on top of existing operating system mechanisms (such as kernel threads) causes difficulties. The first problem is that the kernel threads are scheduled obliviously with respect to user-level activity; the kernel scheduler and the user-level scheduler can interfere with each other. The second problem is that blocking kernel events such as I/O, page faults, and processor preemption are invisible to the user-level. Various mechanisms have been suggested to address specific instances of these problems [MSLM91] [TG89] [Her91], but none of this work addresses all of the difficulties. Scheduler activations present a unified solution.

1.2 Scheduler Activations

Scheduler activations are an alternative to kernel threads for supporting the user-level management of parallelism. As presented in [ABLL92], a scheduler activations environment has the following key characteristics:

- *Processors are allocated to jobs by the kernel.* Processor allocation is managed by the kernel, based on information communicated from user-level. The kernel allocates a processor by providing the job with a *scheduler activation*, an entity much like a traditional kernel thread, but with additional properties noted below.

- *A user-level thread scheduler controls which threads run on a job's allocated processors.* Kernel-provided scheduler activations are simply vessels upon which the user-level scheduler multiplexes threads. Common operations such as thread scheduling and synchronization are performed efficiently at user-level, without kernel intervention.

- *The user-level notifies the kernel of changing demand for processors.* The kernel is notified when the job's parallelism crosses above or below its current processor allocation.

- *The kernel notifies the user-level scheduler of system events that affect the job.* These events include the allocation or preemption of a processor, and the blocking or awakening of an activation in the kernel. The kernel's role changes from *handling events* to being responsible for *communicating events* to the appropriate job's user-level thread management system. This allows the thread system to respond to the event in a manner most appropriate for the job.

- *The kernel never time-slices scheduler activations.* Scheduler activations are the means by which the kernel provides processors to jobs. A job always has exactly as many scheduler activations as it has physical processors. The kernel never multiplexes (a given number of) scheduler activations on (a smaller number of) physical processors.

- *Application programs can remain unmodified, yet take advantage of this new environment.* This is because event management is encapsulated within the user-level thread system, the interface to which remains unchanged.

Readers unfamiliar with this system structure are strongly encouraged to review [ABLL92] before proceeding.

1.3 Scheduler Activations in Mach

Anderson's prototype implementation involved modifying the Topaz operating system on the DEC SRC Firefly multiprocessor workstation [TSS88]. This provided an excellent prototyping environment, but the implementation was inaccessible to others, and experience was limited to systems with at most six processors.

Our goal was to integrate scheduler activations into a kernel that was widely used and that ran on a variety of platforms. We chose Mach 3.0, and conducted our work on a 20-processor Sequent Symmetry [LT88]. We envision that our implementation will fulfill several roles. First, it will allow final validation of the scheduler activations concept with a reasonable number of processors. Second, it will provide a base on which to conduct further investigations into processor allocation algorithms, thread scheduling algorithms, the integration of user-level threads with efficient communication primitives, the provision of services such as I/O and virtual memory management at user-level, and so forth. Third, it will serve as a model for those who wish to integrate scheduler activations with Mach on other platforms.

In undertaking this implementation, we made choices that favored expediency, Mach compatibility, and machine-independence. Toward the goal of expediency, we wanted to minimize modifications to the system, using existing kernel mechanisms whenever possible. To make the resulting system widely usable, we strove for backward compatibility with Mach; for example, we continue to support the existing kernel threads interface. Finally, we made the implementation as machine-independent as possible, so that it could easily be ported to other platforms. We sought to create a testbed for experimentation, rather than an entirely restructured system based upon scheduler activations. Maximal efficiency was not a goal of our implementation. Rather, we expect that experience with a production system will identify areas where optimizations will be most beneficial. We hoped that this approach would make our implementation more comprehensible and modifiable, as well.

1.4 Design Framework

In the scheduler activations model, each job's user-level thread system has control over scheduling on the processors allocated to it, and is notified if this allocation changes. The kernel's responsibility is to provide a scheduler activation (execution context) per processor allocated to the job, and to notify the job of any relevant events.

We implemented scheduler activations by modifying the behavior of Mach kernel threads. One key design decision that affects the structure of our implementation is the introduction of a processor allocation module,

which in some sense replaces Mach's existing kernel thread scheduler. Mach's kernel thread scheduling policy is thread-based and quantum-driven. This policy is adequate for traditional workloads, but can be outperformed by a policy that takes advantage of the information conveyed between scheduler activations applications and the kernel.

The primary requirement of such a policy is the ability to manage processor allocation on a per-task basis, something difficult to do under the basic Mach policy. However, Mach's *processor sets* [Bla90] provided us with a mechanism to circumvent the standard thread-based policy (which is inappropriate for our purposes), and to replace it with our own task-based one. Processor sets are kernel entities to which tasks, threads and processors are assigned; threads execute only on processors assigned to their corresponding processor set.

Our design gives each task its own processor set. A policy module monitors the tasks' varying processor demands, basing its allocation decisions on these demands and on its own calculations of inter-task priorities. This new policy module could be implemented either in the kernel or in a user-level server. We chose the latter because it afforded us the flexibility to easily experiment with various policies. The final policy we chose (described later) is encapsulated in a user-level "processor allocation server", which uses existing kernel mechanisms to enforce its decisions: processors are allocated to a task by assigning them to the task's associated processor set, and are preempted by removing them from that set. As this design implies, our system consists of three components:

- A set of kernel modifications implementing basic mechanisms, such as event notification. These are discussed in the next section.
- A user-level server that manages processor allocation between tasks that are using scheduler activations. The policy used by (and the implementation of) this server is explained in Section 3.
- A user-level threads package that takes advantage of the new kernel, described in Section 4.

2 Kernel Support for Scheduler Activations

This section explains the modifications we made to the Mach 3.0 kernel[†] (MK78) to support scheduler activations. From the perspective of the kernel, a scheduler activation is an ordinary Mach kernel thread, with the additional property that events caused by (or affecting) a task are reflected up to the user-level.

2.1 Initialization

Our modified Mach kernel supports both traditional Mach tasks (those desiring Mach kernel threads) and tasks desiring to use scheduler activations. A task informs the kernel of its desire to use scheduler activations by executing the following new system calls:

- `task_register_upcalls` (*task, upcalls*). Registers a set of entry points in user space: the procedure addresses of the user-level upcall handling routines.
- `task_recycle_stacks` (*task, count, stacks*). Provides the kernel with *count* blocks of user-space memory. When the kernel performs an upcall, it consumes one of these blocks for the user-level execution stack of the scheduler activation performing the upcall. It is necessary for the task to manage these stacks because the kernel is unable to detect when the information on a particular stack is no longer useful to the task. The task must ensure that stacks are always available for the kernel's use during upcalls (see Section 4.4.2).

[†]No modifications to the UX server were required.

- `task_use_scheduler_activations(`*task*`, TRUE)`. Sets a bit in the kernel's data structure for the task, marking the task as using scheduler activations. If a task has this bit – the `using_sa` bit – set, then any kernel threads created within that task will have a (related) bit set. This bit – the `is_sa` bit – signifies that the kernel thread is in fact a scheduler activation. The kernel uses these bits to decide how to handle particular events, as described below.

  As a side effect of this call, the calling kernel thread (assumed to be the only one extant in the task) is converted into a scheduler activation. On return from this call, the task may proceed with its normal computation under scheduler activations semantics.

We control the task's state using multiple system calls in the interest of flexibility. Although this method results in slightly higher overhead at initialization time, this one-time cost seemed an acceptable price for allowing the task finer control over its state (allowing it to change its registered upcall handlers at any time, for example).

2.2 Handling Kernel Events

In the scheduler activations model, certain kernel events dictate that a notification be sent to the task. Notifications are implemented as upcalls from the kernel to user-level, with arguments as shown below. There are four notifications that can be sent from the kernel to the user-level:

- `blocked(`*new_sa, event_sa, interrupted_sas*`)`: *new_sa* is carrying the notification that *event_sa* has blocked. *interrupted_sas* refers to activations that may have been interrupted in order to deliver the notification;[†] as described in Section 2.2.1, this argument is always `NULL` in the case of `blocked` notifications.

- `unblocked(`*new_sa, event_sa, interrupted_sas*`)`: *new_sa* is carrying the notification that *event_sa* has unblocked. *interrupted_sas* refers to any activations that may have been preempted to provide the notification.

- `preempted(`*new_sa, event_sa, interrupted_sas*`)`: *new_sa* is carrying the notification that *event_sa* was preempted due to a processor reallocation. *interrupted_sas* refers to any activations that may have been interrupted to provide the notification.

- `processor_added(`*new_sa, event_sa, interrupted_sas*`)`: *new_sa* is carrying the notification that a new processor has been allocated to this task. In this notification, both *event_sa* and *interrupted_sas* are `NULL` (see Section 2.2.4).

The implementation of these four notifications is described in the next four subsections; we first provide some necessary background. Be forewarned that the remainder of Section 2.2 is fairly detailed.

A crucial component of a scheduler activation's state is its *user-level continuation*, which is similar to a kernel continuation (as described in [DBRD91]), but specifies a kernel routine to be run instead of returning to the user-level. We set this when we wish to gain control of the activation's execution in order to force an upcall to user-level (see Section 2.2.1).

A scheduler activation also has two fields (*event_sa* and *interrupted_sas*) that point at other activations, allowing it to carry information about them to user-level during a notification. This information includes the identities of the activations, the processor on which they were running, and their user-level register state.

The kernel uses the `thread_select()` routine to choose which thread or scheduler activation to run next. We maintain a flag per processor that tells `thread_select()` whether a notification is needed. The values of this flag include `PROC_SA_BLOCKED` for blocked events, `PROC_SA_HANDLING_EVENT` during notification, and

[†]This field is called *interrupted_sas* (plural) because an activation may block in the kernel while preparing a notification. In this case, several activations may be interrupted before the notification can finally be delivered.

PROC_ALLOCATED for processor reallocation between processor sets. When thread_select() notices that a processor's event flag has one of these values, it arranges to send a notification.

The next sections describe the handling of events in general terms; a more detailed description is in Appendix A.3.

2.2.1 Blocking

When an activation blocks, its event_sa is set to point to the activation itself, and its user-level continuation is set to the kernel routine sa_notify() (these are used when it unblocks; see Section 2.2.2). The processor's event flag is set to PROC_SA_ BLOCKED. This causes thread_select() to create a new scheduler activation to carry the notification (instead of choosing some other activation to run). The new activation's *event_sa* is set to point to the blocked activation. The new activation starts at sa_notify(), which fetches a user-level stack on which to place the arguments for the upcall: the current activation and the blocked activation.

The user-level stack is manually set up to look like a trap return context, with the program counter set to be the task's upcall entry point, sa_blocked. Finally, the notifying activation uses thread_exception_return() to begin executing in user space.

2.2.2 Unblocking

An unblocked event is started by the kernel's normal procedure for making an activation runnable after a blocking event (such as a disk read). One of the currently running activations is interrupted in order to run the unblocked activation; the policy driving this decision is described in Section 2.3.3. The interrupted activation calls thread_block() to yield the processor. thread_block() notices that the activation is preempting itself to pick up another runnable activation in the same task[†], and sets the processor's event flag to PROC_SA_HANDLING_EVENT. Next, thread_select() finds the runnable unblocked activation, and because PROC_SA_HANDLING_EVENT is set, adds the interrupted activation to the unblocked activation's interrupted_sas list.

The kernel switches to the unblocked activation to allow it to clean up its kernel state (for example to call pmap_enter() after a page fault I/O is complete). This activation attempts to return to user-level by calling return_from_trap(), which checks to see if the activation has a user-level continuation. Since it does (it was set when the activation blocked), it calls the kernel routine sa_notify() instead of returning to the user-level. This routine uses the unblocked activation itself to perform the notification, fetching a user-level stack and pushing the arguments to the notification: the current (unblocked) activation, the event_sa (set when the activation blocked to point to itself) with its user state, and any interrupted_sas with their user state. The user-level continuation is cleared, and the stack is set up to arrange return to the user-level sa_unblocked handler. Finally, thread_exception_return() is used to return to user-level.

2.2.3 Processor Preemption

Preemption notifications occur only when processors are reallocated from one task to another (by the processor allocator). The task losing the processor receives a preempted notification, as described below. The task gaining the processor receives a processor_added notification, which is described in the next section.

The kernel is told by the processor allocator (via processor_assign()) to reassign a processor. If a scheduler activation is preempted as a result of the reallocation, we send a notification as follows. We create a new scheduler activation and start it at sa_notify(), setting its event_sa to be the preempted activation. The new activation is placed in the run queue of the task losing the processor; it is dispatched to user-level as with an

[†] As opposed to an activation preempted because its processor has been reallocated to another task.

unblocked notification, except that it begins user-level execution at a different entry point, the sa_preempted handler.

Asynchronously to this activity, the preempted processor's event flag is set to PROC_ALLOCATED, and the processor is moved from the old processor set to the new one. If the task in the new processor set is using scheduler activations, it is notified as described below.

Notice that if a task loses its last processor, the notification of this event will not occur until the task again receives processors. It may be reasonable to suppress notification (rather than simply defer it) when a task has lost all of its processors – the last running activation could be resumed at its point of interruption, making the last preemption transparent to the task. There are cases where this is inappropriate: for example, a task managing cache affinity at user-level may require knowledge as to exactly which (as opposed to merely how many) processors it holds. We therefore chose to create notifications in all cases, allowing the task's thread scheduler to make its own decisions when the task is reactivated. While this is in keeping with the scheduler activations philosophy, tasks should perhaps be able to control this feature; this would allow them to avoid the (albeit small) user-level scheduling overhead when they feel it is unnecessary.

2.2.4 Processor Allocation

There are two methods of notifying a task of the allocation of a new processor, corresponding to whether or not the task has any runnable activations when the processor arrives. If there are runnable activations (they would represent undelivered preempted or unblocked notifications) they are simply allowed to return to the user-level – on the new processor – as described above. The user-level thread system knows it has gained a processor because the notification does not refer to an interrupted activation, which is only possible if a new processor is carrying the notification.

If there are no runnable activations in the task, the kernel must explicitly create a processor_added notification. thread_select() tries and fails to find a ready activation to run. Before running the idle thread (as it would normally do), it checks the processor's event flag, which was set to PROC_ALLOCATED when the processor assignment occurred. In this case, it creates a new scheduler activation to carry the processor_added notification. The new activation, running in sa_notify(), simply fetches an upcall stack, pushing a pointer to itself for the new_sa argument but pushing NULL's for event_sa and interrupted_sas. The user-level entry point is set to be the sa_processor_added handler. The notification is sent to user-level via thread_exception_return().

2.3 Implementation Details

The preceding has described the general operation of kernel event management. We now discuss in more detail some of the issues involved in its implementation.

2.3.1 Passing Scheduler Activation State

As described above, the register state of scheduler activations is passed from the kernel to the user-level thread management system. In our current implementation, this state includes all registers saved by the normal trap handler when the activation made the user- to kernel-mode transition. However, Mach's 80386 trap handler is careful to save the (large) state of the processor's floating point unit (FPU) only if this unit was in use when the trap occurred. This leads to an obvious optimization in our notification path: FPU state is copied/passed only if this state had originally been saved by the trap handler.

2.3.2 Scheduler Activation Recycling

A new scheduler activation is created whenever an event occurs. To make this process efficient, the kernel maintains in each task a pool of kernel threads that can be quickly converted to scheduler activations. A tunable number of threads is initially inserted into this pool when the task informs the kernel that it will be using scheduler activations. New kernel threads – to serve as scheduler activations – are created only when the pool is empty. The kernel "recycles" activations automatically. A scheduler activation may be recycled whenever the kernel is sure that it holds no state that will be needed subsequently. More precisely, when notifying about an event, the kernel determines whether or not the associated activations may be recycled, as follows:

- `blocked`: A blocked activation cannot be recycled because it will have to run in the kernel when it unblocks.
- `unblocked`: The unblocked activation is used to perform the notification. This means that unblocked activations may not be recycled. If an activation was interrupted to deliver the notification, its user state is copied out to user-level, and it never runs again, so it can be recycled. (This is true of all interrupted activations.)
- `preempted`: A preempted activation's state is copied out to user-level, and it will never run again, so it may be recycled. As explained above, we can recycle any interrupted activations associated with the notification.
- `processor_added`: There is no *event_sa* for `processor_added` events, and no *interrupted_sas* because the notification corresponds to a new processor's arrival; therefore, nothing is available for recycling in this case.

2.3.3 Preemption Policy and Mechanism

A scheduler activation is the abstraction of a physical processor, and as such, activations are never time-sliced on physical processors by the kernel. Therefore, although scheduler activations are strongly based on kernel threads, the complicated scheduling machinery already in place for threads (priority manipulation, and so on) is irrelevant with respect to activations. In fact, it would be erroneous to use this machinery, because quantum-driven preemption within a scheduler activations task is explicitly to be avoided.

We modified the kernel so that priorities are ignored for tasks (processor sets) using scheduler activations. Preemption occurs only when there is a runnable but not running scheduler activation in the task. The kernel ensures that this only occurs as a result of particular events:

- a previously blocked activation unblocks
- a processor is reallocated from this task to another, and a notification is necessary (see handling of preemption, described above)

Standard mechanisms are used both to check for the preemption condition, and to actually do the preemption when appropriate: the normal periodic checks in the former case, and the Asynchronous Software Trap (AST) mechanism in the latter. This implies that scheduler activations determine autonomously whether they should preempt themselves; the policy outlined above dictates that they do this only if there is a notification (represented by a runnable scheduler activation) to "pick up". This met our goal of modifying the existing system as little as possible, and kept the advantage that scheduling decisions are made in a distributed fashion (in contrast to a possible scheme where the kernel chooses a particular activation to preempt, sets an AST for it, and forces it to pick up the notification).

Normally, the kernel uses priority to choose which activation to preempt. Our modified kernel, when dealing with scheduler activations, keeps track of which scheduler activations are not doing useful user-level work, and thus are candidates for preemption. We use an extra bit in the kernel `thread` structure, *willing_to_yield*, which indicates whether or not the associated scheduler activation is "willing to yield" the processor. Since the kernel cannot determine this without help from the task, this bit is controlled using a new system call, `thread_willing_to_yield()`. If called with `TRUE`, the calling activation's *willing_to_yield* bit is set to `TRUE`. Also, a per-processor-set count (*willing_count*) is incremented, indicating that one more scheduler activation is in this state. If called with `FALSE`, or if the "willing" activation blocks for any reason, the activation's *willing_to_yield* bit is set to `FALSE`, and the processor set's *willing_count* is decremented.

This notion of "willing to yield" is used to determine which scheduler activation should pick up a pending notification. In unmodified Mach, a kernel thread sets an AST for itself if it finds a higher-priority thread in its processor set. In our scheme, scheduler activations ignore priorities, preempting themselves to pick up notifications in the following way:

- If the number of runnable activations in the processor set exceeds the number of activations "willing to yield", then the activation finding this to be true sets an AST for itself. (That is, that activation allows itself to be preempted.)

- Otherwise, the activation checks whether it is "willing to yield". If so, it sets the AST, preempting itself. The *willing_count* is not decremented until the preemption is complete; this ensures that other activations do not unnecessarily preempt themselves because the *willing_count* appears too low (while the preemption is in progress). If *runnable_count* $<=$ *willing_count* but the current activation is not "willing to yield", it does not preempt itself. Instead, it simply assumes that either a) preemptions to pick up the runnable activations are already in progress, or b) other "willing" activations will eventually[†] notice that a preemption is required, and initiate one themselves. If one (or more) of the willing activations becomes busy before noticing the need for preemption, then the next busy activation to check the preemption condition will find *runnable_count* $>$ *willing_count*, and preempt itself.

The scheme described here introduces some latency in notification. Some of this latency is inherent in the existing system: noticing the need for preemption has some delay, and performing the preemption requires some time. We felt that better decisions could be made using the current distributed decision-making scheme than by having the kernel choose specific activations to preempt (in the latter case, the activation's status could change after the preemption request was set but before the activation noticed it, resulting in the preemption of a usefully busy activation rather than another, potentially idle, activation). Extra latency is incurred when there are the same number of "willing to yield" activations as runnable activations, but some of them become busy before all of the notifications have been picked up. The extra latency in this uncommon case was deemed an acceptable price for ensuring the proper activation is preempted.

2.3.4 Kernel Threads vs. Scheduler Activations

Scheduler activations can masquerade as kernel threads by turning off their *is_sa* flag. A scheduler activation thus disguised is allowed to block without causing a notification to be sent. This is done for brief periods in the kernel to prevent notifications from being posted at inopportune times. Consider what would happen if an activation page faults in the kernel during a `blocked` notification. If its *is_sa* bit was on, this would be a new event, which would cause another `blocked` notification, which could cause another fault, and so on. We prevent this by turning the *is_sa* bit off until (just before) the activation reaches user space.

[†]The next time they check for ASTs.

We currently restrict tasks that are using scheduler activations to use them exclusively[†]. This is not a necessity, but rather a decision made to simplify the implementation. The primary reason for this constraint is the difficulty of integrating scheduling policies for scheduler activations and kernel threads.

In order to allow suspension of scheduler activations tasks, we have had to make some of the task manipulation code aware of scheduler activations use. Without any special action, each activation would cause a notification as it suspended, and the notification would continue running. We wanted to support suspension to make scheduler activations tasks backward compatible with Mach tasks, and because task suspension is a normal part of task termination. The task manipulation code has been modified to toggle the `task->using_sa` flag appropriately. `task_hold()` turns off `task->using_sa` (saving the value of the bit) so we can suspend activations without notification. `task_release()` resets `task->using_sa` from the saved value. `task_terminate()` turns off the bit permanently.

A related area, one in which we have made a temporary but serious compromise, is that of exception handling. In our current system, `thread_raise_exception()` turns off `task->using_sa` for the task containing the offending activation. That is, if an activation takes an exception, its task is no longer using scheduler activations; it is the exception server's responsibility to turn the bit back on if the exception proves non-fatal. This is a decision driven by the vagaries of our current implementation environment; it should not be construed to be an assertion that these are the correct exception semantics. We expect to produce a more reasonable solution once these issues are resolved.

3 Processor Allocation Server

The processor allocation policy for tasks using scheduler activations is controlled by a user-level server rather than by the kernel. Each task has its own processor set, to which all of its activations belong and to which processors are assigned by the processor allocation server. This allocator assigns processors to tasks according to the Dynamic policy of McCann et al. [MVZ90].

The Dynamic policy is an adaptive, space sharing policy that attempts to maintain an equal allocation of processors to tasks.[‡] In a system with P processors and T tasks, each task is initially assigned its desired allocation, up to P/T. Subsequently, the allocation may be modified in response to changes in the tasks' instantaneous processor demands: tasks often have varying parallelism, and therefore often experience periods during which their current allocation is either too high or too low.

To realize the fundamental mechanism of the policy – dynamically moving processors from tasks that have too many to tasks that have too few – some communication is required between tasks and the allocator. An application indicates that it has more processors than it can currently use via a per-processor "willing to yield" hint. Unscrupulous programmers could possibly misrepresent their needs to increase their application's performance at the expense of overall system performance. In order to discourage this behavior, the dynamic policy includes an adaptive priority mechanism.

Task priorities are assigned using a scheme that raises them as a reward for using few processors and lowers them as a penalty for using many. A task thus acquires "credit" during periods of low processor demand, which it may later "spend" to acquire (temporarily) more than its fair share of processors. A task's credit is equal to the mean number of processors above or below the equipartition allocation that it has used during its lifetime. By setting priorities in a way that penalizes tasks that are greedy beyond their true needs, the policy discourages malicious behavior.

[†]One possible use for combining kernel threads with scheduler activations in the same task is as a debugging tool. For example, while testing upcall handlers, it may be useful to prevent them from generating additional upcalls.

[‡]Policies based on "spatial" equipartition have been shown to outperform time-sharing policies in this environment [TG89] [MVZ90].

We implement this processor allocation policy using the following per-task information (maintained in a page of memory shared between the task and the allocator):

- *desired* stores the number of processors each task has requested
- *count* stores the number of processors a task has at any moment
- *fair_share* stores the allocation they would receive under equipartition
- *credit* is the weighted average over time of *fair_share* − *count*
- *priority* is calculated by discretizing and sorting by decreasing amounts of credit

The allocator uses this information to make allocation decisions, as follows. First, free processors are given to tasks whose *count* < *desired*, in decreasing priority order. Then any processors marked "willing to yield" are assigned similarly. Processors are taken from tasks with low priority and given to tasks of high priority until either all tasks remaining are of equal priority, or all tasks have as many processors as they desire. Any remaining tasks (all of the same priority now) share processors according to equipartition.

As previously noted, implementing the processor allocator as a user-level server gave us the flexibility to experiment with different allocation policies. We have in fact implemented several policies in addition to the one just described; we expect that still others will be implemented and evaluated as our work in this area continues.

4 User-Level Threads

CThreads [CD88] forms the basis of our user-level thread package. We chose CThreads because it already works with the Mach kernel and has been widely used in the Mach community. CThreads is implemented as a library of functions with which the application program is linked.

The interface exported by our modified CThreads library is the same as that of standard CThreads[†]. The following sections describe how we have changed the CThreads package to take advantage of the new kernel and present some details of the implementation.

4.1 CThreads with Scheduler Activations

The "Mach Thread" implementation of CThreads [CD88] runs user-level threads on top of Mach kernel threads – when a program's parallelism increases, it asks the kernel for more kernel threads, which are used to run user-level threads. Our CThreads implementation continues to multiplex user-level threads onto single kernel entities (scheduler activations instead of kernel threads), but changes the unit of resource requests from threads to processors. When we ask for a processor, what we actually receive is a new scheduler activation, executing our `sa_processor_added()` handler on the newly-allocated processor.. Thus, the kernel is responsible for providing scheduler activations to run on the processor, and the thread package assumes responsibility for deciding what they should do. In summary, our model involves:

- user-level threads
- multiplexing of user-level threads onto scheduler activations, using conventional techniques
- a number of scheduler activations, created by the kernel and guaranteed to be the same as the number of currently allocated processors

[†]Except that `cthread_wire()` and `cthread_unwire()`, which bind and unbind user-level threads to particular kernel threads, no longer have meaning when running with scheduler activations.

4.2 User-Level Components

In order to adapt a user-level thread package to a scheduler activations kernel, a number of components must be added. These include a new initialization phase, new actions at thread creation time, and code to handle upcalls from the kernel. This section details the operation of these components.

At initialization time, our startup code hands to the kernel the addresses of the handlers for the four types of notifications. It then creates a set of user-level threads that will be used to execute these handlers (via the upcall mechanism described in Section 2.2), and marks them as being upcall-handling threads by setting their `is_upcall` bit. It passes the addresses of these threads' stacks to the kernel (via `task_recycle_stacks()`). Next, it contacts the processor allocation server, which creates a processor set for the task and allocates a processor on which the task can run. Finally, the current kernel thread tells the kernel that the task is using scheduler activations, becoming a scheduler activation as a result.

Whenever new user-level work is created, the thread system decides whether to ask for more processors. If the processor allocator gives the task another processor, the kernel ensures that the task will eventually receive a `processor_added` notification; the scheduler activation carrying the notification is then used to execute user-level work on the new processor.

When notified that a scheduler activation has blocked in the kernel, the thread system determines which (associated) user-level thread has blocked, and marks it as such. The next ready thread is dequeued and run.

When notified that a scheduler activation has unblocked in the kernel, the thread system determines which (associated) user-level thread has unblocked, and marks it as such. It then attempts to deal with threads holding locks (as explained in subsequent sections). Next, any threads found to be runnable are enqueued in a ready queue; finally, the next ready thread is run.

If a processor gets taken away from a task, that task receives a preemption notification on one of its remaining processors. When this happens, the preempted thread is handled according to the lock resolution policy, then put on the ready queue. Finally the next ready thread is dequeued and run.

4.3 Critical Sections

Critical sections pose problems for both kernel and user-level threads. The problem arises because preemption can occur while a thread is in a critical section, and this can impede the progress of other threads that wish to enter the critical section.

Various solutions to this problem have been proposed [And91] [BRE92] [Her91]. Each of these solutions makes certain assumptions regarding the prevalence and characteristics of critical sections, and each requires a certain degree of system support, entailing a certain amount of implementation complexity. We chose to defer the evaluation of which (or which combination) of these was the best choice for our purposes, instead selecting a temporary solution that, while imperfect, was attractively easy to implement.

The basis for all user-level activity in our system is CThreads, which makes widespread use of spinlocks to guard critical sections. Our scheme therefore focuses on allowing the system to quickly restart threads that have been preempted while holding spinlocks. We rely upon the use of a per-thread lock counter: lock acquisition increments the counter, while lock release decrements it. Our upcall handlers check the lock counts of all the user-level threads referenced by their arguments, and attempt to run each runnable, lock-holding thread until its lock count reaches zero. This is done *before* attempting to acquire any run queue locks, to prevent deadlocks caused by a preempted thread holding a run queue lock. We return control to the scheduler if we become the subject of a notification while spinning on the lock (we block or are preempted), when the last lock is released, and after failing to acquire a spinlock after a number of attempts.

Further details of our lock handling strategy are presented in Section 4.4.3 and Appendix B.

4.4 Implementation Details

We used the "Mach Thread" implementation of CThreads as the basis of our thread library. This implementation is optimized for use on top of kernel threads, in that it takes measures to explicitly block kernel threads not busy with user-level work. However, a task's need to create more kernel threads than it has processors is an artifact of the poor support for parallelism provided by traditional kernels. That is, the task must create additional kernel threads so that its processors can be kept usefully busy if one of its active kernel threads should happen to perform a blocking operation (or page fault). Further, the task must ensure that these "reserve" kernel threads remain blocked until needed, lest the kernel's time-sharing policy cause them to interfere with their usefully busy colleagues.

A scheduler activations kernel eliminates the need for the task to concern itself with such kernel thread management issues. The kernel ensures that the task has exactly as many scheduler activations as it does physical processors. The task need not create "reserve activations" because the kernel further ensures that the task receives notifications of blocking events: when one of the task's activations blocks, the kernel returns the processor to the task, along with a new activation that may be use to run other user-level work.

A scheduler activations kernel thus simplifies this aspect of the implementation of a user-level threads package. We were therefore able to streamline CThreads by eliminating its kernel thread management code and data structures (in particular the `cproc` structure, used by the original package to represent kernel threads).

Scheduling in the resulting package is simplified because we need deal with only one type of entity: a user-level thread. We do, however, differentiate between threads that are handling upcalls and threads that are performing other work. Upcall-handling threads are run in preference to those doing user-level work because the scheduler needs to maintain an accurate view of the system. Our implementation retains CThreads' FIFO scheduling of threads running user-level work. Threads are run according to the following policy:

- if there is a thread in the stack of upcalls, pop and run it
- otherwise, if there is a thread in the run queue, dequeue and run it
- otherwise, spin for some time waiting for additional work, then mark this processor as "willing to yield"
- spin, waiting for work to arrive, when it does, mark this processor as busy, and run the work

The following sections describe other details of our implementation.

4.4.1 Stack-Based Identification

One of the first duties of each upcall handler is to identify the user-level threads affected by the event(s) that its arguments describe. We considered two schemes for this:

- the existing CThreads mechanism of using a pointer stored at the base of the user-level stack
- a mapping between scheduler activations (as named by the kernel) and the user-level threads they are running

We decided to use the CThreads scheme, although it caused some implementation difficulties (see below), because we thought it easier than trying to maintain the consistency of a mapping between scheduler activations and user-level threads. Instead, we find the base of the stack (which is simple, because CThreads stacks are aligned on multiples of power-of-2 addresses), and then offset to find a pointer to the thread control block, thus identifying the thread.

Although the stack-based identification scheme appears simple, it is complicated by the existence of the Unix system call emulator [GDFR90]. This is a body of code mapped into the address space of each task created by the Unix server. When a task makes a Unix system call, a trap instruction carries it into kernel space. The kernel realizes that the system call is to be emulated, and returns to user space, but in the emulator. Here, the emulator code switches stacks before performing whatever operations are necessary to carry out the call (such as sending a message to the Unix server).

This means that if an activation blocks while executing emulator code, its user-level context could include a stack pointer that does *not* point at a properly initialized stack. If in the `sa_blocked` handler we try to determine our user-level thread identity using the current stack, we would at best get an incorrect answer, and more likely generate an exception. To deal with this, we have to be able to identify emulator stack pointers correctly, and understand how to determine the "real" user-level stack from them.

This introduces an unfortunate dependence between our CThreads package and the operation of the emulator. Modifying the emulator code to properly initialize the new stack would eliminate this problem. However, we were unsure whether such a change would have to be made in all emulators (i.e., emulators providing environments other than Unix) that run on top of Mach. Under the assumption that this were true, we chose to handle the problem in CThreads (which we had already modified) rather than to depend upon emulator alterations.

4.4.2 Stack Management

We mentioned in Section 2.1 that at startup, we provide the kernel with a set of stacks for use when running notifications. Since we preallocate a finite (perhaps even small) number of stacks, we need to ensure that the kernel does not run out as we receive successive notifications. We do this by passing stacks back to the kernel when they are no longer needed (using the `task_recycle_stacks()` system call). This is done from within the context switch function used to leave an upcall handler and continue with user-level work (`cthread_switch_and_recycle()`). Once we have switched to the new stack, we check to see if the old one belonged to an upcall handler; if so, we recycle it. Some caching of these stacks is done at user-level, so that we do not incur the overhead of a system call for every context switch out of an upcall handler.

4.4.3 Lock Handling Details

As described in Section 4.3, when a thread that holds a spinlock is preempted, the upcall that processes the notification is usually able to continue the lock-holding thread until it releases its last lock. When none of the threads in the notification hold any spinlocks, the upcall is able to acquire the global run queue lock[†], and enqueue the runnable threads. At this point, the upcall is finished, so it exits, handing off to the next ready thread.

Occasionally, an upcall that is processing preempted lock-holding threads may itself be preempted. This scenario is depicted in Figure 1. The situation depicted could have occurred due to a processor reallocation, as follows: Upcall U_1 was handling the lock-holding threads T_1 and T_2 (holding locks L_A and L_B respectively) when it was preempted, because the processor it was running on was assigned to another task. Delivering the notification of this preemption event interrupted another lock-holding thread, T_3. T_3 already holds lock L_C, and was preempted while trying to acquire L_B.

In the general (though rare) case, the tree of upcalls handling lock holders is more than one level deep. The threads at each level of the tree may be waiting for locks held by threads at any other level of the tree. To prevent deadlock in this situation, it is sufficient to traverse the tree of upcalls handling locks, in order to run the lock-holding threads until each thread is able to release all of its locks. The traversal is made possible by

[†]It may have to spin for a while if the lock is held on another processor, but it is safe from deadlock.

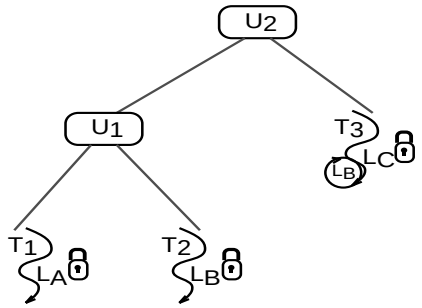

Figure 1: Nested Upcall Handling

having the spinlock acquire code return control to the upcall handler continuing the thread after a number of failed spins[†], so it can attempt to continue other lock-holding threads.

In the example shown in Figure 1, if U_2 continues T_3, T_3 will spin unsuccessfully trying to acquire L_B, eventually returning control to U_2. U_2 continues U_1, which in turn continues T_1 and T_2. T_1 and T_2 will both release their locks after completing their critical sections. U_1 then requeues T_1 and T_2 on the ready queue and exits, handing off back to U_2, which is now able to successfully continue T_3.

Multiple processors may be simultaneously busy running upcalls which are continuing lock-holding threads. Threads on one of the processors may be trying to acquire locks held by threads on another processor and vice versa. Even this will not result in deadlock, because each processor guarantees that all lock-holding threads are able to make progress. For this guarantee to be valid, each lock-holding thread must be either running on a processor or the subject of a notification, so an upcall handler is able to continue it. This condition holds because properly structured programs do not allow threads to block on user-level synchronization operations while holding a spinlock.

A complete description of the implementation of this lock resolution policy can be found in Appendix B.

4.5 Possible Optimizations

As in the case of the kernel, the user-level implementation leaves many opportunities for optimization. In particular, although our package differs (internally) from standard CThreads, it might benefit from the improvements to the latter that have been recently suggested [Dea93]. As described in section 4.4, our package already includes one of these optimizations, that of having idle threads spin (rather than block) waiting for work. We are currently experimenting with the other ideas proposed in [Dea93].

4.5.1 Trap State Restores

The low overhead of context switching at user-level is one of the principal attractions of user-level thread systems. In the common case (a thread switching due to user-level synchronization or termination), our CThreads retains this low overhead. However, when context switching to a thread that was previously the subject of a notification, the overhead is increased somewhat. This is because the state that we must restore was saved by the kernel trap handler when the thread crossed into the kernel (i.e., the complete register state,[‡] rather than just scratch registers). We are provided with the complete state as part of the upcall arguments, and use it without further modification during our context switch routine in a manner similar to the kernel's `thread_exception_return()`.

On the Sequent, some of the register restores are probably redundant, given Mach's use of the 80386's flat address mode [Int87]; in particular, it seems likely that the segment registers do not need to be restored by

[†] Threads that aren't being continued by an upcall continue to spin until they acquire the lock.

[‡] As previously explained, "complete register state" does not necessarily include the floating-point registers, which are passed by the kernel only when appropriate (see Section 2.3.1).

every user-level context switch, since they are constant across all activations in a task. Avoiding these redundant restores, as well as careful attention to register usage in the upcall handlers, could reduce the switch overhead.

4.5.2 Nested Block/Unblock Handling

Due to interactions between Mach's IPC mechanism, virtual memory, and scheduling, a (matched) pair of `blocked`/`unblocked` notifications can often be delivered extremely closely together. If the `blocked` notification is interrupted in order to deliver the `unblocked` notification, we end up in a situation that is potentially quite expensive to handle. The `sa_unblocked` handler will notice that it has been told about an interrupted scheduler activation, and will attempt to run it at some point, necessitating at least two user-level context switches before we get back to running user-level work. It would be preferable to have the `sa_unblocked` handler realize that the interrupted activation is a `blocked` notification regarding *the same activation* that it is notifying has unblocked. It could then short circuit some of the normal scheduling, and run the relevant user-level thread somewhat sooner. This can be of considerable benefit when there are multiple nested sets of `blocked`/`unblocked` notifications, and is simple to implement.

4.5.3 Per-Processor Run Queues

Per-processor queues of ready threads are a commonly suggested optimization technique [ALL89] [FL89]. This could fairly easily be implemented on top of our system, but with mitigated benefits. In a non-preemptive threads package, per-processor ready queues can allow queue operations to proceed without synchronization, significantly improving performance. With scheduler activations, however, notifications can come at any time, possibly interrupting threads manipulating run queues; processing notifications therefore requires synchronized access to run queues. Even so, this may be a worthwhile optimization if the run queue is a bottleneck.

5 Performance

The goal of scheduler activations is to combine the functionality of kernel threads with the performance and flexibility advantages of managing parallelism at the user level within each task. To evaluate the effectiveness of our implementation, we first measure the costs associated with delivering notifications of kernel events. Next, we isolate the impact of preemption, demonstrating that scheduler activations allow applications to overcome its detrimental effects. Finally, we measure the benefits of scheduler activations in practice, using a multiprogrammed workload consisting of several concurrent copies of a real scientific program.

5.1 Upcall Performance

We measure the cost of notification as the time from the start of notification creation to the arrival of the notification at the user-level. The cost of notification can be broken into the following components:

- *kernel overhead:* the time required to return from kernel to user-level.
- *notification preparation:* the time to create and prepare a scheduler activation for notification, not including the data movement cost.
- *data movement:* the time to push the arguments, including activation state, onto the notification stack.

Table 1 shows a breakdown of the cost of each type of notification. Since each notification follows an identical path to transfer from kernel to user space, the kernel overhead is the same for each. Small differences in code

Upcall type	total latency	kernel overhead	notification preparation	data movement
`blocked`	620	90	230	300
`unblocked`	915	90	225	600
`preempted`	860	90	170	600
`processor_added`	490	90	170	230

Table 1: Upcall Latencies (times in microseconds)

paths account for the (correspondingly) small differences in notification preparation time. The major difference between the costs of the notifications – and the major contributor to the cost of each – is the data movement cost.

For `blocked` notifications, state is passed for the new activation (carrying the notification), but only minimal state is passed for the blocked activation (the *event_sa*). This is because this state is not useful to the user-level thread scheduler until the activation has unblocked; the purpose of notification in this case is not so much to inform the task of an interesting event, but rather to return the processor so that it may used while the old activation is blocked. For this reason, `blocked` activations incur about the same data movement cost as do `processor_added` notifications, which have no *event_sa*.

The `unblocked` and `preempted` notifications, however, are more expensive. Each such notification involves an *event_sa* whose state must be passed to user-level; furthermore, delivering either of these notifications typically requires interrupting a running activation, and the state of this activation must also be copied out to user space. The data movement cost associated with `unblocked` and `preempted` notifications is therefore much higher than that incurred by the two previously discussed.

In the remainder of this section, we demonstrate that the benefits of using scheduler activations outweigh the cost of posting (and handling) notifications.

5.2 The Cost of Preemption

When a processor is preempted from a task, the kernel can deal with the (kernel) thread running on that processor in three distinct ways:

- *The kernel thread is suspended for the duration of the preemption*. Since the kernel thread is executing some user-level thread, that thread also remains suspended until the task receives another processor.
- *The kernel thread competes for the remaining processors*. The kernel thread scheduling policy handles the mismatch between kernel threads and processors by multiplexing the threads on the processors (e.g., using time-slicing). User-level threads make progress whenever the kernel policy happens to dictate that their associated kernel thread is run.
- *Scheduler activations are used to notify the task of the preemption.* The task's thread scheduler is given the state of the associated user-level thread, which may now be run according to the user-level scheduling policy.

The first of these approaches (suspension) can lead to extremely poor performance if the suspended thread is critical to the progress of the application. We illustrate this by using an artificial benchmark: a simple "pingpong" program consisting of two user-level threads alternating access to a shared data structure. This task is initially assigned two processors, and runs in its own processor set to isolate it from other activity on the machine. We

arranged processor preemption so that the task experiences a fixed number of preemptions (50), each lasting for *M* msec.

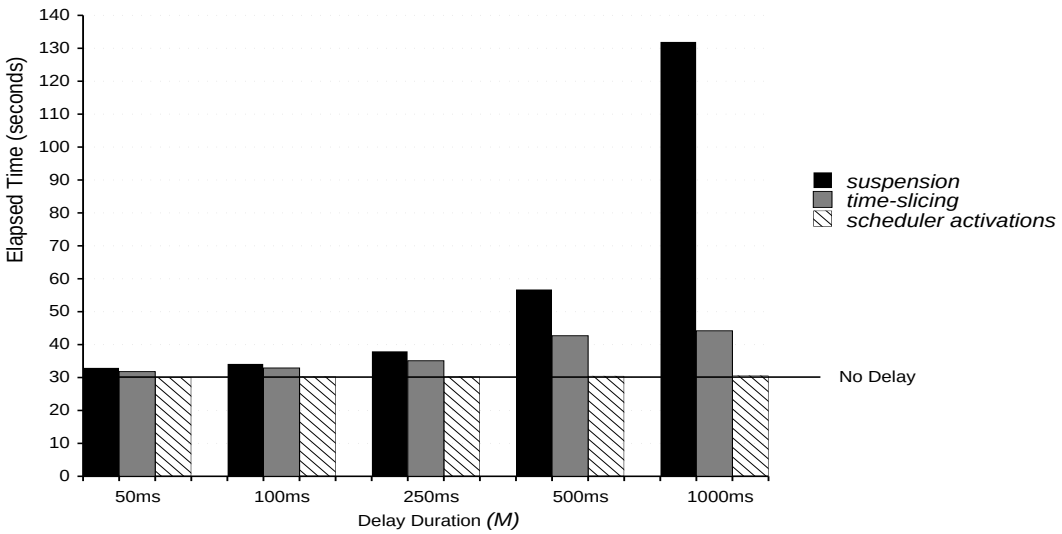

Figure 2: Effect of (50) Preemptions of Varying Length

Figure 2 shows the elapsed time for this program under the three approaches (*suspension*, *time-slicing*, and *scheduler activations*) for various values of *M*. Included is a "reference" line denoting the time required to complete the job on two processors in the absence of preemptions.

Although the benchmark used here is an artificial one, it exemplifies an application that is sensitive to the delaying of any of its threads. The results for the *suspension* case in Figure 2 therefore reflect the "maximum" cost of a preemption, since in this case the application quickly becomes paralyzed if either thread is delayed. The *time-slicing* case performs much better, because the kernel policy ensures that the delayed thread does make progress. However, in this case, kernel threads are scheduled without regard to the user-level work they may be performing. Better performance is achieved by the *scheduler activations* case, in which the task's thread scheduler is allowed to extract the state of the delayed user-level thread, after which the thread scheduler can make its own (wiser) decision as to which thread should be allowed to execute.

These results suggest that for this artificial benchmark, the detriment of oblivious kernel scheduling, while noticeable, is relatively small. In the next subsection, we show that this is a much more important factor in practice.

5.3 Application Performance

To demonstrate the advantages of scheduler activations over oblivious kernel scheduling, we study the average elapsed time experienced by a real application running in a multiprogrammed environment.

The application we use, called *Gravity*, is an implementation of the Barnes and Hut clustering algorithm for simulating the gravitational interaction of a large number of stars over time [BH86]. The program contains five phases of execution; these phases are repeated for each timestep in the simulation. The first phase is sequential,

while the last four are parallel. Between each of the parallel phases is a barrier synchronization at which the parallelism decreases briefly to one.

We derived the speedup curve for the application, finding that performance was best if it were allowed to run (in isolation) on 12 processors. We then arranged to run several (1-5) concurrent instances of *Gravity* under two different processor allocation policies. We first ran the jobs using the existing kernel thread scheduling policy: each job created a kernel thread per processor desired (12), and the jobs were simply allowed to run to completion on our (20-processor) machine.

Next, we used our processor allocator to provide a simple "space-sharing" policy, of the sort previously shown to provide good performance for this type of workload on this class of machine [TG89, MVZ90]. This allocator simply divides the available processors equally amongst the jobs: given P processors and J competing jobs, each job is allocated P/J processors. On each job arrival, the allocator computes a new equipartition allocation, and processors are preempted from active jobs for reassignment to the new arrival. Preemptions are coordinated by using scheduler activations to notify active jobs of processor loss. The jobs' thread schedulers therefore have immediate access to interrupted threads, and the kernel ensures that the jobs always have exactly as many activations as processors (avoiding oblivious kernel scheduling).

Note that, as described in [MVZ90], better performance can be achieved by having the processor allocator respond to dynamic changes in the jobs' parallelism (i.e., processor demand) by reallocating processors from jobs that cannot currently use them to jobs that can. Although we have implemented such a policy (see Section 3), we do not avail ourselves of it here.

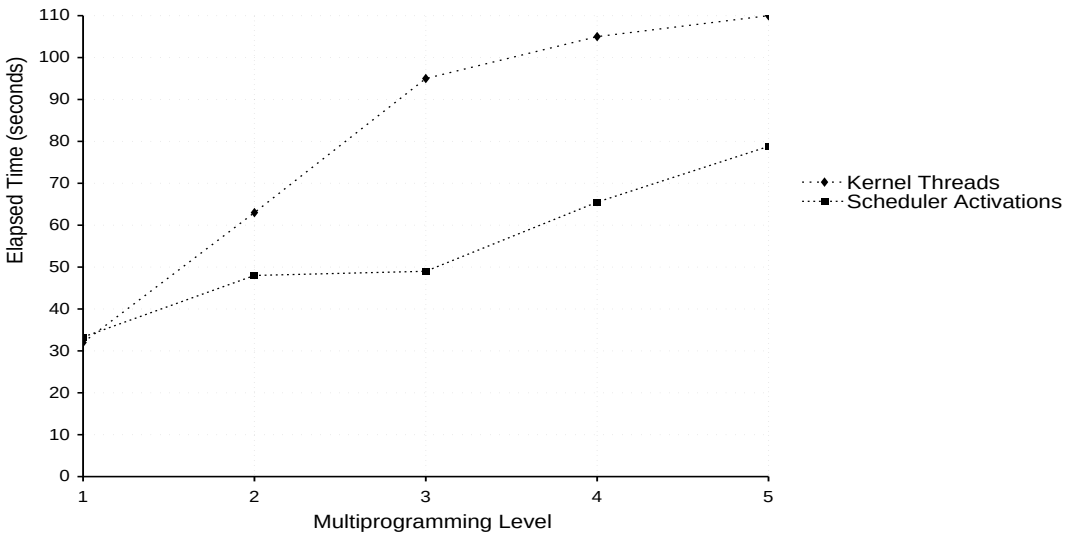

Figure 3: Multiprogrammed Performance of *Gravity*

Figure 3 shows the average runtime of each of the simultaneous runs of *Gravity* as the number of concurrent (competing) jobs increases from one to five.

These results demonstrate that user-level threads built on top of kernel threads can yield extremely poor performance in the presence of competition for processors. Specifically, *Gravity* — which exhibits a "phase" structure common to many scientific programs — can exhibit poor performance if its threads do not reach their internal barriers (between phases) in a timely fashion. Without scheduler activations, scheduling decisions are

made by the kernel, and the application therefore cannot guarantee that working threads are given preferential access (over idle threads) to processors. Conversely, when scheduler activations are used, the tasks' thread schedulers have complete control over which threads are allowed to run, and can arrange a schedule that provides good results.

6 Summary

We have incorporated scheduler activations into the Mach operating system. We have modified CThreads to take advantage of this support, and implemented a processor allocation module which provides an appropriate policy. The result is a version of Mach in which applications using our CThreads library run with high efficiency, and with the desirable functionality of kernel threads in the presence of system events such as page faults, I/O, and processor preemption. Our implementation favored expediency, Mach compatibility, and portability:

- *Expediency.* We successfully leveraged off existing mechanisms. Examples of this are our use of continuations, processor sets, and kernel threads. We did not use all possible existing mechanisms, however. To implement notifications, we use kernel to user-level upcalls instead of Mach's message facility; this was a personal choice of expediency. It also cut through some layers of IPC code, perhaps gaining performance.

  We emphasize that the interface as described here is not the final word on how best to present scheduler activations as a kernel mechanism. Our interface implements the functionality that any scheduler activations kernel must, but it remains an unanswered question whether the interface is more efficiently implemented via Mach's IPC system.

- *Backward compatibility.* Our modified kernel is entirely backward compatible with the unmodified Mach kernel. User-level programs and the kernel both behave exactly as before until a task turns on scheduler activations. Even when scheduler activations tasks are running, our use of the processor set mechanism ensures that other tasks can be isolated from their behavior. Finally, applications can enjoy improved functionality without being rewritten; since the thread system interface is unchanged, applications need only be recompiled and linked to the new thread system that encapsulates event management.

- *Portability.* This issue must be addressed with respect to both the design and the implementation of our system. Since scheduler activations are largely oriented towards shared-memory multiprocessor machines, our design reflects this bias. It can nonetheless be mapped to uniprocessor machines quite easily. The main difficulty on such machines is that the (single) processor must necessarily be time-shared, rather than space-shared as are the (multiple) processors on our Sequent. Decisions regarding the handling of processor allocation in such an environment are necessary (e.g., notifications regarding processor addition and preemption could simply be suppressed). The design of the remaining mechanisms – for performing the other notifications, for example – should be directly portable to uniprocessor architectures.

  The portability of the implementation hinges on the degree to which code is processor-dependent. The lone processor dependency in the (added) kernel code is the routine for building upcall stacks, which inherently relies on processor-specific information regarding stack layout and trap/exception context format. Similarly, our changes to CThreads have not affected its processor-dependence, which remains hidden in the context switch, thread startup and spinlock primitive code. Our upcall handlers have some processor dependencies, but these are also hidden via judicious use of processor-specific header files.

Our implementation should provide an effective testbed for future experimentation with scheduler activations, and a useful model for future implementations. We hope that as experience with scheduler activations grows, their impact as a structuring mechanism for future kernels can be more fully understood and exploited.

7 Acknowledgments

We would like to thank Brian Bershad for structuring suggestions which helped us integrate scheduler activations into Mach as smoothly as possible, and David Black for helping us with some difficulties with the kernel. Brian Bershad and others at CMU also helped us get Mach 3.0 running on our Sequent. Discussions with Tom Anderson regarding his implementation provided useful insights. David Kays and Thu Nguyen were involved in several aspects of this project. Sape Mullender gave valuable advice regarding an initial uniprocessor version of this work, done by Vaswani and Peter Bosch at the University of Twente. Jeff Chase provided helpful comments on this paper.

References

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. Also appeared in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[And91] Thomas E. Anderson. *Operating System Support for High Performance Multiprocessing*. PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991.

[BH86] J. Barnes and P. Hut. A Hierarchical $O(n \log n)$ Force-Calculation Algorithm. *Nature*, 24:446–449, 1986.

[Bla90] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[CD88] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[Dea93] Randall W. Dean. Using Continuations to Build a User-Level Threads Library. In *Proceedings of the Third USENIX Mach Symposium*, April 1993.

[FL89] John Faust and Henry M. Levy. The Performance of an Object-Oriented Thread Package. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1989.

[GDFR90] David Golub, Randell W. Dean, Alessandro Forin, and Richard F. Rashid. Unix As An Application Program. In *Proceedings of the Summer USENIX Conference*, June 1990.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[Int87] Intel Corporation. *i386 Microprocessor Programmer's Reference Manual*, 1987.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[MVZ90] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed, Shared Memory Multiprocessors. Technical Report 90-03-02, Department of Computer Science and Engineering, University of Washington, March 1990. To appear in *ACM Transactions on Computer Systems*.

[RBF+89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Doug Orr, and Richard Sanzi. Mach: A Foundation for Open Systems. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.

[TG89] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[TSS88] Charles P. Thacker, Lawrence Stewart, and Edward Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

A Kernel Implementation Summary

This Appendix describes our new system calls, the kernel data structures modified to support scheduler activations, and pseudocode for kernel event management.

A.1 New System Calls

```
/*
 * register a set of upcall handlers for this task
 */
kern_return_t task_register_upcalls(vm_offset_t *upcalls);

/*
 * turn on scheduler activations for this task.
 * transforms the calling thread into an activation.
 */
kern_return_t task_use_scheduler_activations(boolean_t use_sas);

/*
 * provide a set of stacks for the kernel to use for notifications.
 */
kern_return_t task_recycle_stacks(int num_to_recycle, vm_offset_t *stacks);

/*
 * mark this activation as "willing to yield".
 */
kern_return_t thread_willing_to_yield(boolean_t i_am_idle);
```

A.2 Additions to Kernel Structures

A.2.1 Thread

```
struct thread {
   ...

   /* when non-NULL, called instead of returning to user-level */
   void            (*ul_continuation)();

   /* toggles whether the thread is an activation or not */
   boolean_t       is_sa;

   /* signifies whether an activation is busy with user-level work */
   boolean_t       willing_to_yield;

   /*
    * processor on which the event associated with a notification took
    * place (e.g., where the activation was running when it blocked).
    * note that this is not the same as last_processor.
    */
   processor_t     event_processor;

   /* the following are used during notifications: */
   upcall_t        upcall;           /*  - upcall identifier */
   struct thread   *event_sa;        /*  - activation that caused event
                                     *    for which we're notifying */
   queue_head_t    interrupted_sas;  /*  - activations (possibly none)
                                           preempted for the notification */
   ...
}
```

A.2.2 Task

```
struct task {
   ...

   /* user-level entry points, protected by the normal task lock */
   vm_offset_t         upcalls[SA_NUM_UPCALLS];

   /* lock for all other SA-related structures, held at splsched */
   decl_simple_lock_data(,sa_lock)

   /*
    * task uses this to control whether or not notifications are sent.
    * actually, this needs to be true AND the thread in question needs
    * to be an activation for a notification to be sent.
    */
   boolean_t           using_sa;

   /*
    * task_hold() tries to suspend the whole task (each thread).
    * to do this, we need to turn off activations before trying
```

```
     * to suspend.  for later task_release(), we have to "remember"
     * whether the task was using activations (see task.c)
     */
    boolean_t           was_using_sa;

    queue_head_t        user_stacks;    /* stacks for upcalls */
    queue_head_t        recycled_sas;   /* pool of recycled activations */

    ...
}
```

A.2.3 Processor Set

```
struct processor_set {
    ...

    /* number of processors on which "willing to yield" threads are running */
    int             willing_count;

    ...
}
```

A.2.4 Processor

```
struct processor {
    ...

    /* last event on this processor (e.g., "an activation blocked") */
    int             event;

    ...
}
```

A.3 Event Management Pseudocode

A.3.1 Blocking

- `thread_block()` is called normally by the *blocking activation*.
- a notification is needed when *blocking activation*`->task->using_sa &&` *blocking activation*`->is_sa`.
- `thread_block()` sets:
  - *blocking activation*`->ul_continuation` to `sa_notify()`, as the routine to run when the activation unblocks.
  - *blocking activation*`->event_sa` to *blocking activation* (see Section A.3.2).
  - the processor `event` flag to `PROC_SA_BLOCKED`.
- *blocking activation* falls into `thread_select()`, normally.
- `thread_select()` notices `PROC_SA_BLOCKED` and prepares a `blocked` notification[†]:

[†] `thread_select()`'s normal duty is to choose the next thread to run on a processor. We have modified `thread_select()` to perform notifications (instead of choosing a ready thread) when the processor's `event` flag is set appropriately.

- selected thread = a *new activation*
- `thread_select()` then:
  - sets *new activation*`->event_sa` to *blocking activation*.
  - sets *new activation*'s start function to `sa_notify()`.
  - returns *new activation* for invocation (via `thread_invoke()`).
- when invoked, *new activation* enters `sa_notify()`, which:
  - grabs an upcall stack from *new activation*`->task->user_stacks`.
  - finds the upcall handler address in *new activation*`->task->upcalls`.
  - constructs a dummy trap return context to implement the upcall:
    - `pc` = `sa_blocked` handler
    - `sp` = upcall stack pointer
  - pushes the upcall arguments onto this stack
    - *new activation, blocking activation, interrupted_sas* (`NULL`)
  - finishes the upcall via `thread_exception_return()`.

A.3.2 Unblocking

- some normal kernel mechanism (e.g., the disk interrupt handler) makes an activation (*unblocked activation*) runnable, placing it on some run queue.
- the existence of the runnable activation causes some other activation in the processor set to notice that it should set an AST for itself (see Section 2.3.3).
- this causes the activation taking the AST (*interrupted activation*) to fall through to `thread_block()`. `thread_block()` notices the *interrupted activation*`->task->using_sa` and *interrupted activation*`->is_sa` flags, but that *interrupted activation*'s status is still runnable, which implies that this is a scheduler activation preempting itself to pick up a notification. accordingly, it sets the processor's `event` flag to `PROC_SA_HANDLING_EVENT`, [†] and falls into `thread_select()`.
- `thread_select()` finds *unblocked activation* and adds *interrupted activation* to *unblocked activation*'s `interrupted_sas` list. *unblocked activation*'s `event_sa` was previously set to point to itself, so at this point it is returned for invocation (via `thread_invoke()`).
- *unblocked activation* runs in the kernel to clean up after the unblock (e.g., when handling a page fault, it needs to do a `pmap_enter()` for the new page).
- *unblocked activation* attempts to return to user space: if it was a blocking system call, via `thread_syscall_return()`; if a trap/fault, via `thread_exception_return()`. in either case, it passes through the kernel function `return_from_trap()`.
- `return_from_trap()` notices *unblocked activation* `->ul_continuation` is non-`NULL` (it was set when the activation blocked) and calls it instead of returning to user space.
- the `ul_continuation` is `sa_notify()`, which simply uses *unblocked activation* to carry the notification.
- *unblocked activation* enters `sa_notify()`, which:
  - clears *unblocked activation*`->ul_continuation`.
  - grabs an upcall stack from *unblocked activation*`->task->user_stacks`.
  - finds the upcall handler address in *unblocked activation*`->task->upcalls`.

[†] This serves to differentiate the event from a processor reallocation, for which a notification is required.

- constructs a dummy trap return context to implement the upcall:
  - `pc` = `sa_unblocked` handler
  - `sp` = upcall stack pointer
- pushes the upcall arguments onto this stack
  - *unblocked activation, unblocked activation, interrupted_sas*
    the *interrupted_sas* list includes *interrupted activation*.
- copies the register state (on trap) of *unblocked activation* and all *interrupted_sas* onto the upcall stack, so that it can be passed to the task's thread system.
- finishes the upcall via `thread_exception_return()`. `ul_continuation` is `NULL` this time, so the activation proceeds to user space.

A.3.3 Processor Preemption

Recall that the standard way to move a processor between processor sets involves using an *action thread* to perform the necessary (kernel) data structure updates. Once this is done, the *action thread* blocks, awaiting a new chore to perform.

- the kernel begins the reallocation by forcing the *action thread* onto the processor in question.
- the activation running there notices the runnable *action thread*, mistakes it for a pending notification, and preempts itself.
- the kernel adds this *preempted activation* to the *action thread*'s `interrupted_sas` list.
- once the *action thread* has removed the processor from the old processor set, it creates a *new activation* to perform the preemption notification for (the task in) that set.
- *new activation*'s start function is set to `sa_notify()`.
- *new activation*'s `event_sa` is set to *preempted activation*
  (transferred from *action thread*`->interrupted_sas`).
- *new activation* is made runnable, and is inserted into the old processor set's run queue.
- some other activation in the old processor set will notice the runnable activation, and interrupt itself to deliver the notification.
- the activation interrupting itself (*interrupted activation*) falls into `thread_block()`. `thread_block()` notices *interrupted activation*`->task->using_sa &&` *interrupted activation*`->is_sa`, but *interrupted activation*'s status is still runnable, which implies that this is a scheduler activation interrupting itself to pick up a notification. accordingly, it sets the processor's `event` flag to `PROC_SA_HANDLING_EVENT`, and falls into `thread_select()`.
- `thread_select()` finds *new activation*, and:
  - adds *interrupted activation* to *new activation*`->interrupted_sas`.
  - sets *new activation*'s start function to `sa_notify()`.
  - returns *new activation* for invocation (via `thread_invoke()`).
- when invoked, *new activation* enters `sa_notify()`, which:
  - grabs an upcall stack from *new activation*`->task->user_stacks`.
  - finds the upcall handler address in *new activation*`->task->upcalls`.
  - constructs a dummy trap return context to implement the upcall:
    - `pc` = `sa_preempted` handler
    - `sp` = upcall stack pointer
  - pushes the upcall arguments onto this stack

        - *new activation, preempted activation, interrupted_sas*
          the *interrupted_sas* list includes *interrupted activation*.
    - copies the register state (on trap) of *preempted activation* and all *interrupted_sas* onto the upcall stack, so that it can be passed to the task's thread system.
    - finishes the upcall via `thread_exception_return()`.

Once the processor has been moved into the new processor set (but before it is allowed to run any activations there), the kernel sets the processor `event` flag to PROC_ALLOCATED, signifying that the processor has been newly allocated to the processor set. If the new processor set is using scheduler activations, the task is notified of the processor's arrival. This is described below.

A.3.4 Processor Allocation

Informing a task of a processor's arrival can be done in two ways: implicitly and explicitly. These two methods correspond to whether or not there is already a runnable activation in the new processor set.

As explained above, the processor is moved into the new processor set by the *action thread*, which blocks itself when its work is complete.

- the blocking *action thread* falls into `thread_select()`, looking for an activation (in the new processor set) to run.
- *Implicit:*
    - if `thread_select()` finds a runnable activation, it simply returns it for invocation.
    - the found activation must be carrying some sort of notification. specifically, it must be carrying either an `unblocked` notification or a `preempted` notification[†]. it cannot be a `blocked` notification because those never queue, but rather are launched immediately on the same processor (see above).
    - `unblocked`/`preempted` notifications normally cause some other activation to be interrupted. since this notification has been delivered without having to resort to preemption, its arrival in user space implicitly informs the task that it has gained a processor.
- *Explicit:*
    - if there is no runnable activation in the new processor set, then an explicit `processor_added` notification is required.
    - `thread_select()` finds no activation to run, and would normally return the (standard) idle thread. however, it notices that the `PROC_ALLOCATED` event is set for this processor. in this case, rather than returning the idle thread, it instead:
        - creates a *new activation* to carry the `processor_added` notification.
        - sets *new activation*'s start function to `sa_notify()`.
        - returns *new activation* for invocation (via `thread_invoke()`).
    - when invoked, *new activation* enters `sa_notify()`, which:
        - grabs an upcall stack from *new activation*`->task->user_stacks`.
        - finds the upcall handler address in *new activation*`->task->upcalls`.
        - constructs a dummy trap return context to implement the upcall:
            - `pc` = `processor_added` handler
            - `sp` = upcall stack pointer
        - pushes the upcall arguments onto this stack
            - *new activation,* `NULL`, `NULL`
        - finishes the upcall via `thread_exception_return()`.

[†] A `preempted` notification would be found here only if another processor had recently been taken away, which typically would not occur in such close proximity to a processor being added.

B User-Level Thread System

B.1 Relevant Structures

```
struct cthread
{
   ...

   int             switch_type;         /* trap or user context switch? */
   int             cpu_number;
   thread_fp_state fp_context;

   int             type;                /* upcall, idle, or normal */
   boolean_t       fp_valid;

   struct cthread *next_context;        /* see lock_handler() */
   struct cthread *parent_upcall;
   int             lock_count;

   ...
};
```

B.2 Thread System Operation

This section illustrates the general operation of some key thread system functions (upcall handlers and locking primitives). It should not be considered a comprehensive description of these functions – many details necessary for ensuring correct behavior have been left out in the interest of brevity. The only complete description of these details currently available is the (commented) source code itself.

```
sa_blocked (
   upcall_arg_t  active,
   upcall_arg_t  event,
   upcall_arg_t  interrupted
) {
   queue_t   interrupted_threads = interrupted;

   identify active_thread;

   /*
    * The event thread is useless to us, since we can't do anything
    * with its state (it's blocked, and can't be rolled forward).
    * So, all we need to do is deal with any interrupted threads we
    * might happen to have (usually none).
    */

   finish_upcall(active_thread, interrupted_threads);

   /* NOTREACHED */
}
```

```
sa_unblocked (
   upcall_arg_t  active,
   upcall_arg_t  event,
   upcall_arg_t  interrupted
) {
   queue_t   interrupted_threads = interrupted;

   identify active_thread;
   identify event_thread;

   /*
    * The unblocked thread is runnable, as are any interrupted ones.
    * So, just add the event_thread to the interrupted_threads list,
    * and deal with all of them at once.
    */
   event_thread->next = interrupted_threads;
   interrupted_threads = event_thread;
   finish_upcall(active_thread, interrupted_threads);

   /* NOTREACHED */
}
```

```
sa_preempted (
   upcall_arg_t  active,
   upcall_arg_t  event,
   upcall_arg_t  interrupted
) {
   queue_t   interrupted_threads = interrupted;

   identify active_thread;
   identify event_thread;

   /*
    * As in the case of an sa_unblocked event, the preempted thread
    * is runnable, as are any interrupted ones.  Again, just add the
    * event_thread to the interrupted_threads list, and deal with
    * all of them at once.
    */
   event_thread->next = interrupted_threads;
   interrupted_threads = event_thread;
   finish_upcall(active_thread, interrupted_threads);

   /* NOTREACHED */
}
```

```
sa_add_processor (
   upcall_arg_t  active,
   upcall_arg_t  event,
   upcall_arg_t  interrupted
) {
   identify active_thread;

   /* Nothing to do -- run next ready thread */
   get ready_thread;
   switch to ready_thread;

   /* NOTREACHED */
}
```

```
finish_upcall (
   cthread_t  active_thread,
   queue_t    interrupted_threads
) {
   queue_t    lock_holders;

   if (interrupted_threads != NULL) {
      /*
       * make_holders_list() returns NULL if there are none.
       * Otherwise, it returns a singly-linked list of all the
       * interrupted_threads -- each TCB points to the next, and
       * the last one points to active_thread.
       */
      lock_holders = make_holders_list(active_thread, interrupted_threads);
      if (lock_holders == NULL) {
         enqueue each of the interrupted_threads on normal readyq;
      } else {
         lock_handler(active_thread, lock_holders);
      }
   }

   /* Done -- run next ready thread */
   get ready_thread;
   switch to ready_thread;

   /* NOTREACHED */
}
```

```
lock_handler (
   queue_t   active_thread,
   queue_t   lock_holders_queue
) {
   /* Private storage for threads to be put on readyq */
   queue_t runnable_queue;

   while lock_holders_queue is not empty  {

      /*
       * First filter out any threads that no longer need to
       * remain in the queue.
       *
       * WARNING:  the test for this condition, as presented here,
       * is a drastic simplification.  See the source code ...
       */
      for each member of lock_holders_queue  {
         if (lock_count == 0 && is NOT an upcall handler)  {
            /*
             * Remove from lock_holders_queue.
             * Enqueue on our private runnable_queue.
             * No locks are required to do this.
             */
         }

         /*
          * Threads (apparently) holding locks or that are upcall
          * handlers must be switched to, and therefore remain in
          * the queue.
          */
      }

      /*
       * If we have threads remaining, begin switching to them
       * to allow their locks to be released.
       *
       * Control is passed from one (potentially) lock-holding
       * thread to another when:
       *      - the thread finishes its critical
       *        section (i.e., holds no locks).
       *      - the thread realizes it has been
       *        preempted while trying to acquire
       *        a lock (see lock(), below).
       *      - the thread fails to acquire a spin-
       *        lock after a number of attempts.
       *  Control returns to us after continuing all lock-holding
       *  threads because the last one's next_context points to
       *  active_thread (the thread executing this handler code).
       */
      if (lock_holders_queue not empty)  {
         switch to head of lock_holders_queue;
      }
```

```
      /*
       * At this point, we are back from a pass around the lock_holders_queue.
       * Now, if we are being continued by an upcall handler, we need to allow
       * it to run as well.  If this is the case, then we (active_thread) must
       * be in our parent's lock_holders_queue (note that the ``filtering'' of
       * this queue -- above -- allowed upcall handler threads to remain in
       * the queue).  This ensures that control will eventually return to us,
       * specifically when the parent switches back here via its own
       * lock_holders_queue.  This in turn allows us to get back to
       * processing our lock_holders_queue.
       */
      if (active_thread->parent_upcall) {
         switch to active_thread->parent_upcall;
      }
   } /* end of while lock_holders_queue not empty */


   /* Done -- requeue all threads in our care */
   for each thread in our private runnable_queue {
      move to normal readyq;
   }
}


spin_lock (spin_lock_t lock)
{
   increment self->lock count;

   while (spin_try_lock(lock) != LOCK_ACQUIRED) {
      decrement self->lock_count;

      while (lock is locked) {
         /*
          * We may be preempted while spinning on this lock.  The scheduler
          * will restart us here if it thinks we hold a lock, which could
          * happen either if we really do, or if we were preempted before
          * being able to decrement our lock_count on a failed acquire.
          * In either case, the scheduler stores a thread id in our TCB.
          * So, check to see if this is set, and if it is, switch back
          * to the scheduler-provided thread.
          */
         if (self->next_context != NULL) {
             switch to self->next_context;
         }
      }

      increment self->lock_count for next attempt at lock;
   }
}
```

```
spin_unlock (spin_lock_t lock)
{
   release lock;
   decrement self->lock_count;

   /*
    * If we were preempted while holding a lock, the scheduler will
    * have stored a thread identity in our next_context before
    * resuming us, and expects us to switch to it once we reach the
    * end of our critical section.  So, check to see if this the
    * last lock, and if (so and) necessary switch.
    */

   if (self->lock_count == 0 and self->next_context != NULL) {
      switch to self->next_context;
   }
}
```