DRAFT: work in progress--- comments solicited

Evolving Mach 3.0 to Use Migrating Threads

Bryan Ford *Jay Lepreau*

UUCS-93-022

Center for Software Science
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
E-mail: {baford,lepreau}@cs.utah.edu

August 27, 1993

Abstract

Like most operating systems, Mach 3.0 views threads as statically associated with a single task. An alternative model is that of migrating threads, in which a single thread abstraction moves between tasks with the logical flow of control, and "server" code is passively executed. We have compatibly replaced Mach's static threads with migrating threads, isolating that aspect of operating system design and implementation. The key element of our design is a decoupling of the thread abstraction into the *controllable execution context* and the *schedulable thread of control*, consisting of a chain of contexts. A key element of our implementation is that threads are now "based" in the kernel, and temporarily make excursions into tasks via upcalls. The new system provides cleaner and more powerful semantics for thread manipulation, allows scheduling and accounting attributes to follow threads, simplifies both kernel and server code, and improves RPC performance. We have retained the old thread and IPC interfaces for backwards compatibility, with no changes required to existing client programs and only a minimal change to servers, as demonstrated by a functional Unix single server and clients. Code size along the critical RPC path has been reduced by a factor of three, while its logical complexity has been reduced by an order of magnitude. Initial timings show that the performance of local RPC, doing normal marshaling, has also improved by a factor of three. We conclude that a migrating-thread model is superior to a static model, and that it is feasible to improve existing operating systems in this manner.

1 Introduction and Overview

We begin by defining and explaining four concepts that are key to this paper. They are kernel and user *threads*, *remote procedure call* (RPC), *static threads*, and *migrating threads.* We explain how kernel threads interact in implementing RPC, and the difference between implementing RPC with static or migrating threads.

This research was sponsored in part by the Hewlett-Packard Research Grants Program.

Threads As the term is used in most operating systems and thread packages, **conceptually** a *thread* is a *logical thread of control* (thus the name), i.e., the flow of control resulting from a particular piece of work to be accomplished. In traditional Unix, a single process contains only a single kernel-provided thread. Mach and many other modern operating systems support multiple threads per process (per *task* in Mach terminology), called *kernel threads*. They are distinguished from *user threads*, provided by user-level thread packages, which implement multiple threads of control atop a single kernel-provided thread, by manipulation of the program counter and stack from user-space. In the rest of this paper, we use the term "thread" to refer to a kernel thread, unless qualified.

In most operating systems, a thread includes much more than the flow of control. For example, in Mach 3.0 a thread also (i) is the *schedulable entity*, with priority and scheduling policy attributes; (ii) contains *resource accounting statistics* such as accumulated CPU time; (iii) contains the *execution context* of a computation—the state of the registers, program counter, stack pointer, and handles on the containing task and designated exception handler; (iv) contains points of *thread control*—the *thread control port*[1] and the *suspend count*.

RPC *Remote procedure call*, as the name suggests, models the procedure call abstraction, but implemented between different tasks. The flow of control is temporarily moved to another location (the "procedure" being called) and later returned to the original point and continued. RPC can be used between *remote* computing nodes, but is often used between tasks on the same node: *local RPC*. This paper focuses on the local case, and we use the unqualified term "RPC" to refer to local RPC. To support RPC, a thread in a *server* task typically issues a blocking inter-process communication (IPC) kernel call, which in Mach is a message receive operation. When a thread in a *client* task needs an externally-provided service, such as opening a file, it issues an RPC to the file server by making some sort of IPC kernel call, usually a message send. In *synchronous RPC* the client thread stops executing its code, waiting for a response message from the server. Meanwhile the server thread receives the client's message (containing function name and *marshaled* arguments,), performs the actual work function, and issues another kernel call to send back the response message. (Typically, this is bundled with a message receive as well, waiting for the next request.) *Asynchronous* RPC is the term for an RPC call in which the client thread does not block, but continues executing *client* code. As in most systems, this is rare in Mach, and we use the unqualified term "RPC" to refer only to synchronous RPC.

Static Threads In the RPC mechanism outlined above, each thread was tied to a given task. When the client thread issued the RPC, control passed from the requesting thread in the client to a completely unrelated service thread in the server. We refer to this as the *static thread* model, because threads are statically associated with a given task. (In the object-based world, this is known as an "active object" model[10], because a server "object" contains threads that actively provide service.) The server thread's *resources* are being used to provide service to the client. In switching control from one thread to another, a full *context switch* was involved—a change of address mappings, task, thread, stack, registers, priorities, etc., and often the invocation of the kernel scheduler. The client thread's *state* must be saved, and typically the service thread contains state such as its registers and stack, which must be restored when it is activated.

Migrating Threads An alternate model, that of *migrating threads*, allows threads to move from one task to another as part of their normal functioning. In this model, during an RPC the kernel does not block the client thread upon its IPC kernel call, but arranges for it to continue executing, right into the server's code. A corresponding service thread that is waiting in a message receive is never awakened by the kernel—instead, for the purposes of RPC, the server is merely a passive repository for code. It is for this reason that in the object-based world, this is called the "passive object" model. A partial context switch is involved—the kernel switches address mapping and tasks,

---

[1] In Mach, a *port* is a kernel entity that is a capability, a communication channel, and a name—if one has the name of a port, on can perform operations on the object it represents.

and usually has to switch stacks due to separate protection domains, but does not switch threads or priorities, and does not invoke the scheduler. The client's computational resources (rights to use the CPU: priority, remaining time quantum, etc.) are being used to provide services to itself.

Although most operating systems support RPC using the static thread model, it is important to note that this is not the case for all system services. All OSs using the "process model"[16] for execution of their own kernel code[2] (e.g., Unix, Mach, Chorus, Amoeba), actually migrate the user's thread into kernel code during a kernel call. No context switch takes place—only the stack and privilege level are changed—and the user thread's resources are used to provide services to itself.

Static vs. Migrating Threads In actuality, there is a continuum between these two models. For example, in some OS implementations, certain client thread attributes, such as priority, can be passed along to ("inherited by") the server's thread. Or a service thread may retain no state between client invocations, only providing resources for execution[21]. Thus it can become impossible to precisely classify every thread and RPC implementation. However, most systems clearly lie towards one end of the spectrum or the other.

1.1 Providing Migrating Threads on Mach

Mach uses a static thread model, and a thread contains all of the attributes outlined in the "Threads" paragraph above. In our work, we decoupled these semantic aspects of the thread abstraction into two groups, and added a new abstraction, the *activation stack*, which records the client-server relationships resulting from RPCs. A *thread* is now only: (i) the logical flow of control, including the activation stack between *activations* in tasks, and (ii) the schedulable entity, with priority and resource accounting attributes. An *activation* represents: (i) the execution context of a computation, including the task whose code it is executing, its exception handler, and its program counter, registers, and stack pointer, and (ii) the points of control: activation (formerly "thread") control port and suspend count.

The abstraction exported to the user that corresponds to the old "thread" abstraction is what we internally call the "activation." This is not only what makes sense for the needs of user programs, but provides compatibility with original Mach 3.0, in which a thread was bound to only one task.

However, the real thread as defined above, the schedulable entity, is now a first-class entity, no longer subordinate to a task, as in the original static thread model. In addition, by explicitly recording the relationship between individual activations in the activation stack, we have elevated RPC to an entity fully visible to the kernel, instead of a sequence of message passing operations. Our thread abstraction now more closely models the original *conceptual* basis of a thread: a logical flow of control. It turns out that elevating the thread and RPC abstractions greatly enhances *controllability*, because the kernel can now take more elaborate and precise actions on a single activation or on the entire thread. Another benefit of introducing to the kernel the notion of an inter-task RPC, is that a number of aggressive IPC optimizations become possible. This was one of our original motivations, but many other benefits have since surfaced.

1.2 Outline of Goals and Benefits

Our original goals in this project were several: (i) change Mach 3.0's thread model to a migrating one, (ii) retain backwards compatibility, and (iii) enable performance improvements via RPC optimizations not possible with static threads.

During the design and implementation, we discovered we could achieve much more: (iv) normal marshaling RPC became much faster, (v) thread controllability was much enhanced, (vi) kernel code became much simpler, and (vii) several other advantages, discussed below, became evident.

In addition, to our knowledge, an existing operating system has never had two implementations that differed only in thread model. Now that we have done so, it is possible to make a concrete

The "process model" is in contrast to the "interrupt model," as exemplified by the V operating system[9], in which kernel code must explicitly save state before potentially blocking.

comparison of the two thread models, since we have isolated that aspect of operating system design and implementation.

In the rest of this paper we describe this work in detail. We first discuss related work, then cover the advantages of a migrating thread model, describe our kernel implementation and interface, including extensive discussion of the thread controllability issues, examine how RPC works in the new system, and explore how the Unix server could be changed to leverage migrating threads. Finally, we present the implementation status and preliminary results, outline future work, and the conclusions we draw from this work.

2 Related Work

Most operating systems use a static thread model, but there are a number of exceptions. Sun's Spring[19] operating system supports a migrating thread model very similar to ours, although it uses different terminology. Spring's "shuttle" corresponds to our "thread," and their "thread" corresponds to our "activation." Alpha[13] is oriented to real-time constraints, and its migrating thread abstraction is especially important for carrying along scheduling, exception-handling, and resource attributes. In both of these systems a thread can migrate across nodes in a distributed environment, and indeed Alpha's terminology for a migrating thread is a "distributed thread." Psyche[25] is a single-address-space system that supports thread migration...and lots of other stuff...expand comparison in final paper. The Lightweight RPC system[4] on Taos exploited migrating threads as a critical part of its design, but focused on high-performance local RPC. Object-oriented systems have traditionally distinguished between "active" and "passive" objects, corresponding to static and migrating thread models[10]. Clouds[14] exemplifies a passive object (migrating thread) model, while Emerald[5], as we do, provides both both active and passive objects—support for both styles of execution.

However, all of these systems were designed from the start with a migrating thread model, and are different from traditional operating systems in many ways other than thread model. To our knowledge, heretofore the thread model issue itself has not been separated out and examined. Our goal is to do this by comparing the two thread models in the *same* operating system, providing information focused on the thread model. By implementing migrating threads on Mach 3.0, we also demonstrate how an existing operating system with static threads can be adapted to migrating threads.

3 Motivation

A migrating thread approach has several advantages which are outlined in this section. The majority of the benefits are linked to use with RPC and are described first. But there are also controllability advantages for threads during *all* kernel interaction, and these are outlined in section 3.2. In the context of the Alpha OS, [13] also discusses many advantages offered by migrating threads.

3.1 Remote Procedure Call

Many of the advantages of migrating threads stem from their use in conjunction with RPC. Migrating threads provide a more appropriate underlying abstraction on which to build RPC interfaces than do static threads. Many of the problems with static threads stem from the semantic gap between the control model—a procedure call abstraction within a single thread of control—and the mechanism used to implement the model—two threads executing in different tasks. Using migrating threads for RPC provides benefits in performance, functionality and in ease of implementation. Since RPC is very frequently used, especially in newer microkernel-based operating systems where most internal system interactions are based on RPC, this aspect of the system can be of great importance in determining the performance and functionality of the system as a whole.

3.1.1 Invocation Efficiency

For RPCs to be performed in the static thread model, two threads, one in each task, must synchronize and exchange information in the kernel. Two thread-to-thread context switches are required during the operation: one on call and one on return. However, in the migrating thread model, the entire RPC can be performed by just one thread that temporarily moves into the server task, performs the requested operation, and then returns to the client task with the results. No synchronization, rescheduling, or full context switching needs to be done.

Thread migration also permits well-known optimizations such as LRPC[4], as well as numerous other optimizations in flexibly structured or shared address space systems e.g., Lipto[17], Opal[8][3], FLEX[7], and Mach In-Kernel Servers[22, 18]. In these systems some degree of inter-domain memory sharing is accomplished, thus blurring domain boundaries. RPC implemented by threads that migrate from one domain to another can take advantage of this boundary blurring, providing many optimizations in argument passing, stack handling, etc.

In general, migrating threads provide *invocation* support that is more widely applicable than that offered by static threads. This advantage is relevant to more styles of invocation than simply RPC. In particular, the benefits in an object-based environment are great, because invocation of relatively fine-grained objects is prohibitively inefficient if the objects must be active. With passive objects, it is more feasible to apply similar OS abstractions to both medium and course-grained objects.

3.1.2 Thread Attributes and Real-time Service

In the static thread model, when a client task performs an RPC, control is transferred to an entirely different thread that has its own scheduling parameters such as execution priority, as well as other attributes such as resource limits. Unless specific actions are taken, the attributes of the thread in the server will be completely unrelated to those of the client thread. This can cause the classic problems of *starvation* and *priority inversion*[15], when a high-priority client is unfairly made to compete with low-priority clients that are accessing the same server. On the other hand, if the client thread migrates into the server to perform the operation, all such attributes can be properly maintained with no extra effort. Obviously, this issue is of particular importance to systems that attempt to provide guarantees of real-time service.

A related advantage is in resource accounting, which can be made more accurate since the work done in a server on behalf of a client can automatically be so attributed.

3.1.3 Interruptions

Often, due to asynchronous conditions, it is desired to interrupt an RPC in which a client is blocked, either temporarily or permanently. To do this cleanly in the static thread model, it is not enough merely to abort the message send/receive operation, because the server will continue processing the request without any indication that the client no longer desires its completion. If some entity wants to abort an RPC in which a thread is blocked, it must find the server to which the RPC is directed, know how to interact with that server enough to send it a request to abort an RPC operation, and provide the server with some kind of identification specifying which RPC is to be aborted. This usually proves to be a complex and difficult process. In addition, every server that may be accessed must support these abort operations. This can be difficult to guarantee in practice, especially if any user-mode task can set itself up as a "server" and allow other user threads to make RPCs to it, as Mach 3.0 allows. Migrating threads, on the other hand, provide a channel through which standardized requests for interruption can be propagated.

3.1.4 Server Simplification

Migrating threads simplify the implementation of RPC in servers. RPC servers based on static threads must create and perpetually maintain a "pool" of service threads whose sole purpose is to

Opal claims that threads remain within a protection domain, but closer examination seems to indicate that the thread actually migrates in the intra-node case.

wait for and service incoming RPC requests. Management of this thread pool to achieve maximum performance without excessively large resource consumption is tricky, especially on multiprocessors, where the number of service threads waiting for RPC requests must at all times be carefully balanced to match the number of processors. If instead, server code is simply executed by clients' migrating threads, the balancing occurs automatically.

In the case of "personality servers" that emulate monolithic operating systems such as Unix or OS/2, migrating threads can simplify the server even more, because the original operating system on which the server is based is likely to have used a limited migrating thread model, in which threads "migrate" into the monolithic kernel for system calls. Maintaining this model in the personality server achieves greater code re-use and simplifies the handling of system call interruptions, thread management, and control mechanisms such as Unix signals.

3.1.5 Kernel Simplification

As later shown by our results in Section 8.3, migrating threads greatly simplify the kernel as well. Kernel RPC paths based on migrating threads tend to be short and flow naturally, while optimized RPC paths based on static threads are often long, convoluted, and contain innumerable tests.

While the existence of fast, efficient microkernels based on static threads demonstrates that high performance is possible in this model, such systems usually impose semantic restrictions that distort their implementation towards a migrating thread model. For example, QNX[20], a commercial real-time operating system, supports only unqueued, synchronous, direct process-to-process message passing with priority inheritance. This design makes it a *de facto* migrating threads system even though it does not claim to be one.

3.2 Thread Controllability

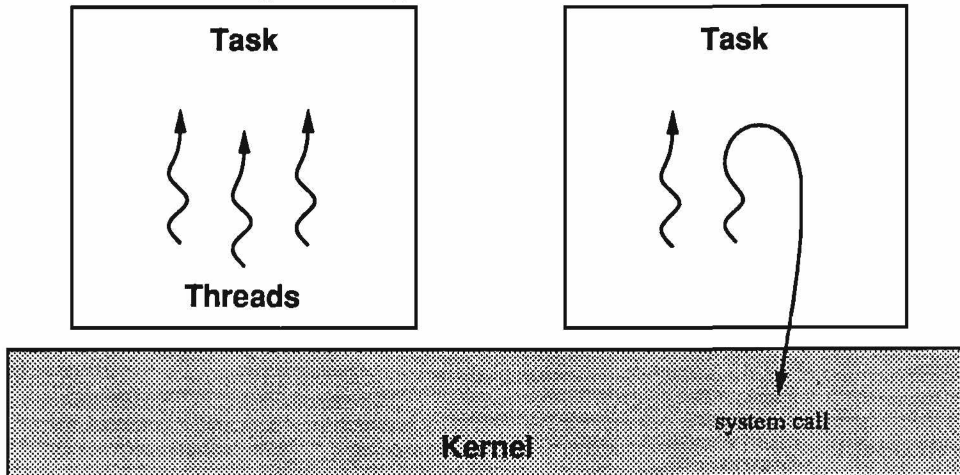
In a static thread model, threads are often intended to be *completely controllable resources.* Ideally, in this model, any entity with appropriate privilege, such as a program holding a "thread control port" in Mach 3.0, is able arbitrarily to stop a thread and modify its state, at any time. Conceptually, threads execute only user-mode instructions, and therefore there is never a time when system integrity could be violated by manipulation of the thread.

Unfortunately, this model in its purest form does not work in real operating systems. Threads must be able to invoke kernel-level code in order to communicate with other entities in the system, if they are to do anything more than pure computation. Since a thread executing unknown kernel code may *not* be arbitrarily manipulated, the model of complete controllability must break down somewhat: it must be possible to defer or reject thread control operations when necessary.

Traditional operating systems have various ways of working around this problem which usually work, but are often complex, inconsistent, and unreliable. For example, Mach 3.0 provides a thread control operation which aborts a system call in which the target thread is blocked, so that the thread can be manipulated. However, many kernel operations cannot be aborted in a transparent, restartable way, so the entity trying to control the thread may have to wait an arbitrary length of time, or retry an arbitrary number of times, before it can safely do so. If this is the case, who is really being controlled— the target thread, or the thread trying to control it?

The ability of threads to migrate merely adds additional situations in which threads cannot be arbitrarily controlled. Since the complete controllability model is not realistic anyway, reducing the ambitiousness of the model, to allow for migrating threads, does not necessarily entail any loss of functionality. In fact, by forcing the boundaries of controllability to be explicitly defined, and explicitly recording the flow of control across tasks, the thread control mechanisms provided by the kernel can be made *more* powerful.

Figure 1: Original Mach 3.0 Thread Model



4 Kernel Implementation

In this section we describe the underlying structure of our implementation of migrating threads in the Mach 3.0 microkernel. Many of the techniques we used could be similarly applied to other traditional multithreaded operating systems such as Unix macrokernels.

4.1 Thread Implementation

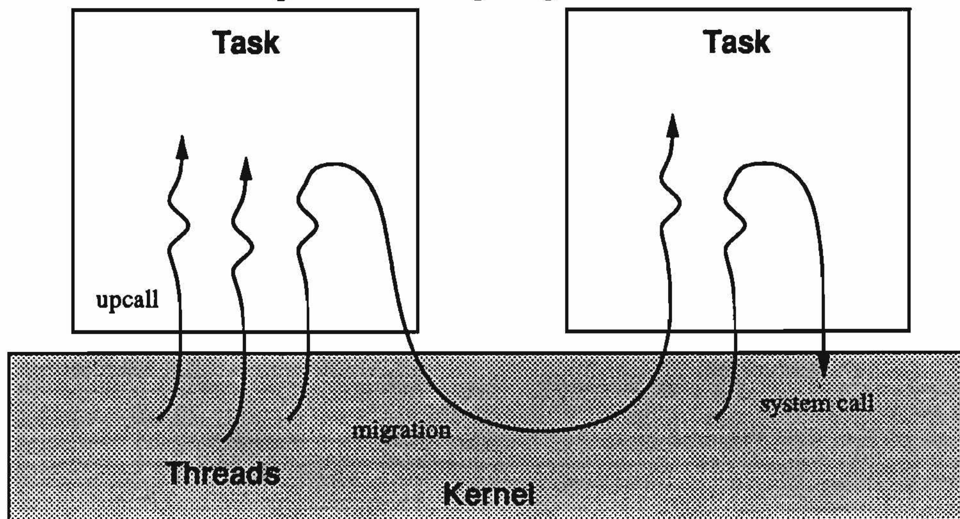
Conceptually, a traditional Mach 3.0 user thread started executing in a particular task, and occasionally trapped into the kernel to communicate with "outside" entities. The kernel later returned from the system call and resumed the user code. The initial and normal location of a thread was in user space, and threads only "visited" the kernel occasionally, to request services. Figure 1 illustrates this system.

In our migrating thread implementation, the situation is in a sense reversed, as illustrated in Figure 2. A thread starts executing as a purely kernel-mode entity, and later makes an upcall[12] into user space to run user code. Conceptually, the kernel is "home base" for all threads: the only time user-level code is executed is during "temporary excursions" into a task. A thread executing in user mode is associated with the task in which it is currently running, but a thread running in the kernel is not tightly associated with *any* user-level task.

While a thread in the kernel can now make upcalls into user space, the traditional kernel/user interface is still preserved. Once a thread is executing in user space, it can make calls *back* to the kernel in the form of traps and exceptions. Alternatively, the kernel can make further upcalls into the same or a different user task. This redefinition of the kernel/user interface is the primary mechanism supporting migrating threads in our implementation.

A distinction should be made between the "kernel" and what we refer to as "glue" code. The kernel is conceptually a protection domain much like a user-level task, in which threads can execute, wait, migrate in and out, and so on; its primary distinction is that it is specially privileged and provides basic system control services. Glue code is the low-level, highly system-dependent code that enacts the *transitions* between the kernel and user tasks. The distinction between the kernel and glue code is often overlooked because both types of code usually execute in supervisor mode and are often linked together in a single binary image. However, this does not necessarily have to be the case; for example, in QNX[20], the 7K "microkernel" consists of essentially nothing but glue code, while the "kernel proper" is placed in a specially privileged but otherwise ordinary process. It will become clear in later sections that even though the kernel and glue code may still be lumped together, in the presence of migrating threads the distinction between them becomes extremely

Figure 2: Mach Migrating Thread Model



important.

4.2 Thread Creation

In Mach 3.0, threads are created by specifying a task to the `thread_create` operation, which constructs a new thread in that task, with execution suspended and undefined CPU state, but otherwise ready to run. Mach returns a "control port" for the new thread, which the caller must use to set the thread's CPU state and resume the thread.

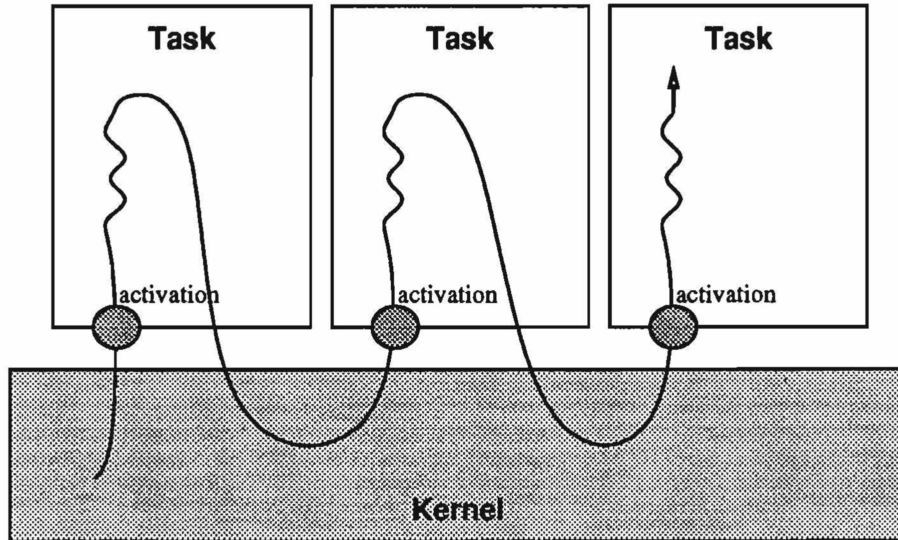
Since in our migrating threads implementation all entry into user-level tasks is done through upcalls, in order to maintain compatibility with the existing thread creation interface a way had to be created for a thread to make an "upcall" into a task, but before actually executing any user-mode instructions, take an immediate trap back into the kernel and suspend itself. We call this a *throughcall*— a call "through" the task and back out. It is relatively easy to implement in system-dependent code by setting up both an upcall frame and a trap frame on the kernel stack at the same time.

While the existing Mach thread creation interface must be supported for backward compatibility, a different interface would be more natural in a migrating thread model. Furthermore, the existing interface suffers a long-standing problem: while the Mach API was intended to be portable, CPU state information is by nature nonportable. Since thread creation requires setting CPU state, it is in turn nonportable. Migrating threads provide an opportunity for a much cleaner mechanism: the creation of a thread in a particular task could be modeled as a generic migrating RPC from the kernel (where the thread was created) into the specified task. The task could then "keep" the thread as long as it wants, and eventually destroy it by returning from the initial RPC. We intend to provide such an interface in the future.

4.3 Control Abstractions and Mechanism

Even in the static thread model, in practice the goal of complete controllability of threads cannot be fully realized. While the case of a thread being in the kernel can to some extent be worked around as a special case, with the addition of migrating threads the controllability issue must be more carefully considered. Now, not only is the kernel "out of bounds" for thread control, but in order to maintain protection between tasks, threads that have migrated to other protection domains may also be uncontrollable. For example, if one thread in a client migrates into a server for an RPC, it would not be permissible for another thread in the same client to arbitrarily stop or manipulate

Figure 3: Activations



the CPU state of the first thread while executing server code.

To provide controllability and protection at the same time, we split the concept of a "thread" into two parts: the part used by the scheduler, and the part allowing arbitrary control. The first, which we still refer to as the "thread," migrates between tasks and enters and leaves the kernel. The second, which is a user-mode activation that we label an *activation*, remains permanently fixed to a particular task. Arbitrary control is permitted *only on a specific activation*, not on the thread as a whole.

Whenever a thread migrates into a task (including the initial upcall from the kernel on thread creation), an activation is added to the top of the thread's "activation stack." When a thread returns from a migration, the corresponding activation is popped off the activation stack. This is illustrated in Figure 3.

Activations are created either implicitly during thread creation, or explicitly by servers expecting to receive incoming migrating threads. An explicitly created activation is *unoccupied* until a thread migrates into the task and "activates" it.

Control of activations is implemented primarily through *asynchronous procedure calls*, or APCs, similar to asynchronous traps (ASTs) in monolithic kernels. When returning from the kernel into an activation, glue code checks for APCs attached to the activation and if present, calls them. For example, to suspend an activation, an APC is attached to that activation which will block until resumed. Previously, Mach dealt with thread suspension as part of the scheduler, adding more complexity to its already-complex state machine; now the scheduler knows nothing about suspension.

4.4 Kernel Stack Management

In our current migrating threads implementation, each activation must have its own kernel stack. This is because some information about the linkage of one activation to the next is stored on the kernel stack by a migrating upcall into a task, and since the activation chain can be broken at any point, the information for each activation must be separate. We have begun to implement a design in which *all* linkage information is stored in the activations themselves, and a single kernel stack is sufficient for the entire thread. In fact, this is required for it to be possible to do task migration across nodes in a distributed system, because state held on a kernel stack cannot be easily encapsulated for migration.

5 Controllability: Semantics, Interface, and Implementation

In this section we describe the semantics of thread control operations, the interface to those operations, and some aspects of the implementation. We again believe our approach could be similarly applied to other traditional multithreaded operating systems. In fact, Mach 3.0 presented an especially difficult case for evolution to migrating threads because of its rich, powerful thread control interface; adaptation of other operating systems with less powerful interfaces should be more straightforward.

5.1 Control Interface

In the original Mach kernel, threads were exported to user-mode programs in the form of *thread control ports*, through which control operations could be invoked. In our system, while threads still exist, the control abstraction presented to user-level code is instead the *activation control port*. This can work because the old thread execution abstraction exported to the user was bound to a single task, very much like activations are now. Internally the kernel now maintains a new "chain of activations" abstraction associated with the schedulable entity— what this paper terms the thread. We maintain compatibility with existing Mach code by making activation control ports direct replacements for thread ports at the binary level— all system calls which previously expected or returned thread ports now use activation ports instead. For compatibility at the source level, appropriate synonyms are provided.

The question remains of how the thread control operations in the old system should map to activation operations. Mach 3.0 provides the following primary operations on thread ports:

- `thread_abort` and `thread_abort_safely`: Abort a kernel operation in progress and return control to user code.
- `thread_suspend` and `thread_resume`: Suspend and resume the thread's execution.
- `thread_terminate`: Destroy a thread.
- `thread_get_state` and `thread_set_state`: Set and query the thread's user-visible CPU state.
- `thread_get_special_port` and `thread_set_special_port`: Set and query the *special ports* attached to the thread, such as the exception port and the `mach_thread_self` port.
- `thread_priority`, `thread_max_priority`, `thread_policy`, and `thread_info`: Set and query the thread's priority and scheduling policy, and retrieve scheduling statistics.

Our methods for handling these operations are described in the following sections.

5.2 Abort Requests

In our migrating threads implementation, we have provided the functionality of Mach 3.0's `thread_abort` and `thread_abort_safely` calls, but in a cleaner and more general form. An *abort request* is a form of asynchronous message passed from a client to the kernel or a server it is calling, asking the callee to abort the requested operation and return control to the client as soon as possible. Abort requests are primarily an information-passing mechanism supported by the kernel in a uniform way. They do not in themselves provide control over threads, because they have no forcefulness: abort requests are merely "requests," not "demands."

An abort request may be "posted" to a specific activation in a number of situations, such as when `thread_abort` (now `act_abort`) is called. If the target activation is not the topmost in its thread, and the activation immediately above it (i.e., the server it is calling) has specified an interest in hearing about abort requests, the request is propagated upward into that activation as well. This continues until the top of the activation stack is reached or until an activation is encountered in which abort requests are blocked.

If the abort request reaches the topmost server, then as soon as that server notices the request, it returns to its caller with an appropriate status indicating that it has complied with the request. If its caller is not the activation on which the request was posted (i.e., the caller is somewhere in the middle), then it also attempts to comply with the request. Eventually the activation stack unwinds and control returns to the client in which the request was made. The kernel can honor abort requests as well as servers, and in fact the unwinding is likely often to begin in a blocked kernel operation.

There are two "flavors" of abort requests: *safe aborts* and *strong aborts.* Safe aborts are to be honored only if the operation in progress can be successfully restarted later. Strong aborts are to be honored whenever possible, regardless of whether the operations are restartable.

The existing `thread_abort` operation now posts a strong abort request, while `thread_abort_safely` posts a safe abort. Neither operation waits for the target thread actually to return to the specified activation; this is a slight deviation from original Mach semantics, but does not cause problems in normal use because of the way it interacts with other control mechanisms (see Section 5.2.3).

By default, new activations added to a thread's stack have abort requests blocked, to prevent interference with an unwary server's functioning. The kernel is already capable of honoring most abort requests, and new servers written to work with migrating threads can be designed to honor them too. In effect, we have provided a generic interruption request mechanism which works uniformly for both migrating RPCs and kernel calls.

In the future we intend to add a third abort operation, `act_abort_immediate`, which first generates a strong abort request at the target activation, then breaks the activation chain, returning control to the client immediately. This will work much like thread termination, discussed below.

5.2.1 Suspension

In Mach 3.0 thread semantics, the basic purpose of suspending a thread is to *prevent it from executing any more user-mode instructions* until it is resumed. Therefore, suspending a task's threads turns that task into a "passive entity," allowing its address space and other state to be examined or modified without interference from its threads. It is not required that all of the thread's computation be immediately stopped, as long as that computation does not occur in the task itself (i.e., it is acceptable for the thread to be in the kernel).

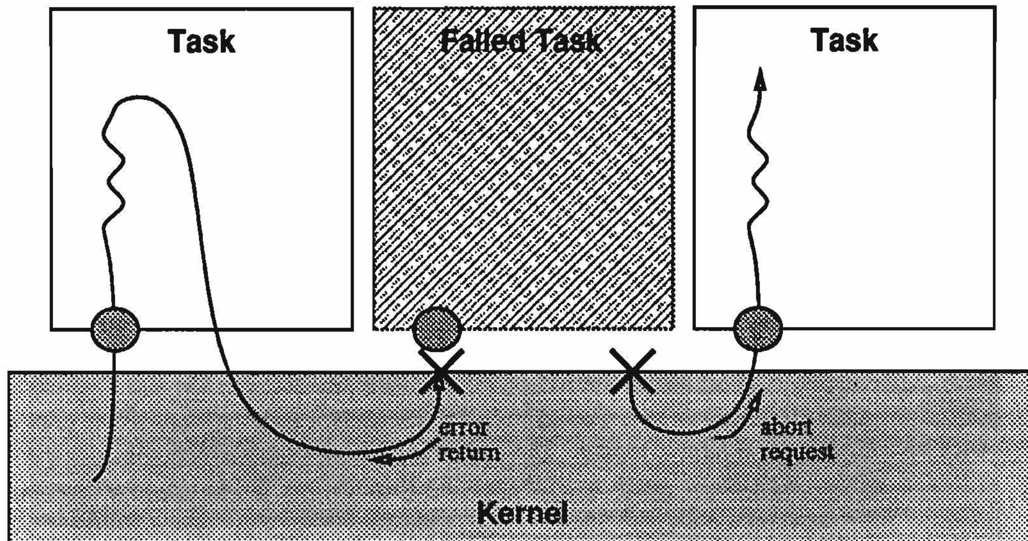
These semantics extend easily to migrating threads. When an activation is suspended, the kernel ensures that no more user-mode instructions will be executed *in that activation*. If the thread is executing elsewhere, it will not be affected until it attempts to return to the suspended activation. An unoccupied activation not currently attached to any thread can also be suspended and resumed; a migrating RPC which tries to enter it is blocked until the activation is resumed.

5.2.2 Termination

Mach 3.0's `thread_terminate` operation can be used both for orderly termination of a thread known to be at a "clean point," most often performed by a thread on itself, and for destroying a thread in a task whose internal integrity has been compromised, usually as an automatic side effect of terminating the task. Mapping thread termination to activation termination requires separate consideration of these two uses.

The former case, orderly termination, makes sense only when performed on a thread's *one and only* activation. Even in a static thread model, a thread in a correctly functioning task may not be terminated while processing an RPC request. Since the *only* purpose of a "guest" thread that has migrated into a server is to process an RPC request, it makes no sense to terminate an activation other than the first, or bottommost. Similarly, if an activation is not the topmost, then it is a client in the middle of an outgoing migrating RPC, which again cannot possibly be considered a "clean point." In fact, explicit activation termination for orderly shutdown is not necessary *at all* in the new model, except to maintain backward compatibility, because the initial activation could simply return to the kernel, "falling off the bottom" of the thread's activation stack and causing the thread to self-destruct.

Figure 4: Activation Termination



Therefore, the only termination case involving threads with multiple activations is the disorderly termination of the thread, in the event of a task failure. Put another way, the kernel may assume that if one of a stack of activations is terminated, it is because of a catastrophic failure in that activation's task, and its job is only to inform other affected parties without breaching their protection.

The termination mechanism in our implementation is illustrated in Figure 4. If a thread's topmost activation is terminated while that thread is executing in user mode, execution in that activation is immediately terminated and the thread returns from that activation with an error code indicating that the server has died. If the activation was the bottommost, as in orderly termination, then the thread destroys itself.

If an activation *not* at the top of a thread's activation stack is terminated, or if the thread is in a kernel call at the time, then the thread splits into two separate threads with identical scheduling parameters. One thread is left with the top part of the activation stack, above the terminated activation, and the other thread is given the bottom part. The thread with the upper segment continues executing in the topmost server uninterrupted, ensuring that thread termination does not violate protection. A strong abort request is automatically propagated upward through this thread, providing a hint that the work being done is probably no longer of value. The thread given the bottom part of the activation stack returns to its now-topmost activation with an appropriate error code, as described above.

This is essentially the same as the termination mechanism used in Spring[19]. Abort requests in this case act like Spring's "alerts."

In our system, not only can a thread be split when running in user mode in a more recent activation than the terminated activation, but also when the thread is executing in the kernel, on behalf of the terminated activation. This in fact provides *more* controllability than in the original Mach 3.0 system. Previously, attempting to terminate a thread could potentially block for some time, while the caller waited for the victim to leave the kernel or otherwise get to a "clean point." In the new model, terminating an activation is *always* an immediate operation: if the terminated activation happens to be calling the kernel, then the carpet is safely yanked out from under the kernel call, so to speak. A thread with no activations is left behind to finish whatever operation it was performing and quietly self-destruct afterwards.

This required careful planning of kernel data structures and locking mechanisms; in particular, the line between the kernel and glue code, described earlier, had to be defined precisely. Once worked

out, however, this technique not only added additional controllability, but considerably *simplified* the implementation of control mechanisms in the kernel, as we show in Section 8.3.

5.2.3 CPU State

The original Mach 3.0 design provided thread operations which, in the "ideal" complete controllability model, would allow a thread's entire CPU state to be saved, restored, examined, and modified at any time. All CPU state operations were provided by two primitives, **thread_get_state** and **thread_set_state**, defined to produce sensible results only while the target thread was suspended.

However, because of the problems with the complete controllability model, many of the things for which the CPU state control mechanisms are commonly thought to be useful, in fact cannot be reliably implemented in Mach 3.0. Appendix A examines some of these potential uses under both a static and a migrating thread model, and briefly describes what would be involved in making them work reliably in both models.

Since the existing CPU state control operations are already problematic, and it would be difficult to achieve complete backward compatibility with them, we chose to structure these operations in our migrating threads implementation to fit *current uses* of these operations: in particular, those made by the Unix server and emulator, and by application programs that create their own threads and control them in straightforward ways.

Mach 3.0 requires **thread_abort** or **thread_abort_safely** to be called on a thread just before examining or setting its state, unless the thread has just been created. Otherwise, the state operation could work with "stale" information, producing useless results. Under migrating threads, aborting an activation before manipulating its state is not strictly required. If not done, the CPU state operations wait patiently until the thread is in the target activation, without interfering with its functioning.

If a thread attempts to abort or manipulate the state of an activation *on the same thread*, instead of deadlocking the thread as might be expected, a special compatibility operation is invoked which allow the operation to complete immediately in a minimally functional way. This is a temporary hack motivated by the way the Unix server currently handles signals, and avoids significant change to the server. In Section 7.1.2 we describe a cleaner and more general way of handling signals under migrating threads, which we will implement later.

5.2.4 Scheduling Parameters

The final Mach 3.0 thread control operations that must be mapped to activation operations are those managing thread scheduling parameters such as priority, scheduling policy, and CPU usage statistics. Unlike the operations described above, these operations are still conceptually performed on threads rather than activations. However, the original Mach 3.0 thread control ports have become activation ports, raising the question of how the interface for these operations should be handled.

Since every active activation is attached to exactly one thread, in our current implementation we export thread operations as operations on activations, and, in the kernel, redirect the operations to the attached thread. However, this raises a protection problem, since *any* activation in the thread can modify the global scheduling state. For example, a server could lower a thread's maximum priority (which cannot be raised without special privileges) while processing an RPC, leaving the client with a "crippled" thread upon return. In our initial implementation, this is not a problem in practice because the Unix server is trusted by all clients. However, a better solution will be needed eventually.

One solution would be to provide an activation operation which forbids future activations higher in the stack from changing global thread state. Thread state could still be manipulated from that activation or lower (assuming it has not also been forbidden at a lower level).

Another, more general approach would be to partially reinstate threads as user-visible entities. Each activation would contain a send right to which all thread operations get redirected. This send

right would by default refer to the actual thread within the kernel, but it could be redirected to an arbitrary user-level port. Activations added to a thread's stack would inherit the thread port from the previous level. Thus, a client could intercept requests by servers to change global thread state, or merely forbid servers from touching it by setting the thread port to an invalid send right.

5.3 Task Control Interface

Most task control operations work the same way under migrating threads as in the original Mach design. Others were modified in straightforward ways to match the new thread model. In particular, the **task_threads** call now returns a list of the activation ports of the task, instead of thread ports. When a task is suspended, resumed, or terminated, all of the activations within it (instead of threads) are similarly suspended, resumed, or terminated.

6 Migrating RPC

Once the basic kernel mechanism for supporting migrating threads was in place, the task remained of demonstrating its effect on RPC performance and complexity. Because our focus at this point is purely on migrating threads, our initial implementation retains the original Mach kernel message interface, with fully marshaled data, etc. In this section we describe this RPC system, as well as the changes required to servers to make them support migrating RPC. (No changes are required to make them run with traditional RPC, since the kernel itself is almost completely backward compatible.)

6.1 Client-side

From the client's point of view, RPC semantics are unmodified. The kernel checks the message options to make sure they are compatible with migrating RPC, and checks the destination port to ensure that the server is capable of handling migrating RPCs. For a request to be compatible with migrating RPC, it must meet these requirements: (i) both a message send and a message receive must be requested at the same time; (ii) the reply port in the outgoing message must match the receive port for the receive operation; (iii) no timeout may be specified; and (iv) no notification requests may be specified. In practice, almost all MIG-generated **mach_msg** calls meet these requirements, so most clients automatically make use of migrating RPC. Note that the data can contain port rights and out-of-line memory.

6.2 Server-side

Initializing a server to support migrating RPCs is done in much the same way as in normal static-threads servers. The difference is that the server must create one or more unoccupied activations, each containing a pointer to a stack in its own address space, and the entry point of its normal dispatch function. Providing this information to the kernel will be encapsulated within a function, probably within the **cthreads** package.

Traditional static-thread RPC is still supported automatically. A large pool of server threads is no longer needed, but at least one must still exist to process occasional asynchronous messages, because in the current implementation this is used as a fallback mechanism when migrating RPC cannot be used.

When a migrating RPC is made into the server, the kernel allocates an unoccupied activation from the server's pool, copies the incoming message onto the server stack, and makes an upcall into the server task to the dispatch routine. This MIG-generated routine is identical to the one used to dispatch traditional messages, except that it returns through a special kernel entry point. On return, the kernel does not need to do any security checks or port manipulation, and the reply port provided by the client in the **mach_msg** call is not used at all.

If a migrating RPC is attempted and the kernel discovers that there are no activations currently available, in our initial implementation the kernel falls back to the normal message path, causing a

normal message to be queued to the port. This is not ideal, and better solutions are obvious. One method is to block incoming migrating RPCs until an activation is available. Another method would be to detect when the last available activation is about to be used for a migrating RPC, and instead of immediately making the requested RPC, temporarily "sidetrack" and make a special notification upcall into the server. At this point the server can create more activations if it deems this desirable. If it does, it returns them to the kernel and the original RPC can proceed. Otherwise, it returns immediately and the RPC blocks until a stack is freed.

6.3 Problems with Migrating RPC

Migrating RPC presents a number of potential problems, most of them infrequently occurring, which we discuss in this section.

6.3.1 Message Ordering

In Mach 3.0, messages sent from a particular thread to a particular port will be received in the same order they were sent. There is one case in which these semantics could be violated in our migrating threads implementation. If a thread sends an asynchronous message to a server the message can be queued on the server's port. If the client then immediately does a migrating synchronous RPC, it could migrate directly into the server, "ahead of" the queued message. We do not expect this to be a problem in practice: asynchronous and synchronous messages are not often mixed, and even if they are, it is likely that the server receiving them has multiple threads receiving messages from the same port, causing ordering to be lost anyway. If necessary, this problem could be avoided by disabling migrating RPCs to a port containing queued messages.

6.3.2 Single-threaded Servers

Another issue is servers that contain only a single service thread, and therefore lack internal synchronization control. Since client threads can migrate in at any time, something must be done to ensure that two threads do not execute simultaneously. Multiple migrating threads can be prevented by allocating only one activation, but that would not prevent a migrating thread and the sole service thread from executing concurrently. For small servers, the easiest way might be to use a `mutex` lock in the server, which is acquired on entry to every server work function and released on return. A more transparent method would be for the kernel, on a thread migration, to check the server port's queue of threads waiting for a message, and if the service thread is on it, remove it for the duration of the activation. If the thread queue is empty, the kernel would block the migrating thread. Since important single-threaded servers are few, we have not yet dealt with this problem.

6.3.3 Unbounded Priority Inversion

There is a potentially serious security problem stemming from the priority inheritance between protection domains that is implicit in migrating RPC. Consider a multithreaded client program which calls a local trusted server on a uniprocessor. One thread in the client could depress its priority to a low value and then issue an RPC, resulting in server code running at low priority, possibly holding a lock internal to the server. Soon after the RPC request is issued, another client thread could loop at its normal priority. This would prevent the first thread (currently inside the server) from executing. If that thread indeed happens to hold a server lock, this scenario could effectively stop the server.

This is the classic problem of unbounded priority inversion[15]. The reason that priority inheritance causes this problem is that we have only half-implemented it. Previously, the priority of the client thread had no effect on server thread priority—there was no priority inheritance, and therefore interactions of this sort among threads caused no security problems. Of course, there were problems maintaining any kind of prioritization of service. But by solving priority inversion in one area we made it a more serious problem in another area. Since we are now transferring priority along with the thread, we therefore need to support priority inheritance in the rest of the system. In particular,

as soon as a high priority thread is blocked by a low priority one, the latter must temporarily be given the priority of the blocked thread. The changes necessary to support this can be confined to the kernel and the **cthreads** library.

At the end of section 9 we outline how our partial decoupling, of the schedulable entity from the activation stack, will make it easier to fully support priority inheritance. This is work we have in the design stage. Meanwhile, the problem could be worked around in a variety of ways. For example, the Mach kernel or trusted server could simply promote to normal the priority of any thread while in the server, and restore it upon return. The kernel would also need to prohibit any lowering of priority directed at any activation but the first in a chain.

6.3.4 User-level Thread Issues

The most important issue with migrating RPC is that the user-level thread management and synchronization package most widely used on Mach, **cthreads**, has significant limitations in the presence of migrating RPC.

Server Thread Management **Cthreads** presents a significant problem to the server of a migrating RPC. Servers use **cthreads** to multiplex user threads on top of kernel threads, replacing kernel-mode context switches with much faster user-level context switches, whenever possible. However, one of the main assumptions made by the user-level threads package is that all of the kernel threads on which it is running its user-level threads are interchangeable—that one kernel thread can be used for an operation just as well as another. This assumption can be satisfied in a static thread model, although in the process it makes real-time monitoring and control of server threads difficult.

In a migrating thread model, however, kernel threads migrating in from clients are *not* interchangeable—they may have different priorities and other attributes. Even ignoring this, the return-to-kernel after an RPC has been processed must be done on the same kernel thread that the RPC came in on. In general, trying to multiplex threads in this manner loses one of the main advantages of our design: providing a kernel entity (the activation stack) which represents a particular piece of work in progress, i.e., an entire logical thread of control. Therefore, multiplexing a server's user-level threads on top of incoming kernel threads is not appropriate. In **cthreads**, multiplexing can easily be avoided by "wiring" the user-level thread.

However, some speed is lost in the elimination of user-level thread multiplexing, because synchronization operations in the server sometimes now require kernel-level context switches instead of user-level context switches. Measuring real applications, including on multiprocessors, will be necessary before we can be sure the gains from better RPC performance are not outweighed by this additional cost. We believe that the speed advantage of user-level context switching is not as significant in typical RPC servers as it is in compute-intensive programs, the traditional benchmarks for thread implementations. In well-designed servers providing "system" functions, we suspect that internal contention can be minimized so that the importance of RPC speed outweighs that of context switch speed. We point out that in many commercial microkernel-based systems, including QNX[20], Chorus[24], and KeyKOS[6], OS servers do not use user-level threads. Instead, these systems either provide multithreading purely with kernel threads, or their functions are are sufficiently decomposed so that each server can be based on a single kernel thread, requiring no internal synchronization. However, until we have performance results for the Unix server with migrating RPC, losing user-level threads when servicing RPCs remains a concern.

Note that it is only for "guest" threads migrating in from other tasks that user-level thread multiplexing is a problem; threads native to the server can still use some kind of user-level thread system, or even a specialized multiplexing mechanism such as scheduler activations.

A More Appropriate Synchronization System Since **cthreads** can no longer multiplex user-level threads on kernel threads in servers, it should be replaced with a synchronization library better optimized to provide synchronization over kernel threads. Also, kernel-visible synchronization will

be necessary to fully implement priority inheritance. We are planning a replacement for **cthreads** that provides synchronization primitives in a user-level library, but in cooperation with the kernel.

7 The Unix Server

To function on the new kernel using traditional RPC, no changes were necessary to the OSF/1 single server and emulator, or to the libraries they use. To support migrating RPC, a few changes will be required. Initially, we are choosing ways which have minimal impact on existing code—better, cleaner mechanisms can be provided in the longer term. The server will be modified to invoke the new setup function in the **cthreads** library and to wire incoming **cthreads**. The existing complex management of the server's thread pool, while basically no longer used, can be retained. We anticipate making no modifications to the emulator. Since we are providing backwards-compatible semantics for thread manipulation, we expect that no modifications will be needed to the existing complex code for handling Unix signals.

We do not anticipate a large performance improvement in the single server. However, our initial goal is not primarily to show performance improvement, but to demonstrate the gain in simplicity and cleanliness provided by migrating threads, and how migrating threads can be implemented in a backward-compatible way in an existing operating system.

7.1 Desirable Modifications

The Unix server could be made simpler with two modifications that take advantage of migrating threads.

7.1.1 Use Abort Requests

Currently, because Mach 3.0 provides no standard way of propagating abort requests into RPCs, the Unix server must manually handle all Unix system call interruptions such as those caused by pending signals. It could be considerably simplified by taking advantage of the propagating abort operations now provided by the kernel. This would also make interruption semantics naturally extend to other servers in the system, such as ones installed by Mach-specific application programs running under Unix.

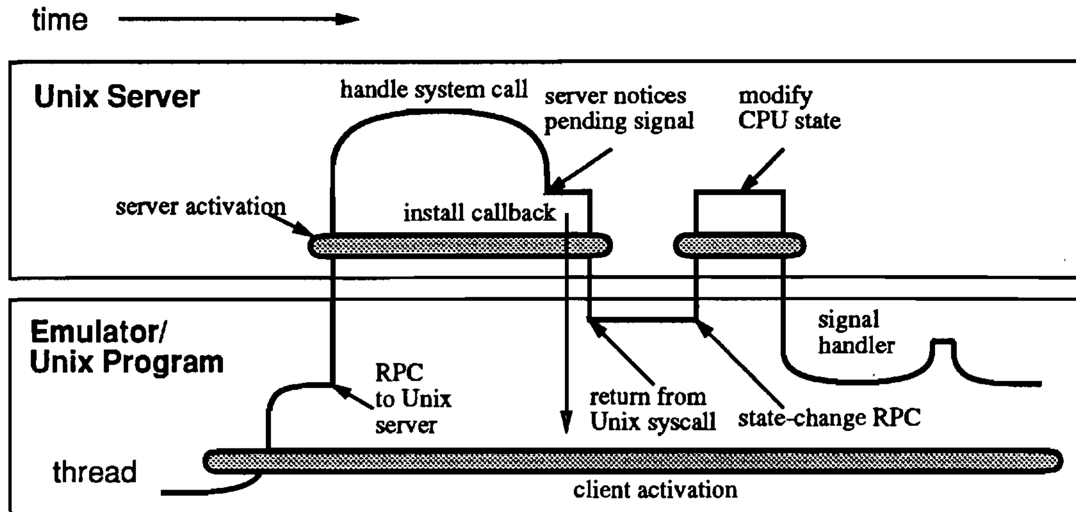
7.1.2 Take Advantage of Migration for Signal Handling

Emulating Unix signal semantics under Mach has always been an extremely complex and error-prone task, because delivering signals in the static thread model requires carefully synchronizing a client and server thread and "atomically" manipulating the client thread's stack frame from within the server. Migrating threads could considerably simplify signal management by taking advantage of the implicit protected synchronization provided by migrating RPC.

We plan to introduce into the kernel an alternate CPU state manipulation operation, with which a generic send right connected to a migrating RPC port can be "installed" into an activation. The next time the thread containing that activation returns to it (or immediately, if already in it), the kernel will make a migrating RPC to the installed port, similar to the exception RPCs already used by the kernel. The RPC will include the activation's complete CPU state, which can be modified arbitrarily. Any necessary modifications to other aspects of the client, such as pushing a signal stack frame on the client's stack, can be handled as they are now. When the RPC returns, execution will be resumed in the activation with the new state.

To deliver a signal with this system, instead of having to forcefully break a client thread out of its server RPC and carefully bring it under control, the Unix server simply can install a state-change callback port in the client's activation. As soon as the client thread has reached a clean, controllable place, it will make a migrating RPC into the server. The server can then modify the client's state and stack frame. When it returns from the RPC, the client will take the new state and begin executing

Figure 5: Cleaner Signal Handling with Migrating Threads



the signal handler. This system, illustrated in Figure 5, will effectively emulate the "asynchronous trap" (AST) mechanism typically used by macrokernels to handle signals.

Besides simplifying the server, this mechanism will obviate the need for the compatibility hack that we had to include in the kernel, described in Section 5.2.3.

8 Results

8.1 Status

At this writing basic kernel support of thread migration is complete, including the elimination of continuations[16], the separation of threads from tasks, support for upcalls into user space, and the new thread creation mechanism. The activation control mechanisms are fully implemented and backward compatibility with the old interfaces is nearly complete. The emulator-based Unix server runs unmodified on this new microkernel, but is still using traditional thread-switching RPC.

The reversal of the kernel/user interface model is fully implemented: threads in the kernel can call into user tasks, and user code can return to the kernel. Threads can migrate from one task to another. Tasks no longer maintain a list of threads, but do maintain a list of activations. Most of the task- and thread-manipulation routines have been modified to correctly redirect to the new activation-manipulation routines.

The required MIG and Unix server work has been started, but is not complete.

8.2 Performance

On an HP9000/730, which has a 67 Mhz PA-RISC 1.1 processor, preliminary timings of a null cross-task migrating RPC shows a cost of 640 cycles round-trip, compared to 2300 cycles with traditional Mach RPC. We expect this time to be further reduced by the time we release the final version of this paper.

We will include RPC performance comparisons for various argument sizes and types, including traditional RPC. We have a detailed breakdown of the time spent in various sections of the critical RPC path under static threads, which we will show along with the equivalent breakdown for the migrating RPC path. Finally, we will measure overall system performance running the OSF/1 single server. We will include counts of the different kinds of message paths, which should show that traditional RPC is no longer used for synchronous messages.

8.3 Code Simplification

8.3.1 Continuations

The first step in this project was disabling continuations[16] in order to make upcalls into user space possible. This required changing 217 lines and deleting 573. While not large, these changes were widespread, occurring in 54 places in many unrelated routines in the kernel. The changes substantially simplified control flow throughout: since continuations are effectively "non-local-gotos," the logical simplification was disproportionately larger than their number.

Note that a continuation model is not strictly incompatible with migrating threads. If the implementation did not use upcalls from kernel stacks, but instead stored all activation linkage information explicitly, a single kernel stack per thread could be used and a continuation model would be possible. We are currently implementing such a scheme, but will be using only a few continuations in the RPC path—not the many scattered uses that now exist.

8.3.2 Confining Controllability

Making threads independent of tasks and uncontrollable outside of user mode greatly reduced code complexity in a number of areas. The source file containing most thread management operations was reduced by more than half, from 72K to 32K. In the new 18K source file supporting activations, management operations account for only 7K. This largely resulted from cleaner management of thread suspension, resumption, and termination. As an example, `thread_terminate` previously had to make a special check to see if the thread being terminated was the current thread, and if so, handled termination separately. Since the kernel is now completely "out of bounds" for such control, an equivalent call to terminate an activation *on* the current thread only affects that activation, not the thread itself. In effect, a thread can now cleanly and safely shoot itself in the foot.

The task management module was reduced by 38%, due to the looser association between tasks and threads, which simplified locking and eliminated many special-case situations.

8.3.3 Migrating RPC

In the original version of Mach 3.0, the critical kernel RPC path executed a total of about 400 machine-dependent assembly language instructions, plus about 900 instructions of the (larger set of) code generated from 1350 lines of system-independent C code. In our as yet incomplete implementation of migrating RPC, the critical path now executes about 250 machine-dependent instructions plus about 140 instructions generated by 100 lines of system-independent code. This represents a reduction of 37% in system-dependent instructions and 84% in system-independent instructions. Overall instruction count reduction is a little more than a factor of three, which corresponds well with the speed improvement. We expect the system-independent code to be significantly further reduced by the time our implementation is complete.

Any programmer comparing the source code for the original RPC path with that of the migrating path will immediately notice the tremendous logical simplification of the code. To roughly quantify the reduction in complexity, we counted the number of branch instructions resulting from 'if' statements and other conditionals on the RPC paths. The old path contained 197 branches in the optimized `mach_msg_trap` path, while the new one contains 22, a reduction by a factor of nine. Note, moreover, that in the new code the common RPC path is the *only* path, which handles *all* migrating RPC. In the old code, only the "common cases" were handled by the path we measured; a much larger, mostly separate "fallback" code path was used for less common cases, meaning that much IPC code was duplicated. If we included both the optimized and unoptimized paths in our branch counts, our improvements in "logical complexity" would be far better than the cited factor of nine.

To understand the context-switch costs of the old and new systems, we must examine the Mach code. From the list of our thread and activation attributes in Section 1.1, it looks unlikely that we save much work on a context switch by not having to change the schedulable entity. But examination of the Mach implementation reveals why the old style thread handoff is so expensive. The kernel

essentially executes a portion of the scheduler, and there are numerous constraints that it must check: both the old and new threads must be in just the right states, they can't be locking resources, there is lots of run queue manipulation, lots of port right manipulation, lots of resources acquired along the way, and lots of care to be able to unroll everything if it were to fall off the optimized path. In essence, it's a very dynamic situation, and with that comes expense. By contrast, switching activations is much more of a static situation, and setting up the actual execution context is straightforward.

8.4 Memory Use

We will quantify the memory required for kernel stacks and activations, which will tend to be increased by the elimination of most continuations, but decreased by our new implementation that uses only one kernel stack for the entire migrating thread. We expect server memory requirements to remain about the same: a large thread pool is no longer needed, but an activation pool with attached server stacks is used instead.

9 Future Work

This work *enables* many further improvements to Mach and and Mach servers, as well as raising areas for further research. Providing an appropriate replacement for the `cthreads` synchronization primitives is important in order to make a fair evaluation of the impact of relying on kernel-level context switches. In Section 7.1 we outlined improvements that could now be made to the Unix server in signal and interruption handling. Other parts of the paper mentioned improvements that could be made to the Mach kernel, such as in the thread creation mechanism. Demonstrating that these claims of potential improvements are indeed true would be useful, and are areas we hope to pursue.

Our earlier work on moving trusted servers into the kernel's protection domain and address space (INKS)[22] used ad-hoc thread migration. By re-working the thread abstraction from scratch, our new system solves all of the problems encountered[18]. Partly as a result of removing continuations, it also allows critical optimizations that original INKS could not perform: server-kernel and server-server interactions. INKS could easily be adapted to our new system, and simplified in the process.

The "NORMA" (NO Remote Memory Access)[2] version of Mach 3.0 allows IPC between different nodes of a distributed memory multiprocessor, implemented in the microkernel. While the current migrating thread system does not support NORMA, extending it do so should not be difficult. Kernels would be able to forward threads across the network, snaking through activations on different machines, in much the same way that messages and ports are forwarded now. A system crash on a particular node can be handled as if every activation on that node failed due to internal errors, breaking distributed threads into multiple segments if necessary. These issues have already been explored in depth in Alpha[13].

The RPC optimizations made possible by migrating threads, especially in local RPC, raise the issue of asynchronous messages, which do not fit the RPC model. In Mach and the Unix server today, the only significant number of asynchronous messages are generated through the pager interface. It is possible to replace those with synchronous messages by redefining the interface. Whether that can be done while preserving a large degree of backward compatibility is currently unknown.

Going in a different direction, our work allows improvements in Mach's support for real-time systems. At the implementation level, we have largely decoupled two portions of the thread abstraction: the schedulable entity (priority, scheduling policies, etc.) from the thread of control (the chain of activations). This makes it feasible to decouple them entirely. While a thread will normally be attached to a particular schedulable entity, it will be possible for the schedulable entity to move briefly to a different thread. For example, if high-priority thread H must block waiting for a lock being held by low-priority thread L, then the logical action, and the necessary one in a real-time system, is for thread H temporarily to raise the priority of thread L to its own, until thread L releases the lock. This is normally done in real-time systems in ad-hoc ways, often made complex by the fact

that a thread's "priority" is often not just a single number: many different scheduling attributes may need to be transferred. Being able to detach temporarily a schedulable entity from its thread should make this much simpler.

An RPC Model More Amenable to Optimization The Mach message format imposes unnecessary overhead on migrating RPC; in the migrating thread model, other designs could provide much higher performance.

We will introduce a migrating RPC mechanism in which the kernel is provided with an interface definition describing the procedures to be called through that port, along with the parameters they take and their data types. The kernel will then be able to precompute optimized RPC paths specialized to individual procedures. RPCs can then proceed directly through the kernel with no redundant marshaling and unmarshaling stages; the kernel can intelligently follow pointers to by-reference arguments and copy them between the client and server as necessary, as well as match client-server argument semantics (e.g., allocation/deallocation), if necessary. This has many similarities to the mechanism used in LRPC[4]. In cases where protection domains have been merged[22], much of the copying can be avoided.

We will also allow this mechanism to be used in thread exception processing, replacing the `catch_exception_raise` mechanisms currently in use. This will allow a no-emulator server, such as the new version of OSF/1-MK[23], to directly provide the kernel with descriptions of Unix system calls to be redirected to the server. In handling these system call exceptions, the kernel will automatically handle the transfer of parameters, including by-reference data. This will obviate the need for the server to perform expensive kernel calls to emulate the many `copyin` and `copyout` operations during Unix system call processing.

10 Conclusion

We draw two main conclusions from our work. First, by changing only the thread model of an existing operating system, and evaluating the two versions, we show that a migrating thread model is superior to a static model. Migrating threads provide superior functionality, performance, and code simplification. In the area of functionality, thread migration (i) provides more powerful semantics for thread manipulation, and (ii) allows scheduling and other attributes to follow threads, especially important for real-time systems. In performance, thread migration (i) improves the performance of ordinary RPC, and (ii) enables a multitude of aggressive RPC optimizations, especially in systems under current research which provide cross-domain memory or address-space sharing. However, thread migration does have the performance disadvantage of not allowing user-level threads to service RPCs. In reducing implementation complexity, thread migration simplifies (i) kernel code, and (ii) server code. In each of these areas, our implementation and measurements have demonstrated the first benefit, while potential gains from the second seem evident, but have not yet been shown.

Our second main conclusion is that it is feasible to improve *existing* operating systems, by changing their thread model from static to migrating. Even in the case of Mach 3.0, which has an unusually rich thread-manipulation interface, we show that this far-reaching change can be made while retaining backward compatibility, and with only moderate implementation effort. A key element of that implementation is "basing" threads in the kernel, which temporarily make excursions into tasks via upcalls.

At this writing, in an intermediate stage of the work, some caveats exist. First, we need to measure the performance of the Unix server with wired `cthreads` for its "guest" migrating threads (or with replacement synchronization primitives). Until that is done, we cannot be sure the higher cost of kernel context switching does not cancel out the speedups we have demonstrated. Ideally, this experiment should also be performed on a multiprocessor. Secondly, since the current implementation has the drawback of using more memory for kernel stacks, and not completely supporting priority inheritance, it remains to be shown that removing those limitations does not add significant

complexity. And finally, until the Unix server is running, we have not sufficiently demonstrated backwards compatibility. However, we expect these issues to be favorably resolved.

Acknowledgements

We thank Mike Hibler for extensive discussion of controllability and signal issues, and Douglas Orr for general input.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] J. S. Barrera. A fast Mach network IPC implementation. In *Proc. of the Second USENIX Mach Symposium*, pages 1–12, 1991.

[3] Paul Barton-Davis, Dylan McNamee, Raj Vasswani, and Edward D. Lazowska. Adding scheduler activations to Mach 3.0. In *Proc. of the Third USENIX Mach Symposium*, pages 119–136, Santa Fe, NM, April 1993.

[4] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[5] A. P. Black, N. Huchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans on Software Engineering*, SE-13(1):65–76, 1987.

[6] Alan C. Bomberger and Norman Hardy. The KeyKOS nanokernel architecture. In *Proc. of the First USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, April 1992.

[7] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A tool for building efficient and flexible systems. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993. To appear.

[8] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. Technical Report UW-CSE-93-04-02, University of Washington Computer Science Department, April 1993.

[9] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[10] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.

[11] Dejan S. Milojičić, Wolfgang Zint, Andreas Dangel, and Peter Giese. Task migration on the top of the Mach microkernel. In *Proc. of the Third USENIX Mach Symposium*, pages 273–289, Santa Fe, NM, April 1993.

[12] David D. Clark. The structuring of systems using upcalls. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, WA, December 1985.

[13] Raymond K. Clark, E. Douglas Jensen, and Franklin D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *Proc. of the First USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Seattle, WA, April 1992.

[14] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wileknloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3, Winter 1990.

[15] Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 23(2):110–120, April 1992.

[16] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Asilomar, CA, October 1991.

[17] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.

[18] Bryan Ford, Mike Hibler, and Jay Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.

[19] Graham Hamilton and Panos Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.

[20] Dan Hildebrand. An architectural overview of QNX. In *Proc. of the First USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[21] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software — Practice and Experience*, 23(2):201–221, February 1993.

[22] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proc. of the Third USENIX Mach Symposium*, pages 39–55, Santa Fe, NM, April 1993.

[23] Simon Patience. Redirecting system calls in Mach 3.0: An alternative to the emulator. In *Proc. of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.

[24] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, December 1989.

[25] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Design rationale for Psyche, a general-purpose multiprocessor operating system. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 255–262, August 1988.

[26] M. Swanson, L. Stoller, T. Critchlow, and R. Kessler. The design of the Schizophrenic workstation system. In *Proc. of the Third USENIX Mach Symposium*, pages 291–306, Santa Fe, NM, April 1993.

Appendix A: Other Issues of Controllability and Migrating Threads

In this section we examine some common ways in which Mach's thread control facilities are useful, or are thought to be useful, and their conceptual and practical differences in a migrating threads system.

Thread Creation

Thread creation is done in Mach by first creating a thread with "unknown" CPU state, and then explicitly setting it to a known state before allowing it to run. This form of "thread control" does not present a problem either with static threads or with migrating threads, because a thread which has never had a chance to run cannot possibly be in a system call or have migrated anywhere.

Debugging

The basic CPU state primitives in Mach 3.0 allow a debugger to examine the state of a thread stopped at a breakpoint or some other point in user code, possibly change the thread's state, and to restart it again according to user commands. The debugger might also want to "catch" an already-executing but uncontrolled thread.

As traditionally defined, debugging is focused on the *particular task* being debugged; therefore, operations outside the task, even if being performed by the same thread, should be ignored and unaffected by the debugger. This means that **thread_abort** cannot safely be used to catch a thread and bring it under control, because that could damage the thread's execution. The existing **thread_abort_safely** call can be used instead, but if it fails, the debugger must keep retrying until successful or until the user gets tired of waiting.

A cleaner and simpler method is possible in our migrating threads design: the debugger can "post" a safe abort on the thread, and then block in the **act_get_state** call until the thread returns to that activation. No constant retrying is necessary, and if the thread takes an inordinately long time to return (i.e., something's wrong), the user always has the option just to terminate the task.

Transparent Cross-node Migration of Tasks and Threads

In this section we discuss encapsulating the state of a task and its activations, in order to transport them to another compute node, where they will be reinstantiated.

In an "ideal" completely controllable system, a thread can be transported to another node (probably as a part of task migration) completely from user mode by saving the state of the original thread, creating a new thread on the target node with the original thread's state, and finally destroying the original thread. However, in practice the same problems must be faced as with debugging— and in this case, it may not be acceptable to wait for an arbitrary amount of time for the thread to be migrated. For example, the node on which the thread is running may be shutting down, requiring immediate evacuation. Such situations can occur often in systems like Schizo[26], which allow "guest" tasks to use the resources of idle workstations without interfering with the workstation's normal functioning.

In practice, the ability of a thread to be transported can sometimes be ensured by limiting a task's operations to basic IPC primitives which can *always* be safely aborted, used in communication only with tasks on other nodes, or with other tasks which can also be migrated if necessary. This limitation could potentially hold on Schizo, for example, where a guest task has practically no interaction with local resources like device drivers.

Still other problems remain, however, such as page fault handling. On some architectures, such as the Motorola 680x0 series, the entire page fault exception state cannot be made visible to or modifiable by unprivileged code. This means that, for user-level task migration to work on such systems, page faults must be allowed to complete before threads are moved. A page fault may involve paging from an entity with unknown performance properties, possibly located an arbitrary distance away on a network, so it may still not be possible to guarantee quick task migration.

A working user-level task migration system for Mach has been created[11], but since it has no way of getting around this basic controllability problem, it can only work well most of the time.

There is no good solution to this problem short of providing some support in specially privileged code, for cross-node thread transportation. (It does not have to be the kernel, but it must be some entity which the kernel trusts.) Moving a thread must be a "primitive" operation of the distribution mechanism, in the same way that migrating an IPC port or a block of distributed shared memory is. In the case of Mach 3.0, to be reliable, thread movement would have to become part of the NORMA[2] code.

The situation under migrating threads is no different. Threads become distributed entities, conceptually "snaking through" the network following the thread's migrating RPC paths. As with static threads, specially privileged code must be involved, but this time only to transport individual activations on a thread rather than the thread as a whole.

Checkpointing

Another way in which highly controllable CPU state is thought to be useful is to provide persistence in the form of transparent checkpointing. If the state of every thread in a task, along with the task's other resources, can be captured and written to stable storage at regular intervals, then it can later be restarted at that point. In practice, the same problems occur as in task migration: the lack of complete controllability of threads may require delaying a checkpoint, and it may not be acceptable for such delays to be arbitrarily long.

Checkpointing presents an additional twist, however. In any checkpointing mechanism, a clear boundary must be drawn around the part of the system to be made persistent. Not everything can be persistent; for example, device drivers must at least know when the system has been restarted so they can reinitialize the hardware they control. Any communication involving entities outside the persistence boundary is subject to failure. In addition, if the persistence boundary includes multiple tasks, all communication among those tasks must also be made persistent.

In Mach 3.0 with static threads, this means that RPC messages directed from persistent tasks to non-persistent entities must be carefully monitored, so a proper failing reply message can be generated if the system is restarted at a checkpoint made during the RPC. If this were not done, the persistent task making the RPC request would hang after restart, awaiting a reply message that will never come. If multiple persistent tasks are involved, then any in-transit messages *between* them must also be checkpointed. In essence, to checkpoint under Mach 3.0, *having control* over a task would not be enough even if it were possible in its ideal form; it is also necessary to keep track of the IPC performed by the task.

In a migrating thread system, the situation is simpler. Only the activations of a thread in persistent tasks, and the connections between these activations, must be made persistent, not the entire thread. After a system restart, any non-persistent activations of a thread simply appear to have been terminated. The persistent parts of a thread can still function even if the thread has been split into multiple disjoint pieces— much like an earthworm that has been chopped up.

Unix is a trademark of USL. OSF/1 is a trademark of the Open Software Foundation.