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

Abstract

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

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 us to explore novel kernel implementation techniques and to evaluate existing techniques. The Fluke kernel's single source implements either the "process" or the "interrupt" execution model on both uniprocessors and multiprocessors, depending on a configuration option affecting a small amount of code.

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 performance advantage in some benchmarks, maximum preemption latency varies nearly three orders of magnitude, average preemption 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% of the cost of the operation.

1 Introduction

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 in-kernel events such as 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—the complete state of a thread is implicitly encoded its stack. In the *interrupt model*, used by systems such as V [8], QNX [19], and the exokernel implementations [13, 22], the kernel uses only one kernel stack per *processor*—thus for typical uniprocessor configurations, only one kernel stack. A thread in a process-model kernel retains its kernel stack state when it sleeps, whereas in an interrupt-model kernel threads must explicitly save any important kernel state before sleeping, since there is no stack implicitly encoding the state. This saved kernel state is often known as a *continuation* [11], 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*—a kernel that uses only one kernel stack per processor, explicitly saving important kernel state before sleeping—and an "atomic" *kernel API*—a system call API designed to eliminate implicit kernel state. These two kernel properties are related but fall on orthogonal 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 as a kernel entrypoint. 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 requires all system calls and exception handling mechanisms to 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 an API with these properties as "atomic." We use this term because, from the user's perspective, no thread is ever in the middle of any system call.

We have developed a kernel, Fluke [16], 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 arbi-

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

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

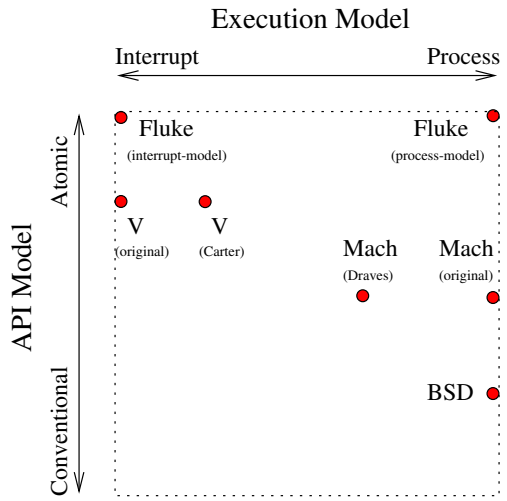

Figure 1: The kernel execution and API model continuums. V was originally a pure interrupt-model kernel but was later modified to be partly process-model; Mach was a pure process-model kernel later modified to be partly interrupt-model. Fluke supports either execution model via compile time options.

trarily delayed. Supporting a purely atomic API slightly increases the width and complexity of the kernel interface but provides important benefits to user-level applications in terms of the power, flexibility, and predictability of system calls.

In addition, Fluke supports *both* the interrupt and process execution models through a build-time configuration option affecting only a small fraction of the source, enabling a comparison between them. Fluke demonstrates that the two models are not necessarily as different as they have been considered to be in the past; however, they each have strengths and weaknesses. Some processor architectures inherently favor the process model and process model kernels are easier to make fully preemptible. Although full preemptibility comes at a cost, this cost is associated with preemptibility, not with the process model itself. Process model kernels use more per-thread kernel memory, but this is unlikely to be a problem in practice except for power-constrained systems. We show that while an atomic API is beneficial, the kernel's internal execution model is less important: an interrupt-based organization has a slight size advantage, whereas a 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. 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: (i) To present a kernel supporting a pure atomic API and demonstrate the advantages and drawbacks of this approach. (ii) To explore the relationship between an "atomic API" and the kernel's execution model. (iii) To present the first comparison between the two kernel *implementation* models using a kernel that supports both in pure form, revealing that the models are not necessarily as different as commonly believed. (iv) 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.

The rest of the paper is organized as follows. In Section 2 we look at other systems, both in terms of the "atomicity" their API and in terms their execution models. 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. Fluke's atomic API is detailed in Section 4. In the following section, we present six 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

Related work is grouped into kernels with atomic or near-atomic system call APIs and work related to kernel execution models.

2.1 Atomic System Call API

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. [18]

Since kernel system calls appear to user-mode code essentially as an extension of the processor architecture, the OS clearly faces a similar challenge. However, few operating systems have met this challenge nearly as thoroughly as processor architectures have.

For example, the Unix API [20] 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 inappropriate 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 are not 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 [6] 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.[1] Amoeba [26] 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 [8, 32] 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.

In scheduler activations [2], user threads have no kernel state at all when they are neither running on a processor in user-mode, nor blocked in the kernel on a system call or page fault. However, threads blocked in system calls have complex state that is represented by a "scheduler activation" kernel object just as it would be by a kernel thread object; that state is not available to the user.

The Cache Kernel [9] and the Aegis [13] and Xok [22] exokernels implement atomic kernel interfaces by restricting the kernel API to managing extremely low-level abstractions so that none of the kernel system calls ever have any reason to block, and therefore they avoid the need for handling restarts or interruptions. Although this is a perfectly reasonable and compelling design strategy, the somewhat higher-level Fluke API demonstrates that strict atomicity can be implemented even in the presence of more complex operations which are not inherently idempotent.

On the other side of the spectrum, the Incompatible Time Sharing (ITS) operating system [12], developed in the 1960s and 1970s at MIT for the DEC PDP-6 and PDP-10 computers, demonstrated the feasibility of implementing a fully atomic API at a much higher levels of abstraction than Fluke, implementing facilities such as process control, file systems, and networking. ITS allowed 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. An unpublished memo [4] describes the design and implementation in detail, though to our knowledge no formally published work has previously identified the benefits of an atomic API and explored the implementation issues.

There are several systems which use concepts similar to Fluke's atomic system call API in different areas of operating systems. Quicksilver [17] and Nonstop [3] are transactional operating systems; in both of these systems, the kernel provides primitives for maintaining transactional semantics in a distributed system. In this way, transactional semantics are provided for high-level services such as file operations even though the basic kernel operations on which they are built may not be. The VINO [29] kernel uses transactions to maintain system integrity when executing untrusted software extensions downloaded into the kernel. These transactions make graft invocations appear atomic, even though invocations of the basic kernel API are not wrapped in transactions.

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 [19], the Cache Kenrel, and the exokernels use the interrupt model exclusively. Other systems such as Firefly's Taos [25, 28] 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 [30] 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 was originally organized around a pure interrupt model, but was later adapted by Carter [7] 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 [10, 11] 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, not all kernel stacks for suspended

[1] 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.

threads were eliminated. 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. Though it was used to good effect within the kernel, user-mode code could not take advantage of this optimization technique because the continuation information was not available to the user. In Fluke, the continuation is explicit in the user-mode thread state, giving the user a full, well-defined picture of the thread's state.

The ITS 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. To ensure the API's atomicity guarantee, implementations of system calls were required to explicitly update the user register state to reflect partial completion of the operation, or to register special cleanup handlers to do so for a system call interrupted during a block. Once a system call blocked, an interruption would discard all context except the user registers,[2] and run these special cleanup handlers. 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 a compile-time configuration option.

3 The Interrupt and Process Models

An essential feature of operating systems is managing many computations on a smaller number of processors, typically just one. Each computation is represented in the OS by a thread of control. 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. 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 but without the overhead of a full kernel stack.

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) stack per thread despite its provision of an atomic API. [5]

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

[2] Each ITS thread (or "job") also had a small set of "user variables" that acted as extra "pseudo-registers" containing additional parameters for certain system calls. Fluke uses a similar mechanism on the x86 because it has so few registers.

analogous to the "CISC to RISC" shift in processor architecture design, in which complex, powerful operations are broken into a series of simpler instructions with more state exposed through a wider register file.

Fluke 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, Fluke's implementation can be configured at compile-time 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 simple 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. This code deals almost exclusively with stack handling. The configuration option to select between the two models has no impact on the functionality of the API. Note that the current implementation of Fluke is not highly optimized, and more extensive optimization would naturally tend to interfere with this configurability since many obvious optimizations depend on one execution model or the other: e.g., the process model implementation could avoid rolling back and restarting in certain cases, whereas the interrupt model implementation could avoid returning through layers of function calls by simply truncating the stack on context switches. However, similar optimizations generally apply in either case even though they may manifest differently depending on the model, so we believe that despite this caveat, Fluke still provides a valuable testbed for analyzing these models. 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

An atomic API provides four important and desirable properties: *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 the Mach and Unix kernels.

4.1 Promptness and Correctness

The Fluke system call API supports the extraction, examination, and modification of the state of any thread by any other thread (assuming the requisite access 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 only one of the promptness and correctness requirements is fairly easy for a kernel to do, but strictly satisfying both is 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 [20], for which promptness is not a primary concern—e.g., if users are unable to stop or debug a process 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 NFS server.

Similarly, if correctness is not an absolute requirement, then if one thread tries to extract the state of another thread at an inconvenient time, the kernel can simply return the target 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` system call 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. To support process migration, the OSF later added a `thread_abort_safely` system call [27] which provides correctness, but at the expense of promptness.

Prompt and correct state exportability are required to varying degrees in different situations. For process control or debugging, correctness is critical since the target thread's state must not be damaged or lost; promptness is not as vital since the debugger and target process are under the user's direct control, but a lack of promptness is often perceptible and causes confusion and annoyance to the user. 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, 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. Mission critical systems often employ an "auditor" daemon which periodically wakes up and tests each of the system's critical threads and data structures for integrity, which is clearly only possible if a correct snapshot of the system's state is available without unbounded delay. Common user-mode deadlock detection and recovery mechanisms similarly depend on examination of the state of other threads. In short, although real operating systems often get away with providing unpredictable or not-quite-correct thread control semantics, in general promptness and correctness are highly desirable properties.

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 appear 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 Fluke 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. 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 more subtle problem is page faults that may occur while transferring IPC messages. Since Fluke IPC does not arbitrarily limit the size of IPC messages, faulting IPC operations cannot simply be rolled back to the beginning. Additionally, the kernel cannot hold off all accesses to the faulting thread's state, since page faults may be handled by user-mode servers. In Mach, a page fault during an IPC transfer can cause the system call to block until the fault is satisfied (an arbitrarily long period).

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. Thus, while waiting for the fault to be satisfied both threads are left in the well-defined state of having transferred some data and about to start an IPC to transfer more. The API for Fluke system 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 Intel x86 [21]. 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 program counter remains at the faulting string instruction. When the fault is resolved, simply jumping to that program counter with that register state resumes the string operation in the exact spot it left off.

Table 1 breaks the Fluke system call API into four categories, based on the potential length of each system call. "Trivial" system calls are those that will always run to completion without putting the thread to sleep. For example, Fluke's `thread_self` (analogous to Unix's `getpid`) will always fetch the current thread's identifier without blocking. "Short" system calls usually run to completion immediately, but may encounter page faults or other exceptions during processing. If an exception happens then the system call will roll back and restart. "Long" system calls are those that can be expected to sleep for an extended period of time (e.g., waiting on a

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 always run to completion. "Short" system calls usually run to completion immediately, but may roll back. "Long" system calls can be expected to sleep indefinitely. "Multi-stage" system calls can be interrupted at intermediate points in the operation.

condition variable). "Multi-stage" system calls are those that can cause the calling thread to sleep indefinitely and can be interrupted at various intermediate points in the operation.

Except for cond_wait and region_search—a system call which can be passed an arbitrarily large region of memory—all of the 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 done in 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.

In [31] we more fully discuss the consequences of providing an atomic API. In summary, the purely atomic API greatly facilitates the job of user-level checkpointers, 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 easily to use either the process or the interrupt model; this factor will be discussed in Section 5.

Object	**Description**
Mutex	A kernel-supported mutex which is safe for sharing between processes.
Cond	A kernel-supported condition variable.
Mapping	Encapsulates an imported region of memory; associated with a Space (destination) and Region (source).
Region	Encapsulates an exportable region of memory; associated with a Space.
Port	Server-side endpoint of an IPC.
Portset	A set of Ports on which a server thread waits.
Space	Associates memory and threads.
Thread	A thread of control, associated with a Space.
Reference	A cross-process handle on a Mapping, Region, Port, Thread or Space. Most often used as a handle on a Port that is used for initiating client-side IPC.

Table 2: The primitive object types exported by the Fluke kernel.

4.3 Examples from Fluke

The Fluke kernel directly supports nine primitive object types, listed in Table 2. All types support a common set of operations including create, destroy, "rename," "point-a-reference-at," "get_objstate," and "set_objstate." Obviously, each type also supports operations specific to its nature; for example, a Mutex supports lock and unlock operations (the complete API is documented in [15]). The majority of kernel operations are transparently restartable. For example, port_reference, a "short" system call, takes a Port object and a Reference object and "points" the reference at the port. If either object is not currently mapped into memory[3], a page fault IPC will be generated by the kernel after which the reference extraction will be restarted. In all such cases page faults are generated very early in the system call, so little work is thrown away and redone.

Simple operations that restart after an error are fairly uninteresting. The more interesting ones are those that update their parameters, or even the system call entry point, to record partial success of the operation. The simplest example of this is the cond_wait operation which atomically blocks on a condition variable, releasing the associated mutex, and, when the calling thread is woken, reacquires the mutex. In Fluke, this operation is broken into two stages: the cond_wait portion and the mutex_lock. To represent this, before a thread sleeps on the condition variable, the thread's user-mode instruction pointer is adjusted to point at the mutex_lock entry point, and the mutex argument is put into the appropriate

[3]In Fluke, kernel objects are mapped into the address space of an application with the virtual address serving as the "handle" and memory protections providing access control.

register for the new entrypoint. Thus, if the thread is interrupted or awoken it will automatically retry the mutex lock and not the whole condition variable wait.

An example of a system call that updates its parameters is the `ipc_client_send` system call, which sends data on an already established IPC connection to a waiting server thread. The call might either be the result of an explicit invocation by user code, or its invocation could have been caused implicitly by the kernel due to the earlier interruption of a longer operation such as `ipc_client_connect_send`, which establishes a new IPC connection and then sends data across it. Regardless of how `ipc_client_send` was called, at the time of entry one well-defined processor register contains the number of words to transfer and another register contains the address of the data buffer. As the data are transferred, the pointer register is incremented and the word count register decremented to reflect the new start of the data to transfer and the new amount of data to send. For example, if an IPC tries to send 8,192 bytes starting from address `0x08001800` and successfully transfers the first 6,144 bytes and then causes a page fault, the registers will be updated to reflect a 2,048 byte transfer starting at address `0x08003000`. Thus, the system call can be cleanly restarted without redoing any transfers. The IPC connection state itself is stored as part of the current thread's control block in the kernel so it is not passed as an explicit parameter, though that state too is cleanly exportable through a different mechanism. Interfaces of this type are relatively common in Fluke, and the majority of the IPC interfaces exploit both parameter and program counter manipulation.

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. As a simple example, consider the requirement that updatable system call parameters be passed in registers. If instead parameters were passed on the user-mode stack, modifying one might cause a page fault—an indefinite wait—potentially exposing an inconsistent intermediate state.

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 rarely called directly from user-mode programs, and are instead 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.

Actual Cause of Exception	Cost to Remedy	Cost to Rollback
Client-side soft page fault	18.9	none
Client-side hard page fault	118	2.2
Server-side soft page fault	29.3	2.5
Server-side hard page fault	135	6.8

Table 3: Breakdown of restart costs in microseconds for possible 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`). "Actual Cause" describes the reason the exception was raised: either a "soft" page fault (one for which the kernel can derive a page table entry based on an entry higher in the memory mapping hierarchy) or a "hard" page fault (requiring an RPC to a user-level memory manager) in either the client or server side of the IPC. "Cost to Rollback" is roughly the amount of work thrown away and redone, while "Cost to Remedy" approximates the amount of work needed to service the fault. Results were obtained on a 200-Mhz Pentium Pro with the Fluke kernel configured using a process model without kernel thread preemption.

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 32-bit "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.

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 3, 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 is not the common case. 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 processor architectures make it impossible for the kernel to provide cor-

rect and prompt state exportability, because the processor itself does not do so. For example, the Motorola 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 68000-class 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*. While there is typically a strong correlation between a kernel's API and its internal execution model, in many ways these issues are independent. 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.

Performance: Depending on the application running on the kernel, either a process-model or interrupt-model kernel can be faster, but not by much. In terms of preemption latency, an interrupt-model kernel can perform as well as an equivalently configured process-model kernel, but a fully-preemptible process-model kernel provides the lowest latency.

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

Architectural bias: Some CISC architectures that insist on providing automatic stack handling, such as the x86, are fundamentally biased towards the process model, whereas most RISC architectures support both models equally well.

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.

```
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: An example IPC send and 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.

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

5.1 Exported API

In kernels with medium-to-high level API's, one of the most common objections to the interrupt-based execution model is that it requires the kernel to maintain explicit continuations. Our observation is that continuations are not a fundamental property of an interrupt-model kernel, but instead are the symptom of the mismatch between the kernel's API and its implementation. In brief, an interrupt-model kernel only requires continuations when implementing a conventional API; when an interrupt-model kernel serves an atomic API, *explicit* user-visible register state of a thread acts as the "continuation."

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 should not 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_rcv` 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, 11]. 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

```
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 and receive path. State defining the "middle" of the send-receive is saved away by the kernel after `msg_send` in the case that the `msg_rcv` is interrupted. Special code, `msg_rcv_continue`, is needed to handle restart from a continuation.

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,

[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 SUCCESS;
}
```

Figure 4: Example IPC send and 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 is already in the thread's user-mode register state.

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.

Interrupt-Model Kernels Without Continuations

Finally, contrast these 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, system call parameters are passed in registers rather than on the user stack. The system call entry and exit code saves the appropriate registers into the thread's control block in a standard format, where the kernel can read and update the parameters. Second, since the system call parameters are stored in the register save area of the thread's control block, no unique, per-system call continuation state is needed. Third, 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. Return values in the kernel are only used for *kernel-internal* exception processing; 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, the kernel updates the user-mode program counter to point to the user-mode system call entrypoint for `msg_rcv` before proceeding with the `msg_rcv` stage. Thus, if the `msg_rcv` must wait or encounters a page fault, it can simply return an appropriate (kernel-internal) result code. 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 a thread's user-mode register 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.

5.2 Preemptibility

Although the use of an atomic API greatly reduces the kernel complexity and the 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 [23] and VSTa [33] 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 non-preemptible. 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 can be configured 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 carefully controlled. 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 support partial preemptibility in a kernel by introducing *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 both execution models. QNX [19] is an example of another existing interrupt model kernel whose IPC path is made preemptible in this fashion.

Configuration	Description
Process NP	Process model with no kernel preemption. Requires no kernel-internal locking. Comparable to a uniprocessor Unix system.
Process PP	Process model with "partial" kernel preemption. A single explicit preemption point is added on the IPC data copy path, checked after every 8k of data is transferred. Requires no kernel locking.
Process FP	Process model with full kernel preemption. Requires blocking mutex locks for kernel locking.
Interrupt NP	Interrupt model with no kernel preemption. Requires no kernel locking.
Interrupt PP	Interrupt model with partial preemption. Uses the same IPC preemption point as in Process PP. Requires no kernel locking.

Table 4: Labels and characteristics for the different Fluke kernel configurations used in test results.

5.3 Performance

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. Since full kernel preemptibility requires the ability to block within the kernel and is therefore incompatible with the interrupt model, there are five possible configurations, summarized in Table 4.

Table 5 shows the relative performance of three 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. For calibration, the raw execution time is given for the base configuration. 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 are:

Flukeperf: 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: Compile a single .c file. This test include running the front end, the C preprocessor, C compiler, assembler

Configuration	memtest	flukeperf	gcc
Process NP	1.00 (2884*ms*)	1.00 (7120*ms*)	1.00 (7150*ms*)
Process PP	1.00	1.01	1.03
Process FP	1.11	1.20	1.05
Interrupt NP	1.00	0.94	1.03
Interrupt PP	1.00	0.94	1.03

Table 5: Performance of three applications on various configurations of Fluke kernel. Execution time is normalized to the performance of the process-model kernel without kernel preemption (Process NP) for which absolute times are also given.

and linker to produce a runnable Fluke binary.

As expected, performance of a fully-preemptible kernel is somewhat worse than the other configurations due to the need for kernel locking. The extent of the degradation varies from 20% for the kernel-intensive *flukeperf* test to only 5% for the more user-mode oriented *gcc*. Otherwise, the interrupt and process model kernels are nearly identical in performance except for the *flukeperf* case. In *flukeperf* we are seeing a positive effect of an interrupt model kernel implementation. Since a thread will restart an operation after blocking rather than resuming from where it slept in the kernel, there is no need to save the thread's kernel-mode register state on a context switch. In Fluke this translates to eliminating six 32-bit memory reads and writes on every context switch.

Because the applications shown are all single-threaded, the results in Table 5 do not realistically reflect the impact of preemption. To measure the effect of the execution model on preemption latency, we introduce a second, high-priority kernel thread which is scheduled every millisecond, and record its observed preemption latencies during a run of *flukeperf*. The *flukeperf* application is used because it performs a number of large, long running IPC operations ideal for inducing preemption latencies.

Table 6 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 high-priority kernel 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 result is not surprising given that the benchmark performs a number of large IPC operations, but it illustrates that a few well-placed preemption points can greatly reduce preemption latency in an otherwise nonpreemptible kernel.

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

Table 6: 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 thread runs during the lifetime of the application and the number of times it failed to complete before the next scheduling interval.

5.4 Memory Use

One of the 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 memory overhead was reduced by 85% when the kernel was changed to use a partial interrupt model [10, 11]. 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 7 memory usage measurements gathered from a number of different systems and configurations. The Mach figures are as reported in [10]: 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 [24]. 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. (Fluke's device drivers were "borrowed" [14] from legacy systems and require a 4K stack.)

To summarize these results, although it is true that interrupt-model kernels most effectively minimize kernel thread memory use, at least for modest numbers of active threads, much of this reduction can also be achieved in process-model kernels simply by structuring the ker-

System	Execution Model	TCB Size	Stack Size	Total Size
FreeBSD	Process	2132	6700	8832
Linux	Process	2395	4096	6491
Mach	Process	452	4022	4474
Mach	Interrupt	690	—	690
L3	Process	1024		1024
Fluke	Process	4096		4096
Fluke	Process	1024		1024
Fluke	Interrupt	300	—	300

Table 7: Memory overhead in bytes due to thread/process management in various existing systems and execution models.

nel 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 them to be a multiple of the page size in order to use a "red zone." These results should apply to other architectures, although 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, where the processor automatically saves the thread's state on kernel entry before kernel code manually switches to the "real" kernel stack. 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 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 exokernels and the CacheKernel. For example, especially on the x86 architecture, it is impractical for a 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 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 [30] 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 [21]; 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 [11] 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 process-model user stacks separate from the kernel's interrupt-model 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 design allows the core kernel to treat these process-level activities just like all other user-level activities running in separate address spaces, except that this particular address space is set up a bit 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 synchronize and 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 or synchronization primitives as the core kernel uses. 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; however, our experience indicates that this cost is negligible. Finally, note that the memory management hardware on some processors, particularly the MIPS architecture, does not support this technique; however, at least on MIPS there is no compelling need for it either because the processor does not have the traditional CISC-architecture bias toward a particular kernel stack management scheme.

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 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 exports 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 a controlled 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. However, the interrupt model can incur vastly higher preemp-

tion latencies unless care is taken to insert explicit preemption points on critical kernel paths. 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 interface issues and their implications for reliability, to Kevin Van Maren for elucidating and writing up notes on other aspects of the kernel's execution model, to the anonymous reviewers, John Carter, and Dave Andersen for their extensive and helpful comments, to Linus Kamb for help with the use of certain system calls, and to Eric Eide for last minute expert formatting help.

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] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[3] J. F. Bartlett. A Non Stop Kernel. In *Proc. of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, Dec. 1981.

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

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

[6] 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, Apr. 1992.

[7] 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, Oct. 1991.

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

[9] 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.

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

[11] 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.

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

[13] 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, Dec. 1995.

[14] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[15] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and HTML available under http://www.cs.utah.edu/-projects/flux/fluke/, 1996.

[16] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151. USENIX Assoc., Oct. 1996.

[17] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, Feb. 1988.

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

[19] 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.

[20] 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.

[21] Intel Corp. *Pentium Processor User's Manual*, volume 3. Intel, 1993.

[22] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. B. no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 52–65, St. Malo, France, Oct. 1997.

[23] 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.

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

[25] 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.

[26] 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.

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

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

[29] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Assoc.

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

[31] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. 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.

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

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