Using Continuations to Implement Thread Management and Communication in Operating Systems

Richard P. Draves Brian N. Bershad Richard F. Rashid
Randall W. Dean

October 1991

CMU-CS-91-115R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A version of this paper appeared at the 13th Symposium on Operating Systems Principles (SOSP) held at Asilomar, CA. in October 1991. This technical report supersedes an earlier (March, 1991) version entitled "Continuations: Unifying Thread Management and Communication in Operating Systems."

Abstract

We have improved the performance of the Mach 3.0 operating system by redesigning its internal thread and interprocess communication facilities to use *continuations* as the basis for control transfer. Compared to previous versions of Mach 3.0, our new system consumes 85% less space per thread. Cross-address space remote procedure calls execute 14% faster. Exception handling runs over 60% faster.

In addition to improving system performance, we have used continuations to generalize many control transfer optimizations that are common to operating systems, and have recast those optimizations in terms of a single implementation methodology. This paper describes our experiences with using continuations in the Mach operating system.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the Open Software Foundation (OSF), the National Science Foundation (NSF). Draves was supported by a fellowship from the Fannie and John Hertz Foundation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.

Keywords: Operating Systems, Communication, Structure, Performance

1 Introduction

We have achieved significant improvements in the performance of the Mach 3.0 operating system kernel [Accetta et al. 86] by redesigning it to use continuations as the basis for control transfers between execution contexts. In our system, a thread blocks in the kernel in one of two ways. It either preserves its register state and stack and resumes execution by restoring this state, or it specifies its resumption context as a *continuation*, a function that the thread should execute when it next runs. By *allowing* a thread to block with a continuation, the kernel programmer can save space and time during thread management. Continuations enable a thread to discard its stack while blocked, thereby reducing the space required to support threads in the kernel. Continuations also allow a thread to present a high-level representation of its execution state while blocked, reducing control transfer overhead because the state can be examined and acted upon. By not *requiring* a thread to block with a continuation, the kernel programmer can rely on a traditional style of concurrent programming when using a continuation would be difficult.

Continuations have enabled us to reduce the storage requirements of the Mach kernel and improve its runtime performance. Compared to earlier, optimized versions of Mach 3.0 on a DECstation 3100, for example, our new system consumes over 85% less space per thread by making kernel stacks per-processor, rather than per-thread, resources. A cross-address space remote procedure call (RPC) executes 14% faster. Exception handling performance, crucial to the emulation of several non-Unix operating systems, has improved by a factor of two to three. Additionally, continuations enable many common control transfer optimizations to be recast in terms of a single implementation methodology.

The system described in this paper is in daily use by researchers at Carnegie Mellon and elsewhere. We believe that our technique of using continuations to manage control transfers will yield similar results for other operating system kernels that, like Mach, rely on threads and interprocess communication to support multithreaded programming and distributed computing [Mullender et al. 90, Rozier et al. 88, Thacker et al. 88].

1.1 Managing Control Flow in Operating System Kernels

In the past, operating systems have relied on one of two distinct models for executing within the kernel: the *process model* and the *interrupt model*. With the process model, the kernel's address space contains one stack for every thread in the system. When a thread traps into the kernel due to a system call or a fault, it uses its dedicated kernel stack to keep track of execution state. In this way, the thread can be descheduled and rescheduled at any time while executing in the kernel, since its entire state is saved while blocked. Unix [Ritchie & Thompson 78] is an example of an operating system that relies on the process model.

In contrast to the process model, the interrupt model treats system calls and faults like interrupts: all execution inside the kernel uses a single per-processor stack in the kernel's address space. Threads that block while in the kernel must first save information about their execution context. This saved information is used to later resume the blocked thread in an appropriate state. QuickSilver [Haskin et al. 88] and V [Cheriton 88] are examples of operating systems that rely on the interrupt model.

The main advantage of the process model is that it can be easily programmed — kernel threads have no "special" constraints on when they can block or reference pageable memory. Unfortunately, the process model has two performance problems. First, since each thread requires a stack in the kernel's address space, the process model can consume large amounts of memory. Second, because a kernel stack reflects the state of a blocked thread at the machine level in terms of return addresses, saved registers and automatic variables, it is difficult to evaluate that state and to implement optimizations that reduce the latency of transferring control from one thread to another.

1.2 A Brief History of Control Transfer in Mach

Early versions of the Mach operating system kernel relied on the process model for two reasons. First, the Mach kernel was patterned after Accent [Rashid & Robertson 81], which used the process model. With much of the overall design of Mach and portions of its code derived directly from Accent, it was natural for Mach to use Accent's process model as well [Rashid 86]. Second, early versions of Mach included a Unix compatibility layer which executed in kernel mode. This layer was implemented with software from BSD Unix [Leffler et al. 89], an operating system based on the process model. Significant programming effort would have been required to use a different control model while that code remained in the kernel.

As Mach evolved, it became clear that the process model was inappropriate. In Accent, thread management primitives were handled in microcode, so they were quite fast relative to the machine's CPU speed. In contrast, Mach was designed to run on a wide range of architectures. so we could not assume microcode to reduce thread management costs. Also, unlike Accent, Mach supports multiple threads of control per address space. This resulted in situations where a small number of programs might be using large numbers of kernel threads. With the process model, each of these threads would consume 4 kilobytes of stack in the kernel's address space. Finally, as the Unix compatibility code moved out of the kernel and into user space [Golub et al. 90], there was no longer a need to use the process model in the kernel purely to support Unix compatibility.

Our reevaluation of the process model was accelerated by our desire to efficiently support small systems, fast RISC uniprocessors, and multiprocessors. Although it is now common to find workstations equipped with 32 megabytes or more of memory, we wanted Mach to run efficiently on small PCs, laptops and notebook computers, which typically have less than 8 megabytes of memory. Consequently, it was critical to keep the kernel's memory requirements low. Furthermore, as processor speed increases, the relative cost of cache and TLB misses also grows. We expected that if we made kernel stacks more of a per-processor, rather than a per-thread resource, then the number of cache and TLB misses on references to kernel stacks would be reduced. Additionally, Mach runs on cache-coherent multiprocessors, and these machines are most efficient when dealing with per-processor data structures [Anderson et al. 89].

We were also concerned about the latency of transferring control from one thread to another. Our experiences designing fast interprocess communication (IPC) systems [Draves 90, Bershad 90] taught us several important lessons regarding low latency control transfer. We wanted to apply these lessons in a general way to other kernel-level control transfer paths. While low latency was important for the cross-address space RPC path, especially since most of the operating system was implemented at user level, other paths, such as exception handling, were also becoming important. Fast exception handling, for example, becomes necessary when using virtual memory primitives from user level [Appel & Li 91], or when emulating one operating system with another [Black et al. 91]. Moreover, because Mach runs on a wide variety of processor architectures, it was important that we could apply our general solutions in a machine-independent fashion — ad hoc assembly language solutions to performance problems were unacceptable.

1.3 Some Inadequate Solutions

Very early on, we realized that we needed to address the size and speed problems associated with managing large numbers of threads in the kernel. As our first solution, we modified our user-level threads package, C-Threads [Cooper & Draves 88], so that it multiplexed user-level threads on top of kernel-level threads [Golub et al. 90]. Our intention was to reduce the kernel space required to support large numbers of threads since one kernel-level thread could support many user-level threads in the same address space. Further, as noted elsewhere [Anderson et al. 91, Marsh et al. 91], user-level threads can also reduce the latency of switching between two threads when both are in the same address space.

Our use of C-Threads eased, but did not solve, the space problems of the process model. First, every address space still required at least one kernel-level thread because kernel-level threads cannot be shared between address spaces. Doing so in Mach would create substantial protection problems. Each emulated Unix program used one kernel thread as its Unix "process," and each multithreaded program running with C-Threads used at least one kernel thread as its "virtual processor." A second problem with putting user-level threads on top of the kernel's process model is that threads that blocked while executing within the kernel still consumed a kernel stack. As a result, we observed that C-Threads, on average, was only able to reduce the number of kernel-level threads in the system by about a factor of two.

User-level threads excel at improving performance when applications have little involvement with the kernel. While *multiprocessor* programs tend to stay away from the kernel, our experience with *multithreaded* programs such as servers has shown them to be kernel intensive, for example, because of IPC, page faults, and exceptions. So, while user-level threads partially address the space requirements of multithreaded programs, they alone are not enough. We needed a kernel solution to solve a kernel problem.

Our experience with other systems such as QuickSilver and RIG [Ball et al. 76] led us to consider the interrupt model, but we finally concluded that it was inappropriate for the Mach kernel. In its favor, the interrupt model consumes few kernel resources, since threads don't have dedicated kernel stacks. On the other hand, the model can be difficult to program, since every potentially blocking operation requires special-purpose code to save and restore state. Further, this code may have to peek over module boundaries so that a blocking module can save state for its callers, hampering a system's maintainability. For Mach, which runs on multiprocessors (i.e., locks are used internally), and supports virtual memory (i.e., the kernel can page fault), we felt that the interrupt model was unmanageable, and therefore unacceptable.

1.4 Restructuring With Continuations

We have restructured the Mach kernel so that a thread can use either the process model or the interrupt model when blocking. When a thread blocks using the process model, its current execution state is recorded on the stack. The blocked thread is resumed with a context-switch. When a thread blocks using the interrupt model, it records the execution context in which it should be resumed in an auxiliary data structure, called a *continuation* [Milne & Strachey 76]. The blocked thread is resumed by means of a call to the saved continuation. Our new approach offers several advantages over one that uses the interrupt model or the process model alone:

- *It provides the ease-of-use advantages of the process model.* A thread may block with its stack context intact at any time within the kernel. This is important when the interrupt model would "not be convenient," say because a thread is deeply nested in a function call chain when it blocks on a semaphore, or because it has taken a page fault while executing in the kernel.
- *It provides the performance advantages of the interrupt model.* When a thread has little or no kernel context, say because it is waiting to receive a message from another thread, or because the next instruction it should execute is in user space, it may relinquish its kernel stack entirely. Furthermore, since a continuation is accessed through a machine-independent interface, it is often possible to examine a continuation at runtime and avoid using it, because the system's current state makes its use unnecessary.
- *It provides a generalized framework and interface with which to implement many runtime optimizations found in other operating systems.* Many low-level optimizations associated with control transfer in operating systems can be recast in terms of continuations. For example, handoff scheduling [Black 90b, Thacker et al. 88], stackless kernel threads [Thacker et al. 88], asynchronous I/O [Levy & Eckhouse 89], kernel-to-user upcalls [Hutchinson et al. 89, Anderson et al. 91, Scott et al. 89], and Lightweight Remote Procedure Call [Bershad et al. 90] each represent an optimization to IPC and thread management systems that can be described and implemented in terms of continuations. Furthermore, by defining a machine-independent interface to continuations, these optimizations can be achieved with portable code.

We have used continuations to handle a variety of control transfers in the Mach kernel and have been able to improve system performance in a large number of places by applying a small set of optimizations in a uniform way.

In this paper we describe the use and performance of continuations in the Mach 3.0 operating system. In Section 2 we describe the implementation of continuations in Mach. We examine the performance improvements that result from using continuations and the optimizations they allow in Section 3. In Section 4 we show how continuations can be used to implement several control transfer functions found in other operating systems. In Section 5 we discuss related work. Finally, in Section 6 we summarize and present our conclusions.

2 Using Continuations in an Operating System Kernel

In this section we describe the kernel programmer's view of continuations, the steps we took to convert a process-based kernel into one that uses continuations, and some general optimization techniques made possible by continuations. We then show the influence that continuations have had on the implementation of several critical kernel services. Finally, we present the machine-independent interface for continuation management in the Mach kernel.

2.1 Creating Continuations

There are two kinds of control transfers that involve continuations: transfers that occur at the user/kernel boundary when a thread traps or faults out of user space and into the kernel, and those that occur within the kernel when one thread transfers control to another. System calls, exceptions and interrupts at user level transfer control to the kernel. The kernel entry routines create a continuation which, when called from the kernel, returns control to the user level. Control does not return to the caller. System calls generate a continuation which is invoked with the return code for the system call. Exceptions and interrupts, which do not return values to user programs, generate a continuation that is later invoked without arguments.

Within the kernel, a thread creates a continuation when it relinquishes its processor. It does this by passing a function pointer to the kernel procedure that blocks threads. The function becomes the thread's continuation and is stored in the kernel's machine-independent thread data structure. In the absence of further runtime optimizations, the thread resumes by calling its continuation.

Before Continuations

```
/* a frequently used system call */

example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block)  {
    /* use process model */
    thread_block();
    P2(arg1);
  } else  {
    P3();
  }
  /* return control to user */
  return SUCCESS;
}
```

After Continuations

```
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block)  {
    /* use continuation */
    save context in thread;
    thread_block(example_continue);
    /*NOTREACHED*/
  } else  {
    P3();
  }
  /* return control to user */
  thread_syscall_return(SUCCESS);
}

example_continue() {
  recover context from thread;
  P2(recovered arg1);

  /* return control to user */
  thread_syscall_return(SUCCESS);
}
```

Figure 1: Transforming a Blocking Kernel Procedure

A function specified as a continuation cannot return as normal functions do; it may only call other functions or continuations. This point distinguishes continuations from closures.

If the blocking thread must preserve any state while blocked, it must do so explicitly, as with the interrupt model. The kernel's thread data structure contains a scratch area large enough for 28 bytes of state. If a thread needs to save more state, it must allocate an additional data structure.

In those cases where it is not possible, convenient, or beneficial to block with a continuation, a null argument to the blocking function will cause the thread to block using the process model. The thread's context will be preserved on the stack, and the thread will resume in that same context.

2.2 Converting the Kernel to Use Continuations

Transforming the Mach kernel to use continuations was a straightforward process. First we identified those kernel functions that could potentially block and required only a small amount of state to be preserved while blocked. We then separated each of those functions into two parts: one before the block and one after. We defined a new function that consisted of the post-block, or continuing, part of the original function, and left only the pre-block part in the original. Next, we identified the stack context that was common to the two parts and modified the pre-block code to store that context in the blocking thread's scratch area. Similarly, we modified the post-block function so that it used the scratch context. In the pre-block function, we changed the call to the kernel's blocking function so that it passed the post-block function as an argument. That function serves as the thread's continuation. Lastly, we changed the post-block function to invoke a continuation on exit, rather than returning to its caller off of the stack. Figure 1 illustrates a sample transformation; one function becomes two, and the second function is named as an argument in the blocking part of the first.

In most cases we did not find it difficult to rewrite blocking kernel functions to use continuations. For user threads that trap into the kernel, the primary cases where blocking occurs are on message receives, exceptions, page faults, and preemptions. Each occurs as a result of a user-to-kernel transfer (system call, exception or interrupt), and each, upon being handled, returns control to the user level by way of the continuation that was created when control transferred into the kernel.

For threads that run only in the kernel, there is no "return-to-user-level" continuation. In practice, most of our kernel threads execute an infinite loop, blocking until an event occurs, doing some work, and then blocking again. For these threads, we define the continuation to be a function containing the body of the loop. The last statement in the function blocks with a continuation that is the function itself, thereby achieving the infinite loop via tail-recursion.

2.3 Optimization Techniques Using Continuations

Continuations enable three general control transfer optimizations that improve the performance of the kernel: *stack discarding*, *stack handoff*, and *continuation recognition*.

A continuation specifies the context in which a thread is to be resumed, so the thread's kernel stack can be *discarded* while the thread is blocked. This saves both space and time. It allows the kernel to save space because the stack of a blocked thread can be used by another thread. Further, if the *next* thread to run has blocked with a continuation and discarded its own stack, that thread can use the blocking thread's discarded stack directly. This second optimization is called *stack handoff*. By reducing the memory requirements of the kernel, continuations can also save time since they decrease the size of the kernel's working set, and therefore increase the effectiveness of caches and TLBs.

Time can be saved during a control transfer to a continuation by first examining the continuation at runtime (the saved continuation, which is a function pointer, can be compared to a set of known values). This technique is called *continuation recognition*, and it allows the kernel to use information that is available at the time a thread is resumed so that a more specific (and faster) code sequence can be used instead of the thread's continuation. Moreover, by not changing the stack pointer after a stack handoff, a resumed thread can execute that specific code sequence within the function call context of the blocking thread.

2.4 Using Continuations for Cross-Address Space RPC

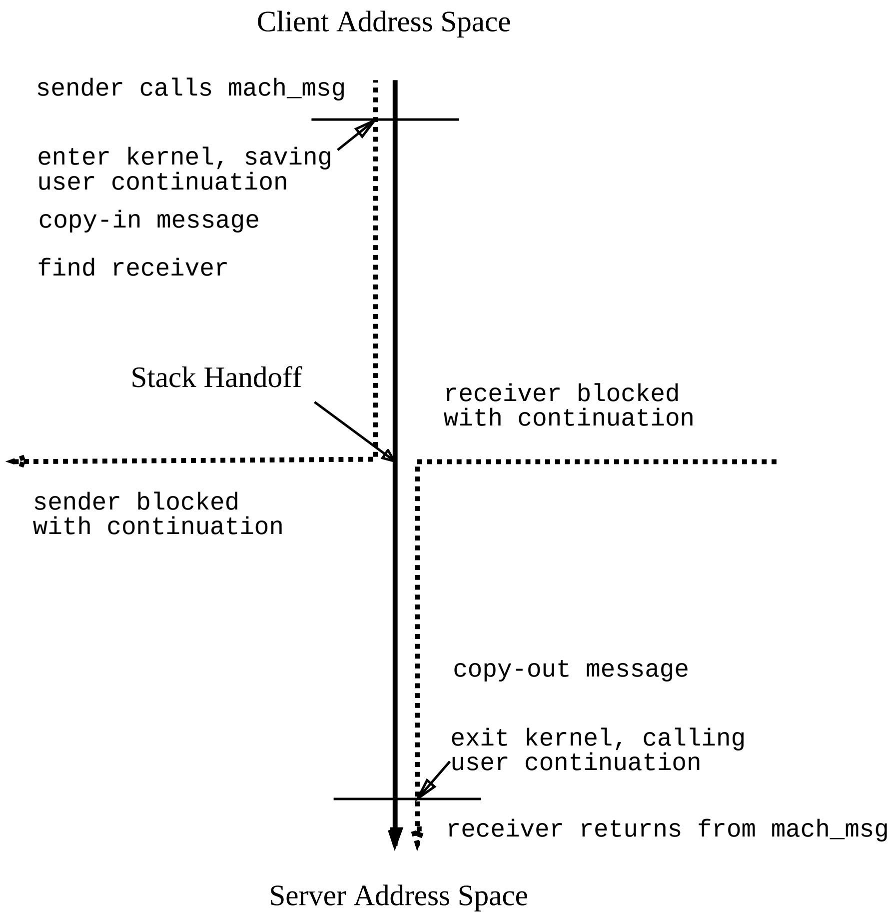

Figure 2: Calling Half of the Fast RPC Path

Operating system services in Mach are accessed by means of cross-address space RPCs to user-level servers. We have used continuations to restructure the kernel's RPC path to improve performance.

Figure 2 shows the fast path through the calling half of a Mach RPC.[1] A single system call, `mach_msg`, combines the sending and receiving phases of an RPC into one operation. A client thread uses `mach_msg` to send an RPC request message to a server, and to receive the reply message. A server thread uses `mach_msg` to send a reply message to the client, and to receive the next request message. In both cases, the sending thread wakes up the receiving thread and blocks itself until a message arrives.

The sending thread enters the kernel and creates a continuation that will return it to user level. It then looks for a thread that is able to receive the message. If it can't find such a thread, then a slow path is taken and the message is queued. If the sender does find the thread, it does a stack handoff to the receiver. This leaves the sending thread blocked with a continuation, `mach_msg_continue`, and no kernel stack. The handoff changes the currently running kernel thread to the receiving thread, but it does not immediately call the receiver's continuation. Instead, the receiving thread runs in the context of the sender's `mach_msg` system call. At this point, the thread checks its own continuation before using it. If it is `mach_msg_continue`, because the receiver had blocked in this same path, then `mach_msg` completes the fast RPC path and transfers out of the kernel into the receiver's address space. Otherwise, `mach_msg` calls the receiver's saved continuation to complete the message's processing within the kernel. This can happen, for example, when a receiver specifies unusual options or constraints, such as maximum size, on messages. This requires extra processing on every receive, so the receiver will be blocked with a different continuation that does further work. In practice, the extra work is rarely required, so most threads block with `mach_msg_continue`.

Using continuations reduces space and time overheads during RPC. Since almost all threads in Mach are normally waiting for a message, stack discarding frees their kernel stacks while they are blocked. Call latency is reduced because the handoff allows the blocking thread to communicate its still-active function call context to the resuming thread. The resuming thread may complete the control transfer by calling its previously stored continuation, or it may take an alternate fast path that takes advantage of information in the inherited context. Within this context, the sender's and receiver's message processing can be optimized together. For example, the fast RPC path avoids queueing and dequeueing the message, repeatedly synchronizing, and repeatedly parsing the message to check for exceptional conditions. Instead, the message is passed implicitly on the common stack, the sender locks shared data structures and the receiver unlocks them, and only the sender, not the receiver as well, checks the message for exceptional conditions such as whether the message is large.

2.5 Using Continuations for Other Types of Control Transfers

Although the RPC path is the most frequently traveled, we have also used continuations on several other kernel paths. Three of these paths, exception handling, thread preemption and user-level page faulting, illustrate the different ways in which continuations can be applied.

- *Exception Handling.* In Mach, every thread has an exception server that the kernel invokes with an RPC whenever the thread raises an exception [Black et al. 88]. For example, an exception is raised when a thread tries to write to a read-only page (because it's running in a shared virtual memory [Appel & Li 91]) or execute a privileged instruction (because Mach is emulating another operating system such as MS-DOS [Duncan 86] or the Macintosh OS [Rose & Hacker 85]). The exception server receives a request message from the kernel, attempts to handle it, and sends a reply message back to the kernel indicating the status of the just-taken exception. In most cases, the server can handle the exception and its reply message causes the kernel to immediately restart the faulting thread in user space.

  Unlike a user-to-user RPC, though, the kernel is an endpoint of communication when interacting with an exception server. We take advantage of this by using continuations to reduce the latency of exception-handling. Before entering the normal path for sending the exception message, the faulting thread, executing here in kernel mode, checks for a server thread waiting to receive a message (with the continuation `mach_msg_continue`). If the faulting thread finds a server thread, then it defers creating the request message and immediately does a stack handoff to the server thread, passing information about the fault directly on the shared stack. This avoids the copying, parsing, and queueing of the message that would occur if the message were sent in the context of the faulting thread and then received in the context of the server thread. If no server thread is waiting with `mach_msg_continue`, then this slower path is taken.

- *Preemptive Scheduling.*

  Thread preemption occurs during a clock interrupt when the current thread's quantum has expired. If the interrupted thread is running at user level, the stack context that the thread builds as it fields the

[1] The return phase is symmetric and works in the same way.

interrupt becomes unimportant, since the thread can simply resume execution at user level. Therefore, the preempted thread blocks with a continuation that returns the thread to user level when called.

Using continuations for preemption means that most runnable, but not actually running, threads do not require kernel stacks. It also reduces rescheduling latency slightly, since the preempted thread does not have to unwind its kernel stack after being rescheduled.

- *User-Level Page Faults.*

  When a thread faults on a non-resident page from user space, it must block in the kernel until a free physical page can be found and filled with data. The kernel's fault handler blocks the thread with a continuation that maps the new page and resumes the thread at user level. This avoids consuming stacks for faulting threads and conveniently reduces the kernel's memory consumption when there is little free memory available.

  This optimization only applies to user-level page faults. When a thread faults in the kernel, its kernel state and stack are preserved. Kernel-level page faults are an example of where it would be quite hard to use continuations since, in general, a thread can fault anywhere while executing in the kernel. Because of this difficulty, we simply fall back on the process model in this case.

Continuations are also used when threads voluntarily relinquish the processor from user level and when internal kernel threads block waiting for work. In the first case, there is no kernel state to save (as with preemptions). In the second case, we can use continuations to implement tail recursion using the technique described in Section 2.2.

2.6 Implementing Continuations in a Portable Operating System Kernel

`stack_attach(thread, stack, cont)`
: Transforms a machine-independent continuation into a machine-dependent kernel stack, attaching the kernel stack to the thread and initializing the stack so that when `switch_context` resumes the thread, control transfers to the supplied continuation function with the previously running thread as an argument.

`stack_detach(thread)`
: Detaches and returns the thread's kernel stack.

`stack_handoff(new-thread)`
: Does a stack handoff, moving the current kernel stack from the current thread to the new thread. `stack_handoff` changes address spaces if necessary. `stack_handoff` returns as the new thread.

`call_continuation(cont)`
: Calls the supplied continuation, resetting the current kernel stack pointer to the base of the stack. This function prevents stack overflow during a long sequence of continuation calls.

`switch_context(cont, new-thread)`
: Resumes the new thread on its preserved kernel stack. This call changes address spaces if necessary. If a continuation for the current thread is supplied, then `switch_context` does not save registers and does not return. Otherwise, `switch_context` saves the current thread's register state and kernel stack and returns when the calling thread is rescheduled, returning the previously running thread.

`thread_syscall_return(return-value)`
: Calls the current thread's user system call continuation to make the thread return to user space from a system call with the specified return value. (Low-level machine-dependent trap code creates system call continuations).

`thread_exception_return()`
: Calls the current thread's user exception continuation to make the thread return to user space from an exception or page-fault. (Low-level machine-dependent trap code creates exception continuations.)

Figure 3: Kernel Interface to Machine-Dependent Control Transfer Functions

```
thread_handoff(cont, new_thread) {
    old_thread = current_thread();
    old_thread->cont = cont;

    /* stack_handoff changes current_thread() */
    stack_handoff(new_thread);
    /* now current_thread() == new_thread */

    /* update scheduling state */
    old_thread->state = WAITING;
    new_thread->state = RUNNABLE;
}

thread_continue(old_thread) {
    new_thread = current_thread();
    thread_dispatch(old_thread);
    (*new_thread->cont)();
    /*NOTREACHED*/
}

thread_dispatch(old_thread) {
    if (old_thread->cont) {
        stack = stack_detach(old_thread);
        /* return stack to free pool */
        stack_free(stack);
    }
    if (old_thread->state == RUNNABLE)
        /* return old_thread to run queue */
        thread_setrun(old_thread);
}
```

```
thread_block(cont) {
    /* stop running the current thread */
    old_thread = current_thread();

    /* select a runnable thread from the ready queue */
    new_thread = thread_select();

    if (new_thread->cont) {
        if (cont) {
            /* stack_handoff changes current_thread() */
            stack_handoff(new_thread);
            /* now current_thread() == new_thread */

            old_thread->cont = cont;
            if (old_thread->state == RUNNABLE)
                /* return old_thread to ready queue */
                thread_setrun(old_thread);

            call_continuation(new_thread->cont);
            /*NOTREACHED*/
        } else {
            /* create a new stack */
            stack = stack_allocate();
            stack_attach(new_thread, stack,
                         thread_continue);
        }
    }

    thread_dispatch(switch_context(cont, new_thread));
}
```

Figure 4: Using the Control Transfer Interface

The Mach 3.0 kernel runs on a variety of processor architectures. This portability is achieved by dividing the kernel into machine-independent modules, which implement the Mach kernel interfaces, and machine-dependent modules, which manage the hardware. Machine-independent modules manage scheduling, inter-process communication, and virtual memory. Machine-dependent modules implement the low-level trap and exception machinery, handle the memory management unit, and export a new internal interface for manipulating stacks and continuations.

The new interface allows the machine-independent thread management and IPC modules to change address spaces, to manage the relationship of kernel stacks and threads, and to create and call continuations. The operations are listed in Figure 3. The interface does not include any functions for examining a blocked thread's continuation. It is stored in the kernel's machine-independent thread data structure, and can be examined directly by any other thread running in kernel mode.

The routines in Figure 3 are the building blocks with which higher level thread management operations are constructed. Figure 4 shows some of these operations and demonstrates the use of the interface within the kernel. The `thread_handoff` call gives control to a specific thread. It does a stack handoff and updates thread scheduling information to indicate that the old thread is blocked and the new thread is running. The function returns control running as the new thread, but does not call the new thread's continuation. This gives `thread_handoff`'s caller a chance to do continuation recognition, as is done on the RPC and exception handling paths. In contrast, the `thread_block` call chooses any runnable thread for execution. If the new thread has a continuation and `thread_block`'s caller has provided a continuation, then `thread_block` takes advantage of the more efficient `stack_handoff` path. Otherwise it must use `switch_context`, which changes stacks.

The implementation of `thread_block` illustrates the interaction of `switch_context`, `stack_attach`, and `stack_detach`. The `thread_block` function cannot detach and free the old thread's stack, or place the old thread on a run queue where another processor might find it, while executing on the old thread's stack. Therefore the implementation first uses `switch_context` to change to the new thread's stack and the new thread then uses `thread_dispatch` to dispose of the old thread. If the new thread doesn't already have a stack, then `stack_attach` initializes it to execute `thread_continue`, which calls the new thread's own continuation after disposing of the old thread.

2.7 The Advantage of a Kernelized System

We believe that the effectiveness of continuations in Mach stems from the fact that Mach is a "kernelized" operating system. It exports a small interface and implements only a few abstractions. As a result, there are few points in the Mach kernel where a thread can block, and even fewer where most threads actually do block. Although there are roughly 60 different points where a thread can block in the Mach 3.0 kernel, over 99% of the blocks occur at only six points. As would be expected, we have focused our reorganization on those few "hot spots." There are still paths in the kernel where continuations are not used, but they are traveled so infrequently that their effect on system performance is negligible.

In contrast, we believe that it would have been much more difficult to use continuations in a "monolithic" operating system such as Mach 2.5. That system implements the BSD Unix interface entirely within the kernel's address space. Most threads that block do so while executing deep within the kernel in the Unix compatibility layer. Generating a continuation for these threads would be difficult because they block only after building up a large amount of state on their kernel stack. Additionally, there are over 180 places in Mach 2.5 where a thread can block, and there are no real hot spots. For these reasons, we believe that there are few places where continuations could be used, and the overall space and time savings from using them would be small.

2.8 Software Engineering Concerns

A practical concern with continuations stems from their potential for overuse. Continuations should be applied judiciously to avoid ending up with an interrupt-model kernel that suffers from the problems described in the introduction. A key advantage of our use of continuations is that most of the kernel can work (and block) in the same way it did under the process model. Consequently, the bulk of the system is no less fragile than it was under that model. Code on paths that use continuations is fragile in the sense that it assumes certain things about its callers (e.g., if the code uses `thread_syscall_return`, it can only be used to implement system calls), but we view this as a reasonable price to pay given that we have been able to adopt a high-level approach to improving system performance.

2.9 Summary

Continuations represent one more case where leverage and uniformity can be gained by promoting an operating system abstraction to a first-class object — an object that can be named and manipulated by kernel code. In this sense, continuations are similar to Mach's *pmap* abstraction [Rashid et al. 87]. A pmap is a first-class object that reflects a sequence of address mappings from virtual to physical memory. By encapsulating the abstraction of memory mapping in a first-class object, and by separating the abstraction from its machine-dependent implementation (page tables and segment registers), the pmap interface is portable and can be used and optimized in ways that were not originally possible [Young et al. 87]. Continuations as a first-class kernel abstraction have yielded similar results.

3 Performance

In this section we examine the effect that continuations have on performance. In terms of space, we show that almost all control transfers in the kernel use continuations and are able to leave the blocking thread without a stack. This effectively makes kernel stacks into a per-processor resource. In terms of time, we also show that most control transfers use continuation recognition. This reduces the latency of cross-address space communication and user-level exception handling.

3.1 Experimental Environment

We measured three versions of the Mach kernel: MK32, MK40 and Mach 2.5. Both MK32 and MK40 are Mach 3.0 "pure" kernels in that they do not implement the Unix system call interface in the kernel's address space. The MK32 kernel does not use continuations, but includes optimizations that reduce the overhead of cross-address space RPC [Draves 90]. The MK40 kernel uses continuations as described in Section 2. Mach

Toshiba 5200 running MK40 and Unix emulation

Operations Using Stack Discard	Compile Test (22 secs) blocks	%	Kernel Build (4917 secs) blocks	%	DOS Emulation (698 secs) blocks	%
message receive	3113	83.4	1391769	86.3	200167	55.2
exception	0	0.0	882	0.0	137367	37.9
page fault	34	0.9	3278	0.2	144	0.0
thread switch	0	0.0	114	0.0	4	0.0
preempt	288	7.7	78602	4.9	19101	5.3
internal threads	239	6.4	135756	8.4	5791	1.6
total stack discards	3674	98.4	1610401	99.9	362574	100.0
no stack discards	60	1.6	2117	0.1	7	0.0

Table 1: Frequency of Stack Discarding with Continuations

Toshiba 5200 running MK40 and Unix emulation

	Compile Test count	%	Kernel Build count	%	DOS Emulation count	%
total blocks	3734	100.0	1612518	100.0	362581	100.0
stack handoff	3614	96.8	1608320	99.7	362567	100.0
recognition	2247	60.2	1166449	72.3	311277	85.9

Table 2: Frequency of Continuation Recognition and Stack Handoff

2.5 is a hybrid kernel that implements the BSD Unix interface in kernel space, does not include the RPC optimizations in MK32, and does not use continuations.

All kernels run on the DECstation 3100 (DS3100) and the Toshiba 5200/100. The DS3100 is a MIPS R2000-based workstation with separate 64K direct-mapped instruction and data caches and a four-stage write buffer. It has a 16.67Mhz clock and executes one instruction per cycle, barring cache misses and write stalls. Our DS3100 was configured with 16 megabytes of memory and a 250 megabyte Hitachi disk drive. The Toshiba 5200 is an Intel 80386-based laptop with a 20Mhz clock and a 32K combined instruction and data cache. Our Toshiba 5200 was configured with 8 megabytes of memory and a 100 megabyte Conner disk drive.

The Mach 3.0 kernel tests were run in an environment in which Unix system calls are implemented as RPCs to a Unix server. We also measured an MS-DOS emulation environment on the Toshiba 5200. The MS-DOS emulator catches the faults resulting from privileged instructions and MS-DOS system calls with a user-level exception handler. The exception handling thread runs in the address space of the emulated MS-DOS program.

3.2 Dynamic Frequency of Continuation Use

The value of continuations depends on the frequency with which they can be used. To determine this, we counted the number of blocking operations that used the continuations in three tests run on the Toshiba 5200 running the MK40 kernel. The first test measured a short C compilation benchmark. The second test measured a Mach 3.0 kernel build where all the files resided in AFS, the distributed Andrew File System [Satyanarayanan et al. 85]. The third test measured the MS-DOS program Wing Commander^{TM}, an interactive video game. The short compilation and MS-DOS tests were run with the machine in single-user mode. The kernel build was run in multi-user mode because AFS requires network services and a user-level file cache manager. Table 1 summarizes the results. On the DS3100, the frequencies are similar, with the exception of the MS-DOS game, which runs only on the Toshiba.

The table shows that about 99% of all control transfers use continuations and take advantage of stack discarding. The most frequent operations are message receive and exception handling. The other operations are page-fault handling, voluntary rescheduling [Black 90a], involuntary preemptions, and blocking by internal kernel threads. The remaining blocking operations (which do not use continuations) occur during kernel-mode page faults, memory allocation, and lock acquisition. Because generating a continuation for these cases is difficult, MK40, using the process model, preserves kernel stacks while threads block.

Table 2 shows that stack handoff occurs on nearly all control transfers. Moreover, continuation recognition, which can occur during cross-address space RPCs and exceptions, happens in over 60% of all blocking operations.

3.3 Time Savings Due to Continuations

	DS3100			Toshiba 5200		
	MK40	MK32	Mach 2.5	MK40	MK32	Mach 2.5
null RPC	95	110	185	535	510	890
exception	135	425	380	525	1155	1410

Table 3: RPC and Exception Times (in μsecs)

	MK40			MK32		
	instrs	loads	stores	instrs	loads	stores
system call entry	64	7	25	67	8	20
system call exit	35	21	1	24	11	1
stack handoff	83	22	18			
context switch				250	52	27

Table 4: Component Costs on the DS3100

We can show that continuations improve the runtime performance of cross-address space RPCs and exception handling. Our RPC test measures the round-trip time for a cross-address space "null" RPC. Our exception handling test measures the time for a user-level server thread to handle a faulting thread's exception. The exception server thread runs in the same address space as the faulting thread; it does not examine or change the state of the faulting thread, so the exception is retaken. The times for the two tests, averaged over a large number of iterations and running on MK40, MK32 and Mach 2.5, are shown in Table 3.

RPC Improvements

The RPC path in MK32 was already highly optimized relative to Mach 2.5, so there was little room for improvement. Although it uses the process model (that is, one stack per thread), MK32 avoids the general scheduler code during RPC transfers. Instead, it context-switches directly from the sending thread to the receiving thread. In contrast, Mach 2.5 queues messages and uses the general scheduling machinery to determine that the receiving thread is the next to run.

Despite the earlier optimizations, RPCs in MK40 are still 14% faster than in MK32. The improvement is mostly due to the stack handoff that replaces the more expensive context switch.[2] Table 4 illustrates the cost differential between stack handoff and context switch in terms of the number of instructions, loads, and stores required on a DS3100. The table shows that a handoff, which doesn't require a complete context save and restore, is substantially more efficient than a context switch.

Runtime Cost of Continuations

There is a small runtime cost associated with the use of continuations in Mach. As Table 4 shows, entering and exiting the kernel takes slightly longer in MK40 than in MK32. This is due to the interaction between continuations and architectural calling conventions. In MK32, the kernel's system call entry routine does not

[2] The Toshiba 5200's RPC latency increased slightly in MK40 because of a performance "bug" that is being fixed. The trap handler on the 5200 saves user registers on the stack during kernel entry, rather than in a separate machine-dependent data structure. As a result, the machine-dependent stack handoff procedure must copy the current thread's state from the stack and copy the new thread's state onto the stack. Once this is fixed, we expect that the Toshiba 5200 times will improve by approximately 50 μsecs.

need to save any user registers on the stack. Registers that are "caller-saved" have already been saved on the user-level stack, and those that are "callee-saved" will be saved on the kernel-level stack as necessary by the system call's compiler-generated prolog. That prolog implicitly assumes the process model and that callee-saved registers will be restored on return from the procedure that saved them. When continuations are used and stacks are discarded, though, a callee-saved register will not be restored on return (since the return never occurs). Consequently, the kernel entry routine must save all callee-saved registers in an auxiliary machine-dependent data structure, and the kernel's exit routine must restore them. The DS3100, for example, has 9 callee-saved registers to which the additional costs in Table 4 can be attributed. For exceptions and interrupts, the kernel entry routine must preserve all user registers, not just those that are callee-saved. This was necessary in MK32 as well, so the relative cost of aggressively preserving callee-saved registers decreases in these cases.

Exception Handling Improvements

As Table 3 shows, exception handling in MK40 is two to three times faster than in MK32. Unlike RPC, the exception handling path had not been optimized in MK32. Consequently, exception handling in MK40 demonstrates a "best case" result for continuations. It also illustrates an important point regarding the use of a general mechanism like continuations in an operating system kernel. Our need for a fast but portable cross-address space RPC mechanism motivated us to develop a general interface for handling control transfer efficiently. Once we had that interface, we were able to apply it easily to the exception handling path. In less than three days of work, we saw a 2-3 fold improvement in the runtime performance of exception handling. We also realized a space savings due to stack discarding. Further, because our optimizations were implemented using machine-independent code, they only had to be done once. Our experience with using continuations on other kernel paths has been similar. This has led us to conclude that we can apply continuations to performance-critical paths and get good results with relatively little effort.

3.4 Space Savings Due To Continuations

Continuations effectively change the kernel stack into a per-processor, rather than a per-thread, resource. For the three test programs, the number of kernel-level threads varied from 24 to 43. (In contrast, the machine on which we read our mail typically supports one to two hundred threads.) Using MK32, there would be as many kernel stacks as kernel-level threads. Using MK40, the number of kernel stacks was, on average, 2.002. Over 99% of the time only two stacks were in use: one for the currently running thread and one for an internal kernel thread that never blocks with a continuation. That thread's flow of control is such that a continuation is difficult to use. Its stack, though, represents a constant per-machine, and *not* per-processor, overhead[3]. The remaining .002 stacks were due to the fact that some control transfers do not use continuations (see the bottom row in Table 1). In the worst of circumstances, we saw the compile test and MS-DOS emulation use 3 stacks, and the kernel build use 6. In the steady state, however, only 2 kernel stacks were used.

	MK40	MK32
MI state	484	452
MD state	206	0
stack	0	4096
VM state	0	116
total	690	4664

Table 5: Thread Management Overhead on the DS3100 (in bytes)

Another way of evaluating the savings due to continuations is to consider the average amount of kernel memory consumed by each thread. Table 5 shows the size in bytes of the per-thread data structures maintained by the MK32 and MK40 kernels on the DS3100. On that machine, continuations reduce the average size of a thread by 85%. On the Toshiba, there is a comparable reduction.

The space required by a kernel-level thread includes machine-independent and machine-dependent state, and possibly a stack. In MK40, the machine-independent state has grown to include space for the continuation (a 4 byte function pointer), and the 28 byte scratch area, making it 32 bytes larger than in MK32. The machine-dependent thread state includes, for example, user registers that are saved when a thread enters the

[3] This special thread will be removed from the kernel in a future version of Mach.

kernel. In MK32, the thread's machine-dependent state is stored on the thread's dedicated kernel stack. In MK40, threads do not have a dedicated kernel stack, so the machine-dependent state is kept in a separate data structure.

The space consumed by a stack includes the stack itself (4 kilobytes), and any data structures used by the virtual memory (VM) system to maintain the stack in the kernel's address space. In MK32, kernel stacks are pageable, so they require an additional 116 bytes of VM data structures.[4] The MK40 kernel takes advantage of the fact that it is not necessary to page kernel stacks (since there are so few of them) and saves space in the VM system. Additionally, MK40 allocates stacks from physical memory on architectures where this is possible, freeing up a TLB entry for other purposes.

4 Generalizing Previous Optimizations with Continuations

Continuations provide a machine-independent framework with which to realize many of the control transfer optimizations found in other operating systems. As an example, we can compare Mach's continuation-based RPC to the control transfer aspects of Lightweight Remote Procedure Call (LRPC) [Bershad et al. 90].

LRPC is a high-performance interprocess communication facility designed for the common case of cross-address space RPC. Part of LRPC's good performance is due to the fact that threads can cross address space boundaries. A thread in the caller's address space traps into the kernel, but returns to the server's address space where it begins executing the server stub immediately. Upon return, the caller's thread traps back into the kernel from the server's address space and transfers back into the caller's address space at the instruction following the trap. The primary performance advantage of the single thread approach is that scheduling and message queueing can be avoided entirely on the fast LRPC path, since all work is being done in the context of a single thread.

Mach's continuation-based RPC achieves many of the same performance advantages as LRPC: no queueing, no scheduling, and sharing a kernel stack between the caller and the callee. In fact, the flow of control through the kernel "looks" similar in the two systems: control enters a kernel procedure from one address space and exits that same procedure into another. Further, continuation-based RPC maintains the logical separation between a client's thread and a server's. Threads remain fixed in their address space, eliminating many of the protection, debugging and garbage collection problems that occur when threads migrate between address spaces [Bershad 90].

A natural extension to the continuation model allows us to completely mimic the LRPC transfer protocol. By default, when a Mach thread traps into the kernel, it generates a continuation that will transfer control back to the same user-level context in which the trap occurred. We are experimenting with an extension to the IPC interface that enables a thread to register an overriding user-level continuation for system call returns.[5] This extension eliminates the cost of saving and restoring register state for the server thread and allows the server thread to discard its user-level stack while blocked waiting for an RPC request.

With the ability to return out of the kernel to a context other than the one that was active at the time the kernel was entered, continuations can be used to implement a rich collection of control transfer mechanisms in a general way. For example, the upcalls required by the *x*-kernel [Hutchinson et al. 89] and Scheduler Activations [Anderson et al. 91] can be implemented by keeping a pool of blocked threads in the kernel, each with a default "return-to-user-level" continuation. To perform an upcall, the default continuation is replaced with one that transfers control out of the kernel to a specific address at user level. Asynchronous I/O [Levy & Eckhouse 89] behaves in a similar fashion; on scheduling an asynchronous I/O, a thread provides the kernel with a continuation to be called when the I/O completes.

5 Related Work

The language community has been experimenting with continuations for almost two decades. Ward used continuations to define the primitives of a message passing algebra called mu-calculus [Ward & Halstead 80] and showed that all control transfer could be expressed in terms of that algebra. Functional languages that support concurrent execution and first-class continuations have been successful in implementing the former in terms of the latter [Wand 80, Haynes & Friedman 84, Cooper & Morrisett 90]. These efforts, however, have concentrated on control transfer at user level between contexts in the same address space. Functional languages

[4] Even when kernel stacks are pageable, threads run often enough that their stacks remain in memory. With MK32, for example, we found that over 90% of kernel stacks remained resident, even when the system paged other memory. When the stack of an idle thread is actually paged out, an *additional* 220 bytes of VM-related data structures per thread are required, so a non-resident stack consumes 336 bytes.

[5] This extension is not part of Mach's main-line release.

often use non-contiguous data structures to implement function call stacks, partially reducing the incentive to discard stacks. (A large portion of a kernel thread's discardable state is the unused stack space below the bottom-most active call frame.) Additionally, most functional languages make continuation recognition hard because they disallow equality comparisons of functional objects. Lampson [Lampson et al. 74] described a generalized control transfer interface based on continuations for an early version of the Mesa programming language [Geschke et al. 77].

A much restricted form of Lampson's interface later appeared in the cross-address space RPC implementation for Topaz [Schroeder & Burrows 90], an operating system designed for the Firefly, DEC SRC's experimental multiprocessor workstation [Thacker et al. 88]. This interface, implemented in assembly language, does a stack handoff, but does not use recognition or take advantage of a shared stack context. (SRC's RPC path is so heavily optimized that the stack contains no useful context because all values are kept in registers). Topaz also allows threads to discard their kernel stacks when blocking if they will execute in user space immediately after being rescheduled. Because Topaz implements the blocking component of semaphores and condition variables in the kernel, this is an important optimization for threads blocked on user-level events. Even with the optimizations for these two cases, though, there are still many places in the Topaz kernel where threads block using the process model and consume a stack. Recent measurements from a five processor, 96 megabyte Firefly at DEC SRC, for example, showed that 886 kernel-level threads were using 212 kernel stacks. Most of the stacks were being used by threads internal to the kernel (28), waiting for a timer to expire (106), waiting for a network packet (20), or waiting to handle an exception (38). While we would not expect continuations to improve the Firefly's cross-address space RPC path, we would expect a reduction in the amount of memory, cache, and bus bandwidth consumed by so many kernel stacks. In Mach, for example, 886 similarly blocked kernel-level threads would require only 6 stacks, one for each of the Firefly's five processors and one for a special kernel thread.

Operating systems implemented with the interrupt model, such as QuickSilver [Haskin et al. 88], V [Cheriton 88], and MS-DOS [Duncan 86], use the equivalent of continuations exclusively. The V kernel, for example, associates a "finish_up" function with a thread descriptor to allow the thread's computation to be resumed after it blocks. However, these systems' inability to use the process model as a "safety net" has made their internal structure complex; for example, page faults must generally be prevented while executing within the kernel, and even simple kernel-level locking necessary for multiprocessing is difficult [Cheriton 91].

We are not aware of any other system that combines the process model and the interrupt model, that uses recognition as a general optimization technique, or that treats the crossing of the user-kernel boundary as a continuation-based control transfer that can be affected by user-level applications.

6 Future Work and Conclusions

Our work with continuations in Mach is ongoing. We are presently experimenting with continuations at the application level within the context of C-Threads, our user-level threads package. We intend to allow user-level threads to use continuations, discarding their stacks and performing recognition when possible. For applications that do their own user-level scheduling and synchronization, we expect that continuations will reduce the space and time overheads normally associated with large numbers of user-level threads.

We are not the first to recognize the power and flexibility of continuations as a mechanism for describing and implementing the transfer of control between contexts. The novelty of our work lies in the fact that we have been able to apply continuations in a general-purpose operating system kernel. The use of continuations has allowed us to implement portably new optimizations, and to recast several optimizations found in other operating systems in terms of a single abstraction. As a result, we have achieved substantial improvements in system performance.

We believe that the methodology and techniques that we have described in this paper can be applied to other operating system kernels to achieve results similar to our own. We invite the reader to examine our system by obtaining the sources for the Mach 3.0 kernel via anonymous ftp from cs.cmu.edu.

Acknowledgements

We'd like to thank Tom Anderson, Lance Berc, John Carter, David Cheriton, Ed Felten, Bill Joy, Bob Kutter, Peter Lee, Tim Mann, Brian Marsh, Sape Mullender, John Ousterhout, Mike Schroeder, Dan Stodolsky, Andy Tanenbaum, Raj Vaswani, John Zahorjan, and Matt Zekauskas for their helpful discussions and comments on earlier drafts of this paper. We'd also like to thank Dave Redell and Mike Burrows for helping us to understand Topaz and for providing us with usage data from that system. Ed Lazowska's tireless shepherding greatly improved this paper.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Anderson et al. 89] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Anderson et al. 91] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy., H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991. This issue.

[Appel & Li 91] Appel, W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[Ball et al. 76] Ball, J. E., Feldman, J. A., Low, J. R., Rashid, R. F., and Rovner, P. D. RIG, Rochester's Intelligent Gateway: System Overview. *IEEE Transaction on Software Engineering*, 2(4):321–328, December 1976.

[Bershad 90] Bershad, B. N. *High Performance Cross-Address Space Communication.* PhD dissertation, University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195, June 1990.

[Bershad et al. 90] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Black 90a] Black, D. L. *Scheduling and Resource Management Techniques for Multiprocessors.* PhD dissertation, School of Computer Science, Carnegie Mellon University, July 1990.

[Black 90b] Black, D. L. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer Magazine*, 23(5):35–43, May 1990.

[Black et al. 88] Black, D., Golub, D., Hauth, K., Tevanian, Jr., A., and Sanzi, R. The Mach Exception Handling Facility. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 45–56, May 1988.

[Black et al. 91] Black, D. L., Golub, D. B., Julin, D. P., Rashid, R. F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G., and Bohman, D. Microkernel Operating System Architectures and Mach. *Journal of Information Processing*, December 1991. To appear.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[Cheriton 91] Cheriton, D. R. Personal Communication, May 1991.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C-Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[Cooper & Morrisett 90] Cooper, E. C. and Morrisett, J. G. Adding Threads to Standard ML. Technical Report 186, School of Computer Science, Carnegie Mellon University, December 1990.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the First Mach USENIX Workshop*, pages 101–121, October 1990.

[Duncan 86] Duncan, R. *Advanced MS-DOS: the Microsoft guide for assembly language and C programmers.* Redmond, Washington, 1986.

[Geschke et al. 77] Geschke, C., Morris, J., and Satterthwaite, E. Early Experiences with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Haskin et al. 88] Haskin, R., Malachi, Y., Sawdon, W., and Chan, G. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[Haynes & Friedman 84] Haynes, C. T. and Friedman, D. P. Engines Build Process Abstractions. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 18–23, August 1984.

[Hutchinson et al. 89] Hutchinson, N. C., Peterson, L. L., Abbott, M. B., and O'Malley, S. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.

[Lampson et al. 74] Lampson, B. W., Mitchell, J. G., and Satterthwaite, E. H. On the Transfer of Control Between Contexts. In *Lecture Notes On Computer Science: Proceedings of the Programming Symposium*, pages 181–203. Springer-Verlag, 1974.

[Leffler et al. 89] Leffler, S., McKusick, M., Karels, M., and Quarterman, J. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.

[Levy & Eckhouse 89] Levy, H. M. and Eckhouse, R. H. *Computer Programming and Architecture: The VAX-11 (2nd Edition)*. Digital Press, Bedford, MA, 1989.

[Marsh et al. 91] Marsh, B., Scott, M., LeBlanc, T., and Markatos, E. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991. This issue.

[Milne & Strachey 76] Milne, R. and Strachey, C. *A Theory of Programming Language Semantics*. Halsted Press, New York, 1976.

[Mullender et al. 90] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[Rashid & Robertson 81] Rashid, R. F. and Robertson, G. G. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[Rashid 86] Rashid, R. From RIG to Accent to Mach: The Evolution of a Network Operating System. In *Proceedings of the ACM/IEEE Computer Society 1986 Fall Joint Computer Conference*. ACM, November 1986.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

[Ritchie & Thompson 78] Ritchie, D. and Thompson, K. The UNIX Time-Sharing System. *Bell System Technical Journal*, July 1978.

[Rose & Hacker 85] Rose, K. and Hacker, B. *Inside Macintosh*. Addison-Wesley, Reading, MA, 1985.

[Rozier et al. 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

[Satyanarayanan et al. 85] Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., and Spector, A. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, December 1985.

[Schroeder & Burrows 90] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Scott et al. 89] Scott, M. L., LeBlanc, T. J., and Marsh, B. D. Evolution of an Operating System for Large-Scale Shared Memory Multiprocessors. Technical Report 309, University of Rochester, School of Computer Science, March 1989.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Wand 80] Wand, M. Continuation-Based Multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, August 1980.

[Ward & Halstead 80] Ward, S. A. and Halstead, Jr., R. H. A Syntactic Theory of Message Passing. *Journal of the ACM*, 27(2):365–383, April 1980.

[Young et al. 87] Young, M., Tevanian, Jr., A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.