Interface and Execution Models in the Fluke Kernel

Bryan Ford Mike Hibler Jay Lepreau Roland McGrath Patrick Tullmann
Department of Computer Science
University of Utah

Technical Report UUCS-98-013
August, 1998

Abstract

We have defined and implemented a new kernel API that makes every exported operation either fully interruptible and restartable, thereby appearing atomic to the user. To achieve interruptibility, all possible states in which a thread may become blocked for a "long" time are completely representable as valid kernel API calls, without needing to retain any kernel internal state.

This API provides important functionality. Since all kernel operations appear atomic, services such as transparent checkpointing and process migration that need access to the complete and consistent state of a process can be implemented by ordinary user-mode processes. Atomic operations also enable applications to provide reliability in a more straightforward manner.

This API also allows novel kernel implementation techniques and evaluation of existing techniques, which we explore in this paper. Our new kernel's single source implements either the "process" or the "interrupt" execution model on both uni- and multiprocessors, depending only on a configuration option affecting a small amount of code. Our kernel structure avoids the major complexities of traditional implementations of the interrupt model, neither requiring ad hoc saving of state, nor limiting the operations (such as demand-paged memory) that can be handled by the kernel. Finally, our interrupt model configuration can support the process model for selected components, with the attendant flexibility benefits.

We report preliminary measurements comparing fully, partially and non-preemptible configurations of both process and interrupt model implementations. We find that the interrupt model has a modest speed edge in some benchmarks, maximum latency varies nearly three orders of magnitude, average latency varies by a factor of six, and memory use favors the interrupt model as expected, but not by a large amount. We find that the overhead for restarting the most costly kernel operation ranges from 2–8%.

1 Introduction

This paper attempts to bring to light an important and useful control-flow property of OS kernel interface semantics that has been neglected in prevailing systems, and to distinguish this *interface* property from the control-flow properties of an OS kernel *implementation*. An essential issue of operating system design and implementation is when and how one thread can block and relinquish control to another, and how the state of a thread suspended by blocking or preemption is represented in the system. This crucially affects both the kernel interface that represents these states to user code, and the fundamental internal organization of the kernel implementation. A central aspect of this internal structure is the execution model in which the kernel handles processor traps, hardware interrupts, and system calls. In the *process model*, which is used by traditional monolithic kernels such as BSD, Linux, and Windows NT, each thread of control in the system has its own kernel stack. In the *interrupt model*, used by systems such as V [7], QNX [14], and Aegis [12], the kernel uses only one kernel stack per *processor*—for typical uniprocessor kernels, just one kernel stack, period. A thread in a process-model kernel retains its kernel stack state when it sleeps, whereas in an interrupt-model kernel threads must manually save any important kernel state before sleeping. This saved kernel state is often known as a *continuation* [10], since it allows the thread to "continue" where it left off.

In this paper we draw attention to the distinction between an interrupt-model *kernel implementation*, which is a kernel that uses only one kernel stack per processor by manually saving implicit kernel state for sleeping threads, and an "atomic" *kernel API*, which is an API designed so that sleeping threads *need* no such implicit kernel state at all. These two kernel properties are related but fall on or-

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269.

Contact information: lepreau@cs.utah.edu. Dept. of Computer Science, 50 S. Central Campus Drive, Rm. 3190, University of Utah, SLC, UT 84112-9205. http://www.cs.utah.edu/projects/flux/.

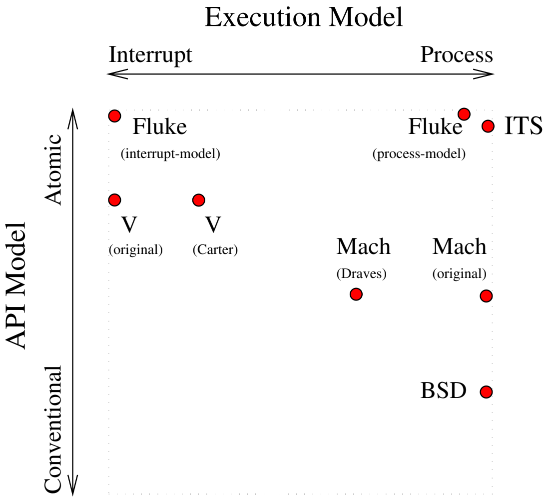

Figure 1: The kernel implementation and API model continuums. V was originally a pure interrupt-model kernel but was later modified to be partly process-model; Mach was a pure process-model kernel later modified to be partly interrupt-model.

thogonal dimensions, as illustrated in Figure 1. In a purely atomic API, *all* possible states in which a thread may sleep for a noticeable amount of time are cleanly visible and exportable to user mode. For example, the state of a thread involved in any system call is always well-defined, complete, and immediately available for examination or modification by other threads; this is true even if the system call is long-running and consists of many stages. In general, this means that all system calls and exception handling mechanisms must be cleanly *interruptible* and *restartable*, in the same way that the instruction sets of modern processor architectures are cleanly interruptible and restartable. For purposes of readability, in the rest of this paper we will refer to API's with these properties as "atomic," as well as the properties themselves.

We have developed a new kernel called Fluke which exports a purely atomic API. This API allows the complete state of any user-mode thread to be examined and modified by other user-mode threads without being arbitrarily delayed. In unpublished work that we were not aware of until very recently, the MIT ITS [11] system implemented an API with a similar property, some 30 years ago. Several other systems came close to providing this property but still had a few situations in which thread state was not always extractable. Supporting a purely atomic API slightly widens the kernel interface due to the need to export additional state information. However, such an API provides the atomicity property that gives important robustness advantages, making it easier to build fault-tolerant systems [24].[1] It also simplifies implementation of a pure interrupt-model kernel by eliminating the need to store implicit state in continuations.

In addition, our kernel supports *both* internal execution models through a build-time configuration option affecting only a small fraction of the source enabling the first "apples-to-apples" comparison between them. Our kernel demonstrates that the two models are not as fundamentally different as they have been considered to be in the past; however, they each have strengths and weaknesses. Some processor architectures have an inherent bias towards the process model—e.g., a 5–10% kernel entry/exit performance difference on the x86. It is also easier to make process-model kernels fully preemptible. Full preemptibility comes at a cost, but this cost is associated with preemptibility, not with the process model itself. Process-model kernels tend to use more per-thread kernel memory, but this is a problem in practice only if the kernel is liberal in its use of stack space and thus requires large kernel stacks, or if the system uses a very large number of threads. Thus, we show that although an atomic API is highly beneficial, the kernel's internal execution model is less important: the interrupt-based organization has a slight size advantage, whereas the process-based organization has somewhat more flexibility.

Finally, contrary to conventional wisdom, our kernel demonstrates that it is practical to use legacy process-model code even within interrupt-model kernels and even on architectures such as the x86 that make it difficult. The key is to run the legacy code in *user mode* but in the *kernel's address space*.

Our key contributions in this work are:

- To present a kernel supporting a pure atomic API and demonstrate the advantages and drawbacks of this approach.
- To explore the relationship between an "atomic API" and the kernel's execution model.
- To present the first "apples-to-apples" comparison between the two kernel *implementation* models using a kernel that supports both, revealing that the models are not as different as commonly believed.
- To show that it is practical to use process-model legacy code in an interrupt-model kernel, and to present several techniques for doing so.

[1] Some examples of this are (i) This property is similar to "chained transactions" [Gray93] which allow a single transaction to progress through intermediate stages while building up state, but is able to rollback only to the last stage. Chained transactions are easier to provide than nested transactions, but yield significant benefits. (ii) It is well known that providing atomicity at lower layers allows higher layers to be written more simply. (iii) ITS exploited an atomic API for a number of properties; it particularly made it easy to write user-mode schedulers, as one could set the state of a thread at *any* time. [4, 3] (iv) Large telecomm applications use an "auditor," a daemon that periodically wakes up and test each of the critical data structures in the system to see if it does not violate some assertions. For critical OS'es one could think of applying the same concept to the kernel. (v) One could both debug and detect deadlock conditions in threads that dive into the kernel, using this property.

The rest of the paper is organized as follows. In Section 2 we look at other systems that have used the interrupt model and have explored the relationships between the interrupt and process model. In Section 3 we define the two models more precisely, and examine the implementation issues in each, looking at the strengths and weaknesses each model brings to a kernel. The atomic API introduced in the previous section is detailed in Section 4. In the 5th section, we present five issues of importance to the execution model of a kernel, with measurements based on different configurations of the same kernel. The final section summarizes our analysis.

2 Related Work

2.1 Interruptibility and Restartability

The clean interruptibility and restartability of *instructions* is now recognized as a vital property of all modern processor architectures. However, this has not always been the case; as Hennessy and Patterson state:

> This last requirement is so difficult that computers are awarded the title *restartable* if they pass that test. That supercomputers and many early microprocessors do not earn that badge of honor illustrates both the difficulty of interrupts and the potential cost in hardware complexity and execution speed. [13]

Since the system calls and other services provided by an operating system appear to user-mode code essentially as an extension of the processor architecture, the OS clearly faces a similar challenge. However, to this point operating systems have rarely met this challenge nearly as thoroughly as processor architectures have: in fact, we have found only one system prior to our own that provides a fully interruptible and restartable API—a system over 30 years old.

For example, the Unix API[28, 15] distinguishes between "short" and "long" operations. "Short" operations such as disk reads are made non-interruptible on the assumption that they will complete quickly enough that the delay will not be noticeable to the application, whereas "long" operations are interruptible but, if interrupted, must be restarted manually by the application. This distinction is arbitrary and has historically been the source of numerous practical problems. The case of disk reads from an NFS server that has gone down is a well-known instance of this problem: the arbitrarily long delays caused by the network makes it no longer appropriate to treat the read operation as "short," but on the other hand these operations cannot simply be changed to "long" and made interruptible because existing applications are not written with the expectation of having to restart file reads.

The Mach API[1] implements I/O operations using IPC; each operation is divided into an RPC-style request and reply stage, and the API is designed so that the operation can be cleanly interrupted after the request has been sent but before the reply has been received. This design reduces but does not eliminate the number of situations in which threads can get stuck in states that aren't cleanly interruptible and restartable. For example, a common remaining case is when a page fault occurs while the kernel is copying the IPC message into or out of the user's address space; the IPC operation cannot be cleanly interrupted and restarted at this point, but handling the page fault may involve arbitrary delays due to communication with other user-mode servers or even across a network. KeyKOS[5] comes very close to solving this problem by limiting all IPC operations to transfer at most one page of data and performing this data transfer atomically; however, in certain corner-case situations it gains promptness by sacrificing correctness.[2] Amoeba[21] allows one user-mode process (or *cluster* in Amoeba terminology) to "freeze" another process for debugging purposes, but processes cannot be frozen in certain situations such as while waiting for an acknowledgement from another network node. V[7, 26] allows one process to examine and modify the state of another, but the retrieved state is incomplete, and state modification is only allowed if the target process is awaiting an IPC reply from the modifying process. The V kernel also contains special support for process migration and checkpointing, which allow the complete state of a process to be saved and reconstructed; however, this state is not made directly available to application code.

The Incompatible Time Sharing (ITS) operating system [11], developed in the 1960s and 1970s at MIT for the DEC PDP-6 and PDP-10 computers, did allow all system calls to be cleanly interrupted and restarted, representing all aspects of a suspended computation in the contents of a thread's user-mode registers. In fact, this property was a central principle of the system's design and substantial effort was made in the implementation to achieve it. We recently learned of an unpublished memo [3] that describes the design and implementation in detail, but no formally published work has previously identified the benefits of an atomic API and explored the implementation issues. The failure of later systems to learn from the experience of this pioneering system is an oversight we hope to rectify.

2.2 Kernel Execution Models

Many existing kernels have been built using either the interrupt or the process model internally: for example, most Unix systems use the process model exclusively, whereas QNX [14] and Aegis [12] use the interrupt model

[2] If the client's data buffer into which an IPC reply is to be received is paged out by a user-mode memory manager at the time the reply is made, the kernel simply discards the reply message rather than allowing the operation to be delayed arbitrarily long by a potentially uncooperative user-mode pager. This usually was not a problem in practice because most paging in the system is handled by the kernel, which is trusted to service paging requests promptly.

exclusively. Other systems such as Taos [20, 23] were designed with a hybrid model where threads often give up their kernel stacks in particular situations but can retain them as needed to simplify the kernel's implementation. Minix [25] used kernel threads to run process-model kernel activities such as device driver code, even though the kernel "core" used the interrupt model. The V kernel [7] was originally organized around a pure interrupt model, but was later adapted by Carter [6] to allow multiple kernel stacks while handling page faults. The Mach 3.0 kernel [1] was taken in the opposite direction: it was originally created in the process model, but Draves [9, 10] later adapted it to use a partial interrupt model by adding continuations in key locations in the kernel and by introducing a "stack handoff" mechanism. However, they did not eliminate all kernel stacks for suspended threads. Draves et al also identified the optimization of *continuation recognition*, which exploits explicit continuations to recognize the computation a suspended thread will perform when resumed, and do part or all of that work by mutating the thread's state without transferring control to the suspended thread's context. But since this information is *not* explicit in the user-mode thread state, there is no way for user code to take advantage of these same optimization techniques in threads examining and coordinating with each other.

The ITS [3] system used the process model of execution, each thread always having a private kernel stack that the kernel switched to and from for normal blocking and preemption. However, the system guaranteed that—when necessary—a thread's state could always be precisely represented by some state of its user-mode registers and a small set of per-thread OS state variables (called "user variables"), whose values had well-defined meanings (such that user code could in fact store a "mid-operation" value at any time, and know what results to expect from "restarting" a complex operation whose earlier stages might not in fact ever have happened). When a thread's exact state needed to be recorded, either because another thread explicitly asked to examine the state, or because the thread incurred a page fault or other exception whose handler must able to inspect and/or restart the faulting operation, any system call in progress would be either promptly finished or backed out to a clean state, updating the registers and user (thread) variables to reflect the progress of the kernel operation. The implementations of system calls were required to register cleanup handlers before calling any potentially-blocking kernel primitive; thereafter, the system call might be interrupted and its context discarded entirely except for running the cleanup handlers. The PC and registers remained at their system call entry state, requiring the system call code or its cleanup handlers to update the PC, registers, and user variables explicitly to reflect partial completion of the operation. The implementation burden of these requirements was eased by the policy that each user memory page touched by system call code was locked in core until the system call completed or was cleaned up and discarded.

We are not aware of any previous kernel that simultaneously supported both the "pure" interrupt model and the "pure" process model through configuration options.

3 The Interrupt and Process Models

An essential feature of operating systems is managing many computations, or threads of control, on a smaller number of processors (often just one). When a thread is suspended either because it blocks awaiting some event or is preempted when the scheduler policy chooses another thread to run, the system must record the suspended thread's state so that it can continue operation later. The way an OS kernel represents the state of suspended threads is a fundamental aspect of its internal structure.

In the "process model," each thread of control in the system has its own kernel stack. When a thread makes a system call or is interrupted, the processor switches to the thread's assigned kernel stack and executes an appropriate handler in the kernel's address space. This handler may at times cause the thread to go to sleep waiting for some event, such as the completion of an I/O request; at these times the kernel may switch to a different thread having its own separate kernel stack state, and then switch back later when the first thread's wait condition is satisfied. The important point is that each thread retains its kernel stack state even while it is sleeping, and therefore has an implicit "execution context" describing what operation it is currently performing. Threads may even hold kernel resources, such as locks or allocated memory regions, as part of this implicit state they retain while sleeping.

An "interrupt-model" kernel, on the other hand, uses only one kernel stack per *processor*—for typical uniprocessor kernels, just one kernel stack, period. This stack only holds state related to the *currently running* thread; no state is stored for sleeping threads other than the state explicitly encoded in its thread control block or equivalent kernel data structure. Context switching from one thread to another involves "unwinding" the kernel stack to the beginning and starting over with an empty stack to service the new thread. In practice, putting a thread to sleep often involves explicitly saving state relating to the thread's operation, such as information about the progress it has made in an I/O operation, in a *continuation* structure. This continuation information allows the thread to "continue" where it left off once it is again awakened. By saving the required portions of the thread's state, it essentially performs the function of the per-thread kernel stack in the process model.

3.1 Kernel Structure vs. Kernel API

The internal thread handling model employed by the kernel is not the only factor in choosing a kernel design. There tends to be a strong correlation between the kernel's execution model and the *kinds* of operations presented by the kernel to application code in the kernel's API. Interrupt-model kernels tend to export short, simple, atomic operations that don't require large, complicated continuations to be saved to keep track of a long running operation's kernel state. Process-model kernels tend to export longer operations with more stages because they are easy to implement given a separate per-thread stack and they allow the kernel to get more work done in one system call. There are exceptions, however; in particular, ITS used one (small, 40 word) [4] stack per thread despite its provision of an atomic API.

Thus, in addition to the execution model of the kernel itself, a distinction can be drawn between an "atomic API," in which kernel operations are designed to be short and simple so that the state associated with long-running activities can be maintained mostly by the application process itself, and a "conventional API," in which operations tend to be longer and more complex and their state is maintained by the kernel invisibly to the application. This stylistic difference between kernel API designs is comparable to the "CISC versus RISC" debates in the area of processor architecture design. However, although there is an obvious relationship between a kernel's internal execution model and its exported API, the exact nature of this relationship has to this point not been well understood.

Fluke, a new microkernel we have designed and implemented, exports a *fully interruptible and restartable ("atomic") API*, in which there are *no* implicit thread states relevant to, but not visible and exportable to application code. Furthermore, its *implementation* can be configured to use either execution model in its pure form (i.e., either exactly one stack per processor or exactly one stack per thread); to our knowledge it is the first kernel to do so. In fact, it is Fluke's atomic API that makes it relatively painless for the kernel to run using either organization: the difference in the kernel code for the two models amounts to only about two hundred assembly language instructions in the system call entry and exit code, and about fifty lines of C in the context switching, exception frame layout, and thread startup code. Notably, this difference is due almost exclusively to dealing with the stacks. The configuration option to select between the two models has no impact on the functionality of the API. The API and implementation model properties of the Fluke kernel and their relationships are discussed in detail in the following sections.

4 Properties of an Atomic API

As mentioned above, the Fluke API is an atomic API, in which all possible thread states relevant to application code are well-defined in the API and are exported to the application. Such an API provides several important and desirable properties, including *prompt* and *correct* exportability of thread state, and full *interruptibility* and *restartability* of system calls and other kernel operations. To illustrate these basic properties, we will contrast the Fluke API with the more conventional APIs of kernels such as Mach and Unix.

4.1 State Exportability

In the Fluke API, any thread can extract, examine, and modify the state of any other thread, assuming that appropriate permission checks are satisfied. The Fluke API requires the kernel to ensure that one thread always be able to manipulate the state of another thread in this way without being held up indefinitely as a result of the target thread's activities or its interactions with other threads in the system. Such state manipulation operations can be delayed in some cases, but only by activities internal to the kernel that do not depend on the promptness of other untrusted application threads; this is the API's *promptness* requirement. For example, if a thread is performing an RPC to a server and is waiting for the server's reply, its state must still be promptly accessible to other threads without delaying the operation until the reply is received.

In addition, the Fluke API requires that, if the state of an application thread is extracted at an arbitrary time by another application thread, and then the target thread is destroyed, re-created from scratch, and reinitialized with the previously extracted state, the new thread must behave indistinguishably from the original, as if it had never been touched in the first place. This is the API's *correctness* requirement.

Fulfilling one or the other of these requirements is fairly easy for a kernel to do, but strictly satisfying both is much more difficult. For example, if promptness is not a requirement, and the target thread is blocked in a system call, then thread manipulation operations on that target can simply be delayed until the system call is completed. This is the approach generally taken by debugging interfaces such as Unix's `ptrace` and `/proc` facilities [28], for which promptness is not a primary concern—e.g., if users are unable to stop and debug a thread because it is involved in a non-interruptible NFS read, they will either just wait for the read to complete or do something to cause it to complete sooner, such as rebooting the server.

Similarly, if correctness is not an absolute requirement, then if one thread tries to extract the state of another at an inconvenient time, the kernel can simply return the thread's "last known" state in hopes that it will be "good enough." This is the approach taken by the Mach 3.0 API, which provides a `thread_abort` to forcibly break a thread out of a system call in order to make its state accessible; this operation is guaranteed to be prompt, but in some cases may affect the state of the target thread so that it will not behave properly if it is ever resumed. OSF later added a `thread_abort_safely` operation [22] which provides correctness, but at the expense of promptness.

Prompt and correct state exportability are required to varying degrees in different situations. For debugging, correctness is critical since the debugger must be able to perform its function without affecting the state of the target thread, but promptness is not as vital since the debugger and target process are under the user's direct control. For conservative garbage collectors which must check an application thread's stack and registers for pointers, correctness is not critical as long as the "last-known" register state of the target thread is available. Promptness, on the other hand, is important because without it the garbage collector could be blocked for an arbitrary length of time, causing resource shortages for other threads, or even deadlock. User-level checkpointing, process migration, dumping, and similar services clearly require correctness, since without it the state of re-created threads may be invalid; promptness is also highly desirable and possibly critical if the risk of being unable to checkpoint or migrate an application for arbitrarily long periods of time is unacceptable. Most programmers have probably encountered promptness or correctness problems in some form on all mainstream operating systems: e.g., the inability to interrupt a networking application under Windows 95, or the occasional situation under Unix where stopping and restarting a process causes it to fail.

4.2 Atomicity and Interruptibility

One natural implication of the Fluke API's promptness and correctness requirements for thread control is that all system calls a thread may make must either be completely *atomic*, or must be cleanly divisible into user-visible atomic stages.

An atomic system call is one that always completes "instantaneously" as far as user code is concerned. If a thread's state is extracted by another thread while the target thread is engaged in an atomic system call, the kernel will either allow the system call to complete, or will transparently abort the system call and roll the target thread back to its original state just before the system call was started. (This contrasts with the Unix and Mach APIs, for example, where user code is responsible for restarting interrupted system calls. In Mach, the restart code is part of the Mach library that normally wraps kernel calls; but there are intermediate states in which system calls cannot be interrupted and restarted, as discussed below.) Because of the promptness requirement, the kernel can only allow a system call to complete if the target thread is *not* waiting for any event produced by some other user-level activity; the system call must be currently running (i.e., on another processor) or it must be waiting on some kernel-internal condition that is guaranteed to be satisfied "soon" without any user-mode involvement. For example, a short, simple operation such as Fluke's equivalent of `getpid()` will always be allowed to run to completion; whereas sleeping operations such as `mutex_lock()` are interrupted and rolled back.

While many Fluke system calls can easily be made atomic in this way, others fundamentally require the presence of intermediate states. For example, there is an IPC system call that a thread can use to send a request message and then wait for a reply. Another thread may attempt to access the thread's state after the request has been sent but before the reply is received; if this happens, the request clearly cannot be "un-sent" because it has probably already been seen by the server; however, the kernel can't wait for the reply either since the server may take arbitrarily long to reply (and may even never reply). Mach addressed this scenario by allowing an IPC operation to be interrupted between the send (request) and receive (reply) operations, later restarting the receive operation from user mode.

A subtler problem is that page faults may occur while transferring IPC messages. Since Fluke IPC doesn't arbitrarily limit the size of IPC messages, faulting IPC operations can't simply be rolled back to the beginning; however, since page faults may be handled by user-mode servers, the kernel cannot hold off all accesses to the faulting thread's state either. In Mach, a page fault mid-transfer in either the sender or the receiver can cause IPC system calls to block for arbitrarily long periods, until the fault is satisfied. Fluke's atomic API allows the kernel to update system call parameters in place in the user-mode registers to reflect the data transferred prior to the fault. While waiting for the fault to be satisfied, both threads are left in well-defined states of having transferred some data and about to restart the IPC to transfer more. The API for Fluke calls is directly analogous to the interface of machine instructions that operate on large ranges of memory, such as the block-move and string instructions on machines such as the x86. The buffer addresses and sizes used by these instructions are stored in registers, and the instructions advance the values in these registers as they work. When the processor takes an interrupt or page fault during a string instruction, the parameter registers in the interrupted processor state have been updated to indicate the memory about to be operated on, and the PC remains at the faulting string instruction. When the fault is resolved, simply jumping to that PC with that register state resumes the string operation in the exact spot it left off.

4.3 Multi-Stage System Calls

The Fluke API handles this problem by breaking long operations such as these into small, atomic *stages*. Entry to each can be completely represented in the thread's user-mode register state. As an example of how this can be done, consider the Unix `read()` system call. If a page fault or other interruption occurs part way through

Type	Examples	Count	Percent
Trivial	`thread_self`	8	7%
Short	`mutex_trylock`	68	64%
Long	`mutex_lock`	8	7%
Multi-stage	`cond_wait`, IPC	23	22%
Total		107	100%

Table 1: Breakdown of the number and types of system calls in the Fluke API. "Trivial" system calls are those that always run to completion without putting the thread to sleep (`thread_self()` is analogous to Unix's `getpid()`). "Short" system calls usually run to completion immediately, but may encounter page faults or other exceptions during processing which causes them to roll back the thread's state. "Long" system calls are those that can be expected to sleep for an extended period of time. "Multi-stage" system calls are those that can sleep indefinitely and can be interrupted at various intermediate points in the operation.

Actual Cause of Exception	Cost to Remedy	Cost to Rollback
Unmapped memory	23536	446
Server Page Fault (Unmapped mem)	26973	1360
Server Page Fault (VTOP translation fault)	5851	496

Table 2: Breakdown of restart costs for various kernel-internal exceptions during a reliable IPC transfer, the area of the kernel with the most internal synchronization (specifically, `ipc_client_connect_send_over_receive()`). The 'Actual Cause' describes the reason the exception was raised; for example a KR_PAGE_FAULT is raised for a virtual to physical translation fault, an unmapped page, and for a page not actually in memory. Note that for the latter two, an IPC to the user-mode memory manager is made to map in the required page. The 'Cost to Rollback' is roughly the amount of work thrown away and redone that did not need to be, while the 'Cost to Remedy' approximates the amount of work needed to service the fault. All costs are in cycles; results were obtained on a 200-Mhz Pentium Pro with the Fluke kernel configured using a process model without kernel thread preemption.

a `read()`, the kernel could adjust the buffer and size parameters on the user's stack according to the amount of data still to be read, changing the user's instruction pointer so that it once again points to the `read()` system call. When the interrupted thread eventually starts executing again, it will automatically restart the system call to read the remainder of the data. Although Unix kernels don't do this[3], this example illustrates how system calls *can* be made atomic.

This is exactly what is done in the Fluke API. Table 1 shows a breakdown of the number and types of system calls in the API. For example, `cond_wait()`, which works as in POSIX `pthreads` [16], must reacquire the condition variable's associated mutex after waiting on the condition (successfully or not). Fluke does this by changing the thread state to point to the `mutex_lock()` system call entrypoint; thus, `mutex_lock()` is the second "stage" of `cond_wait()`.

Except for `region_search`, which can be passed an arbitrarily large region of memory in which to locate kernel objects, all of the other multi-stage calls in the Fluke API are IPC-related. Most of these calls simply represent different options and combinations of the basic send and receive primitives. Although all of these entrypoints could easily be rolled into one, as is often done in other systems such as Mach, the Fluke API's design gives preference to exporting several simple, narrow entrypoints with few parameters rather than one large, complex entrypoint with many parameters. This approach enables the kernel's critical paths to be streamlined by eliminating the need to test for various options. However, the issue of whether system call options are represented as additional parameters or as separate entrypoints is orthogonal to the issue of atomicity and interruptibility; the only difference is that if a multi-stage IPC operation in Fluke is interrupted, the kernel may occasionally modify the user-mode instruction pointer to refer to a different system call entrypoint in addition to updating the other user-mode registers to indicate the amount of data remaining to be transferred.

The implications of providing an atomic API are discussed more fully in [2]. In summary, the purely atomic API greatly facilitates the job of user-level checkpointer, process migrators, and distributed memory systems. The correct, prompt access to all relevant kernel state of any thread in a system makes user-level managers themselves correct and prompt. Additionally, the clean, uniform management of thread state in an atomic API frees the managers from having to detect and handle obscure corner cases. Finally, such an API simplifies the kernel itself and is fundamental to allowing the kernel implementation to use either explicit or implicit continuations to represent blocked threads internally; this factor will be discussed in Section 5.

4.4 Disadvantages of an Atomic API

This discussion reveals several potential disadvantages of an atomic API:

- **Design effort required**: The API must be carefully designed so that all intermediate kernel states in which a thread may have to wait indefinitely can be represented in the explicit user-accessible thread state. Although the Fluke API demonstrates that this can be done, in our experience it does take considerable effort and discipline.

- **API width**: Additional system call entrypoints (or additional options to existing system calls) may be required to represent these intermediate states, effectively widening the kernel's API. For example, in the Fluke API, there are five system calls that are

[3] Instead, they just abort the system call and return EINTR, which has historically been the source of innumerable subtle bugs. Also, the `read()` operation's return value would cause trouble if this operation was to be made transparently restartable; this problem could be fixed by making the return value indicate the *remaining* number of bytes in the buffer *not* read, rather than the number of bytes successfully read.

rarely called directly from user-mode programs, and are instead are usually only used as "restart points" for interrupted kernel operations. However, we have found in practice that although these seldom-used entrypoints are mandated by the fully-interruptible API design, they are also directly useful to some applications; there are no Fluke entrypoints whose purpose is solely to provide a pure interrupt-model API.

- **Thread state size**: Additional user-visible thread state may be required. For example, in Fluke on the x86, due to the shortage of processor registers, two "pseudo-registers" implemented by the kernel are included in the user-visible thread state frame to hold intermediate IPC state. These pseudo-registers add a little more complexity to the API, but they never need to be accessed directly by user code except when saving and restoring thread state, so they do not in practice cause a performance burden. Furthermore, they amount to only two 32-bit words on the x86, and would be unnecessary on most other architectures.

- **Overhead from Restarting Operations**: During some system calls, various events can cause the thread's state to be rolled back, requiring a certain amount of work to be re-done later. Our measurements, summarized in Table 2, show this not to be a significant cost. Application threads rarely access each other's state (e.g., only during the occasional checkpoint or migration), so although it is important for this to be possible, it does not have to be highly efficient. The only other situation in which threads are rolled back is when an exception such as a page fault occurs, and in such cases, the time required to handle the exception invariably dwarfs the time spent re-executing a small piece of system call code later.

- **Architectural bias**: Certain older architectures, such as the 68020/030, make it impossible for the kernel to provide correct and prompt state exportability, because *the processor itself* does not do so. For example, the 68020/030 saved state frame includes some undocumented fields whose contents must be kept unmodified by the kernel; these fields cannot safely be made accessible and modifiable by user-mode software, and therefore a thread's state can never be fully exportable when certain floating-point operations are in progress. However, most other architectures, including the x86 and even other 680x0 processors such as the 68040, do not have this problem.

In practice, none of these disadvantages has caused us significant problems in comparison to the benefits of correct, prompt state exportability.

5 Kernel Execution Models

We now return to the issue of the execution model used in a kernel's *implementation*. Although typically there is a strong correlation between a kernel's API and its internal execution model, in many ways these issues are independent and orthogonal. In this section we report our experiments with Fluke and, previously, with Mach, that demonstrate the following findings.

- **Exported API**: A process-model kernel can easily implement either style of API, but an interrupt-model kernel has a strong "preference" for an atomic API.

- **Preemptibility**: It is easier to make a process-model kernel preemptible, regardless of the API it exports; however, it is easy to make interrupt-model kernels partly preemptible by adding preemption points.

- **Memory use**: Naturally, process-model kernels use more memory because of the larger number of kernel stacks in the system; of course, the size of kernel stacks sometimes can be be reduced to minimize this disadvantage.

- **Architectural bias**: Some architectures, such as the x86 architecture, are fundamentally biased towards the process model, whereas others support both models equally well. CISC architectures tend to be biased because they insist on providing automatic stack handling, whereas RISC architectures usually don't.

- **Legacy code**: Since most existing, robust, easily available OS code, such as device drivers and file systems, is written for the process model, it is easiest to use this legacy code in process-model kernels. However, it is also possible to use this code in interrupt-model kernels with a slight performance penalty.

The following sections discuss these issues in detail and provide concrete measurement results where possible.

5.1 Exported API

One of the most common objections to the interrupt-based execution model is that it requires the kernel to manage explicit continuations. However, our observation is that continuations are not a fundamental property of an interrupt-model kernel, but instead are the symptom of a mismatch between the kernel's API and its implementation. In brief, continuations are only required to implement a conventional API with an interrupt-model kernel; in an interrupt-model kernel exporting an atomic API, the thread's *explicit* user-visible register state acts as the thread's "continuation," holding all the state necessary for the thread to continue where it left off.

```
msg_send_rcv(msg, option, send_size, rcv_size, ...)
{
        rc = msg_send(msg, option, send_size, ...);
        if (rc != SUCCESS)
                return rc;

        rc = msg_rcv(msg, option, rcv_size, ...);
        return rc;
}
```

Figure 2: Example IPC send-receive path in a process-model kernel. Any waiting or fault handling during the operation must keep the kernel stack bound to the current thread.

```
msg_send_rcv(msg, option, send_size, rcv_size, ...)
{
    rc = msg_send(msg, option, send_size, ...);
    if (rc != SUCCESS)
            return rc;

    cur_thread->continuation.msg = msg;
    cur_thread->continuation.option = option;
    cur_thread->continuation.rcv_size = rcv_size;
    ...

    rc = msg_rcv(msg, option, rcv_size, ...,
                 msg_rcv_continue);
    return rc;
}

msg_rcv_continue(cur_thread)
{
    msg = cur_thread->continuation.msg;
    option = cur_thread->continuation.option;
    rcv_size = cur_thread->continuation.rcv_size;
    ...

    rc = msg_rcv(msg, option, rcv_size, ...,
                 msg_rcv_continue);
    return rc;
}
```

Figure 3: Example interrupt-model IPC send-receive path. State defining the "middle" of the send-receive is saved away by the kernel in `msg_send_rcv` in the case that the `msg_rcv` is interrupted. Special code is needed to handle restart from a continuation, `msg_rcv_continue()`.

5.1.1 Continuations

To illustrate this difference, consider the IPC pseudocode fragments in Figures 2, 3, and 4. The first shows a very simplified version of a combined IPC message send-and-receive system call similar to the `mach_msg_trap` system call inside the original process-model Mach 3.0 kernel. The code first calls a subroutine to send a message; if that succeeds, it then calls a second routine to receive a message. If an error occurs in either stage, the entire operation is aborted and the system call finishes by passing a return code back to the user-mode caller. This structure implies that any exceptional conditions that occur along the IPC path that *shouldn't* cause the operation to be completely aborted, such as the need to wait for an incoming message or service a page fault, must be handled completely *within* these subroutines by blocking the current thread while retaining its kernel stack. Once the `msg_send_receive` call returns, the system call is complete.

Figure 3 shows pseudocode for the same IPC path modified to use a partial interrupt-style execution environment, as was done by Draves in the Mach 3.0 continuations work [10, 9]. The first stage of the operation, `msg_send`, is expected to retain the current kernel stack, as above; any page faults or other temporary conditions during this stage must be handled in process-model fashion, without discarding the stack. However, in the common case where the subsequent receive operation must wait for an incoming message, the `msg_rcv` function can discard the kernel stack while waiting. When the wait is satisfied or interrupted, the thread will be given a new kernel stack and the `msg_rcv_continue` function will be called to finish processing the `msg_send_rcv` system call. The original parameters to the system call must be saved explicitly in a continuation structure in the current thread, since they are not retained on the kernel stack.

Note that although this modification partly changes the system call to have an interrupt-model *implementation*, it still retains its conventional *API semantics* as seen by user code. For example, if another thread attempts to examine this thread's state while it is waiting continuation-style for an incoming message, the other thread will either have to wait until the system call is completed, or the system call will have to be aborted, causing loss of state.[4] This is because the thread's continuation structure, including the continuation function pointer itself (pointing to `msg_rcv_continue()`), is part of the thread's logical state but is inaccessible to user code.

5.1.2 Interrupt-Model Kernels Without Continuations

Finally, contrast these first two examples with corresponding code in the style used throughout the Fluke kernel, shown in Figure 4. Although this code at first appears very similar to the code in Figure 2, it has several fundamental differences. First of all, in this environment, system call parameters are generally passed in registers rather than on the stack. The low-level system call entry/exit code does not need to copy parameters from the user's stack to the kernel's; instead, it merely saves the appropriate registers into the thread's control block in a standard format, and the system call handlers take their parameters directly from there. (With the use of simple

[4] In this particular situation in Mach, the `mach_msg_trap` operation gets aborted with a special return code; standard library user-mode code can detect this situation and manually restart the IPC. However, there are many other situations, such as page faults occurring along the IPC path while copying data, which, if aborted, cannot be reliably restarted in this way.

```
msg_send_rcv(cur_thread)
{
        rc = msg_send(cur_thread);
        if (rc != SUCCESS)
                return rc;

        set_pc(cur_thread, msg_rcv_entry);

        rc = msg_rcv(cur_thread);
        if (rc != SUCCESS)
                return rc;

        return 0;
}
```

Figure 4: Example send-receive path for a kernel exporting an atomic API. The `set_pc()` operation effectively serves the same purpose as saving a continuation, using the user-visible register state as the storage area for the continuation. Exposing this state to user mode as part of the API provides the benefits of a purely atomic API and eliminates much of the traditional complexity of continuations. The kernel never needs to save parameters or other continuation state on entry because it's already in the thread's user-mode register state.

preprocessor macros or inline functions, this does not necessarily introduce significant machine dependencies into otherwise machine-independent code.) Second, when an internal system call handler returns a nonzero result code, the system call exit layer does *not* simply complete the system call and pass this result code back to the user. Instead, it leaves the user's program counter pointing *just before* the instruction causing the system call, then passes the result code to an exception handling routine in the kernel. Thus, return values in the kernel are only used for *kernel-internal* exception processing that are intended to be transparent to the user; results intended to be seen by user code are returned by modifying the thread's saved user-mode register state. Finally, if the `msg_send` stage in `msg_send_rcv` completes successfully, then before proceeding with the `msg_rcv` stage, the kernel updates the user-mode program counter to point to the user-mode system call entrypoint for `msg_rcv`. This way, if the `msg_rcv` must wait or encounters a page fault, it can simply return a nonzero (kernel-internal) result code, and the thread's user-mode register state will be left so that when normal processing is eventually resumed, the `msg_rcv` system call will automatically be invoked with the appropriate parameters to finish the IPC operation.

The upshot of this is that in the Fluke kernel, the thread's explicit user-mode register state acts as the "continuation," allowing the kernel stack to be thrown away or reused by another thread if the system call must wait or handle an exception. Since this state is *explicit* and fully visible to user-mode code, it can be exported at any time to other threads, thereby providing the promptness and correctness properties required by the atomic API. Furthermore, this atomic API in turn simplifies the interrupt-model kernel implementation to the point of being almost as simple and clear as the original process-model code in Figure 2.

Model	Kernel Preemption	Locking
Process	None	None
Process	Partial	None
Process	Full	Mutex locks
Interrupt	None	None
Interrupt	Partial	None

Table 3: Characteristics of different Fluke kernel configurations measured. Shown for each are the execution model of the kernel (Process or Interrupt), the availability of kernel preemption (None, Partial, or Full) and the type of locking implied.

5.2 Preemptibility

Although the use of an atomic API greatly reduces the kernel complexity and inconvenience burden traditionally associated with interrupt-model kernels, there are other relevant factors as well, such as kernel preemptibility. Low preemption latency is a desirable kernel characteristic, and is critical in real-time systems and in microkernels such as L3 [18] and VSTa [27] that dispatch hardware interrupts to device drivers running as ordinary threads (in which case preemption latency effectively becomes interrupt-handling latency). Since preemption can generally occur at any time while running in user mode, it is the kernel itself that causes preemption latencies that are greater than the hardware minimum.

In a process-model kernel that already supports multiprocessors, it is often relatively straightforward to make most of the kernel preemptible by changing spin locks into blocking locks (e.g., mutexes). Of course, a certain core component of the kernel, which implements scheduling and preemption itself, must still remain nonpreemptible. Implementing kernel preemptibility in this manner fundamentally relies on kernel stacks being retained by preempted threads, so it clearly would not work in a pure interrupt-model kernel. The Fluke kernel, besides supporting both the interrupt and process models, is optionally configurable to support this form of kernel preemptibility in the process model.

Even in an interrupt-model kernel, important parts of the kernel can often be made preemptible as long as preemption is done in a carefully controlled way. For example, in microkernels that rely heavily on IPC, many long-running kernel operations tend to be IPCs that copy data from one process to another. It is relatively easy to introduce *preemption points* in select locations such as on the data copy path. Besides supporting full kernel preemptibility in the process model, the Fluke kernel also supports partial preemptibility in this way in either execution model. QNX [14] is an example of another existing interrupt-model kernel whose IPC path is made preemptible in this fashion.

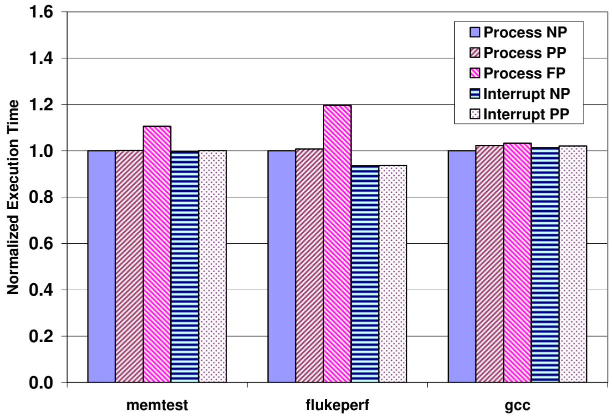

Figure 5: Performance of the Fluke kernel in different configurations. For each application, the execution time is normalized to the performance of the process-model kernel without kernel preemption (Process NP), the first bar in each group. The remaining bars are, left to right, process-model with partial kernel preemption (Process PP), process-model with full kernel preemption (Process FP), interrupt-model without kernel preemption (Interrupt NP), and interrupt-model with partial kernel preemption (Interrupt PP).

5.3 Performance of Different Configurations

The Fluke kernel supports a variety of build-time configuration options that control the execution model of the kernel; by comparing different configurations of the same kernel, we can analyze the properties of these different execution models. We explore kernel configurations along two axes: interrupt versus process model and full versus partial (explicit preemption points) versus no preemption. [5] Since full kernel preemptibility is incompatible with the interrupt model, there are effectively five possible configurations, summarized in Table 3.

Figure 5 shows the relative performance of various applications on the Fluke kernel under various kernel configurations. For each application, the execution times for all kernel configurations are normalized to the execution time of that application on the "base" configuration: process model with no kernel preemption. The non-fully-preemptible kernels were run both with and without partial preemption support on the IPC path. All tests were run on a 200MHz Pentium Pro PC with 256KB L2 cache and 64MB of memory. The applications measured include:

- *Perftest* runs a series of tests to time various synchronization and IPC primitives. It performs a large number of kernel calls and context switches.

- *Memtest* accesses 16MB of memory one byte at a time sequentially. Memtest runs under a memory manager which allocates memory on demand, exercising kernel fault handling and the exception IPC facility.

- *Gcc* compiles a single .c file. This test include running the front end, the C preprocessor, C compiler, assembler and linker to produce a runnable Fluke binary.

[5] Additionally, Fluke supports another axis, multi- versus uniprocessor configurations, that are omitted for brevity.

Test	flukeperf			
	latency		schedules	
	ave	max	run	miss
`Process FP`	5.14	19.6	9212	0
`Process PP`	18.0	1200	7805	5
`Process NP`	28.9	7430	7594	132
`Interrupt PP`	18.7	1272	7531	7
`Interrupt NP`	30.4	7356	7348	141

Table 4: Effect of execution model on preemption latency. We measure the average and maximum time (μs) required for a periodic high-priority kernel thread to start running after being scheduled, while competing with lower-priority application threads. Also shown is the number of times the kernel threads runs over the lifetime of the application and the number of times it failed to complete before the next scheduling interval.

As expected, non-fully-preemptible kernels perform better than the fully-preemptible equivalents since they include no locking overhead. The interrupt and process model kernels are nearly identical in performance except for the *perftest* case. In *perftest* we are seeing a positive effect of using a single processor stack: better cache locality on context switches.

To measure the effect of the execution model of preemption latency, we introduce a high-priority kernel thread which is scheduled every millisecond, and record its observed preemption latencies during a run of the *flukeperf* application from the previous graph. *Flukeperf* is used because it performs a number of large, long running IPC operations ideal for inducing preemption latencies. Table 4 summarizes the experiment. The first two columns are the average and maximum observed latency in microseconds. The last two columns of the table show the number of times the thread ran over the course of the application and the number of times it could not be scheduled because it was still running or queued from the previous interval. As expected, the fully-preemptible (FP) kernel permits much smaller and predictable latencies and allowed the high-priority thread to run without missing an event. The non-preemptible (NP) kernel configuration exhibits highly variable latency for both the process and interrupt model causing a large number of missed events. Though we implement only a single explicit preemption point on the IPC data copy path, the partial preemption (PP) configuration fares well on this benchmark. This is not surprising given that it performs a number of large IPC operations.

5.4 Memory Use

Traditionally, one of the primary perceived benefits of the interrupt model is the memory saved by having only one kernel stack per processor rather than one per thread. For example, Mach's average per-thread kernel mem-

System	Execution Model	TCB Size	Stack Size	Total Size	Procs	Mem. Used
FreeBSD	Process	2132	6700	8832	5	44K
Linux	Process	2395	4096	6491	5	32K
Mach	Process	452	4022	4474	N/A	
Mach	Interrupt	690	—	690	N/A	
L3	Process	1024		1024	N/A	
Fluke	Process	4096		4096	19	76K
Fluke	Process	1024		1024	19	19K
Fluke	Interrupt	300	—	300	19	6K

Table 5: Comparison of the kernel model of various existing systems and the overhead due to thread/process management. TCB, stack, and total sizes are reported in bytes. The 'Procs' column lists the number of processes or threads (in the case of Fluke) to run a minimal system. The 'Memory Used' column indicates roughly the amount of kernel memory given to these processes and threads.

ory overhead was reduced by 85% when the kernel was changed to use a partial interrupt model [9, 10]. Of course, the overall memory used in a system for thread management overhead depends not only on whether each thread has its own kernel stack, but also on how big these kernel stacks are and how many threads are generally used in a realistic system.

To provide an idea of how these factors add up in practice, we show in Table 5 memory usage measurements gathered from a number of different systems and configurations. The Mach figures are as reported in [9]: the process-model numbers are from MK32, an earlier version of the Mach kernel, whereas the interrupt-model numbers are from MK40. The L3 figures are as reported in [19]. For Fluke, we show three different rows: two for the process model using two different stack sizes, and one for the interrupt model.

The two process-model stack sizes for Fluke bear special attention. The smaller 1K stack size is sufficient only in the "production" kernel configuration which leaves out various kernel debugging features, and only when the device drivers do not run on these kernel stacks. Section 5.6 will describe Fluke's device driver support in more detail; however, the important point for now is that the device drivers we use are borrowed from legacy systems and are considerably more stack-hungry than the kernel itself.

To summarize these results, although it is true that interrupt-model kernels tend to minimize kernel thread memory use most effectively, at least for modest numbers of active threads, much of this reduction can also be achieved in process-model kernels simply by structuring the kernel to avoid excessive stack requirements. At least on the x86 architecture, as long as the thread management overhead is about 1K or less per thread, there appears to be no great difference between the two models for modest numbers of threads. However, real production systems may need larger stacks and also may want to have them be a multiple of the page size in order to use a "red zone." These results should apply to other architectures just as well, though the basic sizes may be scaled by an architecture-specific factor. For all but power-constrained systems, the memory differences are probably in the noise.

5.5 Architectural Bias

Besides the more fundamental advantages and disadvantages of each model as discussed above, in some cases there are advantages to one model artificially caused by the design of the underlying processor architecture. In particular, traditional CISC architectures, such as the x86 and 680x0, tend to be biased somewhat toward the process model and make the kernel programmer jump through various hoops to write an interrupt-model kernel. With a few exceptions, more recent RISC architectures tend to be fairly unbiased, allowing either model to be implemented with equal ease and efficiency.

Unsurprisingly, the architectural property that causes this bias is the presence of automatic stack management and stack switching performed by the processor. For example, when the processor enters supervisor mode on the x86, it automatically loads the new supervisor-mode stack pointer, and then pushes the user-mode stack pointer, instruction pointer (program counter), and possibly several other registers onto this supervisor-mode stack. Thus, the processor automatically *assumes* that the kernel stack is associated with the current thread. To build an interrupt-model kernel on such a "process-model architecture," the kernel must either copy this data on kernel entry from the per-processor stack to the appropriate thread control block, or it must keep a separate, "minimal" process-model stack as part of each thread control block, which is the stack the processor switches to on kernel entry, and then switch to the "real" kernel stack just after entry. Fluke in its interrupt-model configuration uses the former technique, while Mach uses the latter.

Most RISC processors, on the other hand, including the MIPS, PA-RISC, and PowerPC, use "shadow registers" for exception and interrupt handling rather than explicitly supporting stack switching in hardware. When an interrupt or exception occurs, the processor merely saves off the original user-mode registers in special one-of-a-kind shadow registers, and then disables further interrupts until they are explicitly re-enabled by software. If the OS wants to support nested exceptions or interrupts, it must then store these registers on the stack itself; it is generally just as easy for the OS to save them on a per-processor interrupt-model stack as it is to save them on a per-thread process-model stack. A notable exception among RISC processors is the SPARC, with its stack-based register window feature.

To examine the effect of architectural bias on the x86, we compared the performance of the interrupt and process-model Fluke kernels in otherwise completely equivalent configurations (using no kernel preemption). On a

100MHz Pentium CPU, the additional trap and system call overhead introduced in the interrupt-model kernel by moving the saved state from the kernel stack to the thread structure on entry, and back again on exit, amounts to about six cycles (60ns). In contrast, the minimal hardware-mandated cost of entering and leaving supervisor mode is about 70 cycles on this processor. Therefore, even for the fastest possible system call the interrupt-model overhead is less than 10%, and for realistic system calls is in the noise. We conclude that although this architectural bias is a significant factor in terms of programming convenience, and may be important if it is necessary to "squeeze every last cycle" out of a critical path, it is probably not a major performance concern in general.

5.6 Legacy Code

One of the most important practical concerns with an interrupt-based kernel execution model is that it appears to be impossible to use pre-existing legacy code, borrowed from process-model systems such as BSD or Linux, in an interrupt-model kernel, such as the Exokernel [12] and the CacheKernel [8]. For example, especially on the x86 architecture, it is impractical for any small programming team to write device drivers for any significant fraction of the commonly available PC hardware; they must either borrow drivers from existing systems, or support only a bare minimum set of hardware configurations. The situation is similar, though not as severe, for other types of legacy code such as file systems or TCP/IP protocol stacks.

There are a number of reasonable approaches to incorporating process-model legacy code into interrupt-model kernels. For example, if kernel threads are available (threads that run in the kernel but are otherwise ordinary process-model threads), process-model code can be run on these threads when necessary. This is the method Minix [25] uses to run device driver code. Unfortunately, kernel threads can be difficult to implement in interrupt-model kernels, and can introduce additional overhead on the kernel entry/exit paths, especially on architectures with the process-model bias discussed above. This is because such processors behave differently in a trap or interrupt depending on whether the interrupted code was in user or supervisor mode [17]; therefore each trap or interrupt handler in the kernel must now determine whether the interrupted code was a user thread, a process-model kernel thread, or the interrupt-model "core" kernel itself, and react appropriately in each case. In addition, the process-model stacks of kernel threads on these architectures can't easily be pageable or dynamically growable, because the processor depends on always being able to push saved state onto the kernel stack if a trap occurs. Ironically, on RISC processors that have no bias towards the process model, it is much easier to implement process-model kernel threads in an interrupt-model kernel.

As an alternative to supporting kernel threads, the kernel can instead use only a *partial* interrupt model, in which kernel stacks are usually handed off to the next thread when a thread blocks, but can be retained while executing process-model code. This is the method that Mach with continuations [10] uses. Unfortunately, this approach brings with it a whole new set of complexities and inefficiencies, largely caused by the need to manage kernel stacks as first-class kernel objects independent of and separable from both threads and processors.

The Fluke kernel uses a different approach, which keeps the "core" interrupt-model kernel simple and uncluttered while effectively supporting something almost equivalent to kernel threads. Basically, the idea is to run process-model "kernel" threads in user *mode* but in the kernel's *address space*. In other words, these threads run in the processor's unprivileged execution mode, and thus run on their own user stacks separate from the kernel's stack; however, the address translation hardware is set up so that while these threads are executing, their view of memory is effectively the same as it is for the "core" interrupt-model kernel itself. This allows the core kernel to treat these process-level activities just like any other user-level activities, which run in a separate address space from the other user-level address spaces; but this particular address space is just set up a little differently.

There are three main issues with this approach. The first is that these user-level pseudo-kernel threads may need to perform *privileged operations* occasionally, for example to enable or disable interrupts or access device registers. In the x86 this isn't a problem because user-level threads can be given direct access to these facilities simply by setting some processor flag bits associated with those threads; however, on other architectures these operations may need to be "exported" from the core kernel as pseudo-system calls only available to these special pseudo-kernel threads. Second, these user-level activities may need to *share data structures* with the core kernel to perform operations such as allocating kernel memory or installing interrupt handlers; since these threads are treated as normal user-mode threads, they are probably fully preemptible and do not share the same constrained execution environment as the core kernel. Again, a straightforward solution, which is what Fluke does, is to "export" the necessary facilities through a special system call that allows these special threads to temporarily jump into supervisor mode and the kernel's execution environment, perform some arbitrary (nonblocking) activity, and then return to user mode. The third issue is the *cost* of performing this extra mode switching; our calculations indicate that this cost is negligible, [but we will measure it to be sure.]

6 Conclusion

In this paper, we have explored in depth the differences between the interrupt and process models and presented a number of ideas, insights, and results. Our Fluke kernel demonstrates that, contrary to conventional wisdom, the need for the kernel to manually save state in *continuations* is not a fundamental property of the interrupt model, but instead is a symptom of a mismatch between the kernel's implementation and its API. Our kernel is only the second to export a purely "atomic" API, in which all kernel operations are fully interruptible and restartable; this property has important benefits for fault-tolerance and for applications such as user-mode process migration, checkpointing, and garbage collection, and eliminates the need for interrupt-model kernels to manually save and restore continuations. Using our configurable kernel which supports both the interrupt-based and process-based execution models, we have made an "apples-to-apples" comparison between the two execution models. As expected, the interrupt-model kernel requires less per-thread memory. Although a null system call entails a 5–10% higher overhead on an interrupt-model kernel due to a built-in bias toward the process model in common processor architectures such as the x86, the interrupt-model kernel exhibits a modest performance advantage in some cases, although it can incur vastly higher latencies. Our conclusion is that it is highly desirable for a kernel to present an atomic API such as Fluke's, but that for the kernel's internal execution model, either implementation model is reasonable.

Acknowledgements

We are grateful to Alan Bawden and Mootaz Elnozahy for interesting and enlightening discussion concerning these issues and their implications for reliability, Kevin Van Maren for elucidating and writing up notes on other aspects of the kernel's execution model, John Carter and anonymous reviewers for comments on earlier drafts of this paper, and Linus Kamb for help with the use of certain system calls.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] Anonymous. User-level Checkpointing Through Exportable Kernel State. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 85–88, Seattle, WA, Oct. 1996. IEEE.

[3] A. Bawden. PCLSRing: Keeping Process State Modular. Unpublished report. ftp://ftp.ai.mit.edu/pub/alan/-pclsr.memo, 1989.

[4] A. Bawden. Personal Communication, Aug. 1998.

[5] A. C. Bomberger and N. Hardy. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, Apr. 1992.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 152–164, Asilomar, CA, Oct. 1991.

[7] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[8] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Assoc., Nov. 1994.

[9] R. P. Draves. *Control Transfer in Operating System Kernels*. PhD thesis, Carnegie Mellon University, May 1994.

[10] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, Asilomar, CA, Oct. 1991.

[11] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. ITS 1.5 Reference Manual. Memo 161a, MIT AI Lab, July 1969.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1989.

[14] D. Hildebrand. An Architectural Overview of QNX. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, Apr. 1992.

[15] Institute of Electrical and Electronics Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, 1994. Std 1003.1-1990.

[16] Institute of Electrical and Electronics Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, 1996. Std 1003.1, 1996 Edition.

[17] Intel Corporation. *Pentium Processor User's Manual*, volume 3. Intel, 1993.

[18] J. Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, Dec. 1993.

[19] J. Liedtke. A Short Note on Small Virtually-Addressed Control Blocks. *Operating Systems Review*, 29(3):31–34, July 1995.

[20] P. R. McJones and G. F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. In *Proceedings of the Winter 1989 USENIX Technical Conference*, pages 393–404, San Diego, CA, Feb. 1989. USENIX.

[21] S. J. Mullender. Process Management in a Distributed Operating System. In J. Nehmer, editor, *Experiences with Distributed Systems*, volume 309 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[22] Open Software Foundation and Carnegie Mellon Univ. *OSF MACH Kernel Principles*, 1993.

[23] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, Feb. 1990.

[24] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.

[25] A. S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[26] V-System Development Group. *V-System 6.0 Reference Manual*. Computer Systems Laboratory, Stanford University, May 1986.

[27] A. Valencia. An Overview of the VSTa Microkernel. http://-www.igcom.net/~jeske/VSTa/vsta_intro.html.

[28] X/Open Company Ltd., Berkshire, UK. *CAE Specification: System Interfaces and Headers, Issue 4, Version 2*, Sept. 1994.