Control Transfer in Operating System Kernels

Richard P. Draves

May 13,1994

CMU-CS-94-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

Thesis Committee:
Rick Rashid, Chair
Brian Bershad
Eric Cooper
Alan Demers, Xerox PARC

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. Support was also provided by a Fannie and John Hertz Foundation Fellowship and by Microsoft Corporation.

Keywords: operating system, kernel, control transfer, execution model, Mach, continuation

Abstract

Control transfer is the fundamental activity in an operating system kernel. The resource management functionality and application programmer interfaces of an operating system may be delegated to other system components, but the kernel must manage control transfer. The current trend towards increased modularity in operating systems only increases the importance of control transfer.

My thesis is that a programming language abstraction, continuations, can be adapted for use in operating system kernels to achieve increased flexibility and performance for control transfer. The flexibility that continuations provide allows the kernel designer when necessary to choose implementation performance over convenience, without affecting the design of the rest of the kernel. The continuation abstraction generalizes existing operating system control transfer optimizations.

This dissertation also makes two practical contributions, an interface for machine-independent control transfer management inside the kernel and a recipe for converting an existing operating system kernel to use continuations. The control transfer interface exposes enough functionality to let continuation-using code be machine-independent without sacrificing performance. It provides more functionality than the current state of the art, while still hiding the machine-dependent details of control transfer, such as switching register state and changing address spaces. The recipe provides a set of techniques and advice for converting existing code and writing new code with continuations. Taken together, these contributions form a blueprint for putting continuations into practice.

I have implemented continuations in the Mach 3.0 kernel from Carnegie Mellon University. This dissertation reports the performance improvements observed in that environment.

Acknowledgments

I would like to thank the people who have made this dissertation possible. My wife Martha provided me with motivation and encouragement; her support was crucial. My advisor Rick Rashid provided the environment and challenged me to grow. Brian Bershad’s early feedback greatly improved this work. I would also like to thank the other members of my thesis committee: Eric Cooper and Alan Demers. Alfred Spector gave me invaluable guidance in my first years at CMU.

The Mach project brought together a great group of researchers. I would like to single out Michael Young for tirelessly answering a neophyte’s questions and David Golub for his help with the machine-dependent hacking.

My fellow students made CMU SCS so hard to leave: Stewart Clamen, Tom Mathies, Marc Ringuette, Jay Sipelstein, Bennet Yee, Scott Draves, and everyone else who remembers BBQ night at G’s.

Finally, I would like to thank my coworkers at Microsoft for their patience and understanding: Joe Barrera, Bill Bolosky, Bob Fitzgerald, Alessandro Forin, Mike Jones, Steve Levi, and Gilad Odinak.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

My thesis is that a programming language abstraction, continuations, can be adapted for use in operating system kernels to achieve increased flexibility and performance for control transfer. The continuation abstraction generalizes many operating system control transfer optimizations. The framework that I’ve developed puts continuations to work in existing operating systems written in conventional programming languages.

Control transfer is the fundamental activity in an operating system kernel, the part of the operating system that must run in the processor’s privileged mode. As the privileged part of the system, the kernel manages control transfer between non-privileged address spaces. In contrast, other operating system functionality, such as resource management and application programmer interfaces, can be moved outside the kernel into other operating system modules. The current trend towards increased modularity in operating systems only increases the importance of control transfer in the kernel.

A continuation contains the saved state that represents a suspended computation or thread. Continuations were originally a mathematical abstraction invented to help define programming language control transfer semantics [Strachey & Wadsworth 74; Milne & Strachey 76]. Some modern programming languages have exposed first-class continuations, a general-purpose control transfer construct based on this abstraction. First-class continuations provide the programmer with tremendous power and flexibility in control transfer.

Applying the continuation abstraction to control transfer in operating system kernels, I have developed a framework that allows the state of a blocked thread to be represented in two different ways. When a thread blocks, the programmer can choose to use the natural machine-dependent representation for the continuation, saving the thread's register context and stack frames, or the programmer can choose to use a compact machine-independent representation, saving a continuation function and a structure with the important state variables. This framework replaces first-class continuations in conventional systems programming languages, such as C and C++, that lack native support for continuations. The flexibility of having a choice of thread state representations and the accessible nature of the machine-independent representation permits the optimization of common control transfer situations.

I have implemented continuation-based control transfer in the Mach 3.0 kernel. Compared to previous versions of Mach 3.0, the new system consumes 85% less space per thread—kernel stacks have effectively become a per-processor resource instead of a per-thread resource. The performance of control transfer operations important to the emulation of other operating systems on Mach improved; cross-address space remote procedure calls execute 14% faster and exception handling runs over 60% faster. The new system has been ported to many CPU architectures, including MIPS, Intel x86, Alpha, VAX, 88K, 68K, i860, ns532, HP PA, and Power PC.

1.1 Motivation

Because of the fundamental nature of kernel control transfer, its overheads and performance are an important determinant of an operating system's overall performance. The space overhead of the data structures that support control transfer, such as kernel stacks, and the latency of control transfer paths can both be significant. User-level thread packages can mitigate many control transfer performance problems, but they can only be partial solutions because they do not address kernel control transfer issues.

Control transfer is the fundamental activity in an operating system kernel because it provides the foundation for thread management, inter-process communication, page-fault handling, and exception handling. Only the kernel, as the privileged component of the operating system, can change in which address space a processor is executing. Only the kernel can trap and dispatch device interrupts, system calls, exceptions, and page faults.

Control transfer latency is a performance issue in several types of application environments. Others have already extolled the advantages of distributing system functionality across multiple address spaces [Rashid *et al.* 89; Bershad 90]; with such a distributed system communication and control transfer between address spaces becomes correspondingly more important. As another example, efficient page-fault handling becomes important when using virtual memory primitives from user level [Appel & Li 91]. Efficient exception handling becomes important when emulating one operating system with another [Black *et al.* 91].

Space overhead is also a performance issue in some application environments. This is certainly not obvious, given the exponentially decreasing cost of memory. However, in the marketplace for consumer devices, even a small amount of wasted or unnecessary memory puts a product at a disadvantage. It will either cost more or be less capable than competing devices. For small portable devices, power consumption and packaging issues make large memories problematic. The combined effect of these considerations means that memory remains a precious resource for portable consumer devices.

On high-end platforms with large main memories, space overhead can still be a performance issue. Caching in the memory hierarchy makes memory reference locality very important. If the kernel uses less memory on performance-critical paths, then it will probably suffer fewer cache and TLB misses on those paths. Furthermore, multiprocessors with cache-coherent memory perform better with per-processor data structures, which reduce cache contention. One effect of using continuations is that kernel stacks effectively become a per-processor resource instead of a per-thread resource. On multiprocessors that allow this, it becomes possible to allocate kernel stacks out of non-cache-coherent memory.

User-level threads are an important technique, but they do not go far enough. For example, the Mach project tried multiplexing user-level C threads [Cooper & Draves 88] on top of kernel-level threads [Golub *et al.* 90]. This reduced the kernel's memory usage (the primary goal of that work) and improved user-thread to user-thread control transfer latency, but because we hadn't attacked the source of the problem—control transfer in the kernel—the user-level threads package left most of the problem unsolved. A user-level solution works well for numerical multiprocessor programs, which tend to have little involvement with the kernel, but multithreaded programs like servers tend to be kernel-intensive, because of inter-process communication, page faults, and exceptions.

Scheduler activations [Anderson *et al.* 92] allow user-level threads to retain the advantages of kernel-level threads. In effect, scheduler activations provide a more powerful control transfer model for the user-level thread runtime. In contrast, continuations provide a more powerful control transfer model for the kernel itself.

1.2 Contributions

My dissertation makes four main contributions. First, I have taken a programming language abstraction and successfully applied it in a hostile environment, operating system kernels written in conventional programming languages. Second, I have generalized many existing operating system control transfer optimizations with a single unifying abstraction. Third, I have developed an interface for the machine-independent management of control transfer in the kernel. This interface offers more flexibility and performance than the existing alternatives. Fourth, I have developed a set of techniques for converting existing code and writing new code with continuations. Taken together, these contributions form a blueprint for putting continuations into operating systems practice.

In this dissertation, I provide a framework for using continuations in operating system kernels written in conventional programming languages. There is no question that the first-class continuations found in some programming languages constitute a more elegant expression of the continuation abstraction. Unfortunately, first-class continuations have two drawbacks in a kernel environment. First, operating system kernels today are written in systems programming languages such as C and C++, not modern programming languages. Producing an implementation of first-class continuations suitable for use in kernels, which have severe efficiency and concurrency constraints, is an unsolved research problem. Second, my framework provides some important optimizations that are not possible with first-class continuations. These optimizations operate when a thread blocks with a continuation function, which is an easily accessible machine-independent representation of the blocked thread's state. Because first-class continuations are opaque objects, they inhibit these optimizations.

Continuations generalize many existing operating system control transfer optimizations. For example, LRPC [Bershad *et al.* 90] provides a direct control-transfer path from a client address space to a server address space and back. Continuations not only allow the same direct path in cross-address space RPC, but also allow the same techniques to apply in other situations, such as exception and page-fault handling. As another example, the V kernel [Cheriton 88] avoids context-switches by using "finish-up" functions to specify the actions that a blocked thread will take when it resumes. Continuations not only allow this optimization, but also allow a thread to block with saved register context and stack frames when a function pointer can not conveniently represent the state of the thread.

In support of continuations, I have developed an interface for the machine-independent management of control transfer inside the kernel. The basic problem in control transfer management is to provide excellent performance in the face of problems such as wide variations in hardware "support" and synchronization issues on multiprocessors. The conventional solution is a ContextSwitch primitive that saves the register state of the current thread, changes address spaces if necessary, and restores the register state of the new thread. My interface provides additional functionality. It allows kernel stacks to be detached from and reattached to threads. It also provides a fast stack-handoff primitive that changes address spaces while avoiding the register saving and restoring overhead of a full context-switch. These features enable the space and time performance gains that arise from using continuations.

I have developed a methodology for using continuations effectively. This includes advice for selecting the important control transfer paths, depending on the performance goals, and for selecting from a set of coding techniques that I provide, depending on the structure of those control transfer paths. The advice and techniques apply both to converting existing kernel code to use continuations and writing new code with continuations.

1.3 Structure of the Dissertation

In Chapter 2 I survey control transfer from both operating system and programming language perspectives. Existing operating systems either have taken a process-based approach that uses context-switches for all control transfer, or have taken an interrupt-based approach that disallows context-switches inside the kernel. Meanwhile, the programming language community has developed continuations as a unifying abstraction for control transfer. User-level thread management based on first-class continuations has been successfully demonstrated. From this, I draw the conclusion that continuations are the appropriate unifying abstraction for control transfer in operating systems.

In Chapter 3 I examine the management of control transfer in the kernel. I discuss the design and implementation of a machine-independent interface for control transfer management which addresses issues such as wide variations in hardware “support” and synchronization on multiprocessors, while offering more flexibility than existing alternatives.

In Chapter 4 I provide a set of techniques and advice for using continuations. This chapter answers the practical questions about using continuations in an operating system kernel: “When should they be used?” and “How are they used?” Using MockIPC, an extended example based on Mach IPC, I demonstrate five general-purpose techniques for using continuations. I also briefly review the use of continuations in the Mach 3.0 kernel.

In Chapter 5 I examine the performance of continuations in the Mach 3.0 kernel. I show that 99.9% of all control transfer operations use continuations. As a result, the new system consumes 85% less space per thread, compared to previous versions of Mach 3.0. Effectively, kernel stacks have become a per-processor resource instead of a per-thread resource. In addition, there are latency improvements in the most frequent control transfer paths: cross-address space remote procedure calls execute 14% faster and exception handling runs over 60% faster.

Finally, in Chapter 6 I make some concluding remarks. In two appendices I provide more details on the actual Mach 3.0 interfaces that are relevant to the dissertation. Appendix A reviews the Mach 3.0 IPC system call interface and Appendix B presents a detailed view of the internal control transfer management interface.

Chapter 2

Control Transfer and Continuations

One can approach control transfer and continuations from both operating system and programming language perspectives. From an operating system perspective, a continuation-based framework for control transfer provides a way of optimizing the kernel environment to achieve the performance of event-driven kernels while preserving the software-engineering benefits of a multi-threaded programming style. In addition, continuations generalize a number of specific operating system optimizations. From a programming language perspective, first-class continuations provide a very powerful and general programming construct suitable for expressing operating system concepts such as multiprocessing and preemption. In this context, my framework for using continuations provides a pale approximation of this for traditional systems programming languages that do not support first-class continuations.

2.1 Control Transfer in Operating Systems

An operating system kernel has the responsibility of responding to device interrupts, system calls, and exceptions in such a way that it implements the user-level abstractions of threads and processes. Saving and restoring user-level CPU state on a per-thread basis accomplishes a large part of this, but does not address the question of an execution model for the kernel itself. Existing operating systems adopt one of two solutions: a process-based approach in which the kernel itself takes advantage of the process and thread abstractions to simplify its structure, or an interrupt-based approach in which the kernel handles interrupts, system calls, and exceptions on an equal basis in a purely event-driven manner.

An operating system is fundamentally an event-driven program with no life of its own. The operating system executes in response to interrupts or requests from other parties, such as hardware devices and software applications. Although an operating system may of its own volition occasionally execute "maintenance" code, this is the exception, not the rule.

Roughly speaking, the operating system kernel is the "privileged" part of the system. On processors that distinguish between a system mode and a restricted user mode, the kernel runs in the processor's system mode. Although some operating system components or applications may also execute in system mode for reasons of convenience or efficiency, this definition excludes them from the kernel.

Some functionality, such as filesystem implementations or network protocol stacks, may or may not belong to the kernel, depending on the design of the operating system. For example, in a Mach 3.0 system most filesystem and network code executes in user mode, either in a system process or as part of an application. However, in most Unix implementations the equivalent code resides in the kernel, and executes under the same constraints as other kernel code.

The stimuli that trigger execution of the operating system kernel may be classified as follows:

- *Device interrupts.* Most processors support an interrupt mechanism to let hardware devices get the attention of the operating system software, along with a corresponding way for the operating system to block interrupts temporarily. Because a device interrupt occurs without the cooperation of the currently executing code, the initial interrupt processing must save any CPU state that the interrupt handler's own execution might modify.

- *System calls.* Unprivileged software must sometimes request a service from the operating system kernel. In this case, the executing code has the opportunity to prepare in advance for the system call that it will make. This means that the system call processing in the kernel can in general save less register state, because the calling code can save and restore the remaining registers if it so wishes.

- *Exceptions.* Occasionally the processor may encounter an exceptional condition, such as a page-fault or an arithmetic error, and transfer to an exception handler in the kernel. Depending on the processor and the type of exception, it may or may not be possible to resume execution of the code that triggered the exception. From the kernel's point of view, exception processing resembles interrupt processing because exceptions are not anticipated, and the exception handler must save any CPU state that it might modify if it is possible to resume from the exception.

Unprivileged software executes in processes. As originally conceived [Saltzer 66], a process is a program in execution and as such combines an address space and a thread of control. The thread of control executes instructions, makes system calls, and takes exceptions. The operating system may multiplex the execution of multiple processes, but this is transparent to the processes involved.

Many operating systems extend the process concept to allow multiple threads of control. Programs that take advantage of this generalization must cope with unpredictably interleaved execution and possible actual concurrency on multiprocessor hardware. From the operating system's point of view, however, supporting multiple threads in a single address space does not change anything fundamental: the techniques for multiplexing execution and handling system calls and exceptions apply unchanged.

2.1.1 Process-Based Organization

Many operating systems organize the kernel's own execution around the same process and thread abstractions that the operating system implements for applications. Some examples include BSD Unix [Leffler *et al.* 89], Windows NT [Custer 93], OS/2 [Letwin 88], Sprite [Ousterhout *et al.* 88], Amoeba [Tanenbaum *et al.* 90], and Taos [McJones & Swart 89]. With this approach, the kernel enjoys the software engineering advantages of multi-threaded programming, with an accompanying performance cost.

The process-based organization gives threads an existence inside the operating system kernel. When the thread requests a service of the kernel, either explicitly with a system call or implicitly with an exception, the thread "enters" the kernel and proceeds to service its own request. While inside the kernel, a thread can access pageable memory, be preempted, and block for locks or other data structures to change state.

Device interrupt handlers, on the other hand, operate in a very restricted execution environment. Interrupt handlers must always run to completion without blocking, except for temporary preemption by higher-priority interrupts. If multiple interrupt handlers are active, their executions nest strictly.

This process-based approach has significant software-engineering advantages. Threads executing inside the kernel employ the same multi-threaded programming techniques as threads executing outside the kernel, with some additions to synchronize with device interrupts. The implementation of one system service may in turn use other system services without regard for their implementation, enabling good modularity and code reuse. The device interrupt handlers, which suffer from a much more constrained execution environment, should form only small part of the kernel.

From a performance viewpoint, the process-based approach suffers from two drawbacks. First, every thread needs its own stack for execution inside the kernel. Because interrupt handlers execute in a nested fashion, they can "borrow" the kernel stack of the currently executing thread, but if a thread blocks it needs its own kernel stack to preserve its execution state. Second, a context-switch between threads requires more effort, because kernel-level register state must be saved and restored as well as user-level register state.

2.1.2 Interrupt-Based Organization

An alternative execution model embraces the essential event-driven nature of the kernel and places interrupts, system calls, and exceptions on an equal footing. Some examples include V [Cheriton 88], QuickSilver [Haskin *et al.* 88], and QNX [Hildebrand 92]. This interrupt-based approach produces a light-weight implementation at the expense of significantly constraining the implementation, to the extent that it becomes very difficult for the kernel to provide some types of functionality.

The interrupt-based organization does not use threads inside the kernel. Instead of threads entering the kernel and doing things for themselves, the kernel executes and does things to threads. The notion of a "current thread" is non-existent (or at least very weak) in an interrupt-based kernel. When the kernel finishes executing in response to an interrupt, system call, or exception, the exit path restores the user-level register state of the thread at the head of the ready queue. This may or may not be the same thread that was executing when the kernel was entered.

Figure 2-1 gives interrupt-based and process-based implementations of the same two `read` and `write` system calls. The `read` system call dequeues and returns a character from a "`clist`." The `write` system call adds a character to the `clist`. The interrupt-based code explicitly shuffles threads between the scheduler's ready queue and a queue of waiting threads. It manipulates a thread's register state to set its return value from the `read` system call. In contrast, the process-based code uses `sleep` and `wakeup` scheduling primitives that implicitly affect the ready queue. The `read` system call returns a character in the normal fashion.

Interrupt-Based

```
void read(thread *thd) {
    if (clist empty) {
        remove thd from readyq;
        enqueue thd in waiters;
    } else {
        dequeue c from clist
        thd->rv = c;
    }
}
void write(char c) {
    if (threads waiting) {
        dequeue thd from waiters;
        enqueue thd in readyq;
        thd->rv = c;
    } else {
        enqueue c in clist;
    }
}
```

Process-Based

```
char read() {
    while (clist empty) {
        sleep(in waiters queue);
    }
    dequeue c from clist;
    return c;
}
void write(char c) {
    enqueue c in clist;
    if (threads waiting) {
        wakeup(from waiters queue);
    }
}
```

Figure 2-1: Interrupt-Based *vs* Process-Based

The interrupt-based implementation has several performance advantages. First, the kernel only needs one kernel stack for each processor instead of a kernel stack per thread. This reduces the memory overhead and improves the locality of the kernel. Second, the kernel never context-switches directly, so blocking system calls are less expensive. State is saved once at kernel entry time, instead of once at kernel entry and again inside `sleep`. Finally, the interrupt-based organization makes little optimizations easier. For example, in the `read`/`write` example the `write` system call can hand a character directly to a waiting thread; the character doesn't have to be enqueued and immediately dequeued.

However, interrupt-based implementations also suffer from software engineering disadvantages. Most interrupt-based systems do not support demand-paged virtual memory, because the kernel can not access pageable memory in a natural way. Kernel-mode computations are not preemptible or otherwise subject to the scheduling algorithms. Heap memory allocation and locking for multiprocessors become more difficult. Considering Figure 2-1 again, it is clear that the interrupt-based organization makes simple code complex.

The developers of the V system experienced these disadvantages. Because V uses an interrupt-based organization "everything had to be handcrafted, and was difficult to maintain" [Cheriton 91]. The Munin distributed shared memory system [Carter *et al.* 91] uses a V kernel, substantially modified to support paged virtual memory. Because of the difficulties caused by kernel-mode page faults, Munin modified V's original execution model [Carter 93]. In Munin, when the kernel takes a page fault, the current kernel stack is placed in an exception queue and a new kernel stack is allocated.

2.1.3 Other Organizations

A few operating systems use variations on the basic process-based and interrupt-based organizations. One such variation uses dedicated server threads inside the kernel to field requests for kernel services. Another dynamically spawns threads inside the kernel to handle interrupts. One can imagine further variations along these lines, but the essential distinction between multi-threaded and event-driven programming paradigms remains valid.

The Minix kernel [Tanenbaum 87] contains a small number of threads dedicated to performing different tasks. Most of these threads manage hardware peripherals; a special system thread implements much basic operating system functionality. A light-weight message-passing mechanism allows applications to communicate with each other and with internal kernel threads. The kernel turns device interrupts into messages sent to the appropriate device thread.

At heart, Minix uses an interrupt-based organization. The message-passing code executes on a dedicated kernel stack as described in the previous section; this may be regarded as the “true” Minix kernel. The internal kernel threads execute in the kernel address space but otherwise operate like independent user threads.

The SunOS 5.0 kernel [Eykholt *et al.* 92; Khanna *et al.* 92] extends the process-based organization to encompass device interrupts. In this system, device interrupts spawn “interrupt threads” that terminate when interrupt processing is complete. This removes some of the awkward restrictions on interrupt handlers that exist in most process-based kernels. The SunOS kernel does continue to block interrupts of equal and lower priority until an interrupt thread terminates, so interrupt threads can not for example access pageable memory.

2.2 Generalizing with Continuations

Using continuations, I have developed a control transfer framework that generalizes the process-based and interrupt-based kernel organizations. The continuation technique starts with a multi-threaded programming model, like the process-based organization, but then gives programmers the possibility of optimizing important control transfer situations to achieve the performance of an interrupt-based organization. This continuation-based framework improves upon a number of related control-transfer optimizations:

- Finish-up functions in V [Cheriton 88].

- Stack-less threads in Taos [McJones & Swart 89].

- Direct cross-address space control transfer in LRPC [Bershad *et al.* 90].
- Deferred procedure call in Windows NT [Custer 93].
- Rendezvous optimization in Ada [Habermann & Nassi 80].

After introducing continuation-based control transfer for operating system kernels, I examine each of these related optimizations in more detail in subsequent subsections.

Continuations

My framework for using continuations gives programmers a choice of representations for a blocked thread's continuation. A continuation saves the state of a thread's computation; it controls the thread's subsequent execution. In the normal case, the thread's continuation consists of stack frames and register context saved on the kernel stack. However, the programmer may also specify an explicit continuation function. In this case, the thread resumes execution by calling its continuation function, discarding its previous execution context.

Explicit Continuation

```
void read() {
    while (clist empty) {
        sleep(in waiters queue,
            read);
        // NOTREACHED
    }
    dequeue c from clist
    thread_syscall_return(c);
}
void write(char c) {
    enqueue c in clist;
    if (threads waiting) {
        wakeup(from waiters queue);
    }
}
```

Implicit Continuation

```
char read() {
    while (clist empty) {
        sleep(in waiters queue,
            NULL);
        // resume here
    }
    dequeue c from clist;
    return c;
}
void write(char c) {
    enqueue c in clist;
    if (threads waiting) {
        wakeup(from waiters queue);
    }
}
```

Figure 2-2: Continuation-Based Control Transfer

Figure 2-2 illustrates this choice with the `read`/`write` example of the previous section. The `sleep` primitive now takes an additional argument, a continuation function. If this is `NULL`, then the `sleep` function returns normally when the thread unblocks. If the continuation function is not `NULL`, then `sleep` does not return. In this particular example, `read` is called again instead.

The implementation that sleeps with an explicit continuation function also enables internal stack discarding and stack handoff optimizations. While the thread calling `read` is blocked with a continuation function, its kernel stack is not needed and can be discarded. If it context-switches to another thread that blocked with a continuation function and hence has no kernel stack, the kernel stack can be directly transferred to the resuming thread with a stack handoff. If the predominate control transfer paths use explicit continuations, then the aggregate performance will approximate that obtainable with an interrupt-based organization because most of the time a single kernel stack will suffice for the system's threads and the context-switch overhead of saving and restoring registers will be avoided.

The use of explicit continuation functions also enables continuation recognition optimizations. Continuation recognition occurs when an executing thread examines and possibly modifies the continuation of a blocked thread. In the `read`/`write` example of Figure 2-1, the interrupt-based code was able to hand a character directly from the writing thread to the reading thread. Continuation recognition could achieve the same optimization in the explicit continuation code of Figure 2-2 by having the writing thread check for a `read` continuation in the blocked reader. (Although in this example, the benefit would not be worth the bother.)

2.2.1 V System: Finish-Up Functions

The V system [Cheriton 88], an experimental message-passing operating system from Stanford University, uses an interrupt-based organization, augmented with continuation functions (known as "finish-up" functions in the V source code) [Cheriton 91]. Although the finish-up functions make the interrupt-based organization somewhat easier to deal with, they do not alleviate its fundamental software engineering problems or change its performance characteristics. My continuation-based control transfer framework improves upon V's approach by starting with a more powerful multi-threaded programming model and then allowing threads to block with a continuation function.

The V kernel uses an interrupt-based organization. Machine-independent code uses no "get current thread" function; instead, a thread-descriptor pointer is explicitly passed to those functions that need it.[1] Inside the kernel, the `AddReady` function adds a thread to the ready queue (a single queue, sorted by priority). `RemoveQueue` removes a thread from whatever queue it might be in, including the ready queue. Upon return from the kernel to user space, the function `ActivateReadyqHead` removes a thread from the head of the ready queue, and the kernel switches to its address space. At this point, the kernel checks for a finish-up function associated with the thread. If the thread has a finish-up function, the kernel calls it. Otherwise the kernel restores the thread's registers and returns to user mode.

[1]In the V system's terminology, the executable entities are processes. A group of processes that share an address space form a team.

The finish-up function allows V to associate with a thread code that will access the thread's user address space. For example, copying a message out to a user-mode buffer is easily done in a finish-up function. Without finish-up functions, the V kernel would need more code to explicitly manage changes to the current address space.

2.2.2 Taos: Stack-Less Threads

Taos [Schroeder & Burrows 90; McJones & Swart 89], an operating system designed for the Firefly, DEC SRC's experimental multiprocessor workstation [Thacker *et al.* 88], uses a process-based organization. In addition, Taos allows threads to discard their kernel stacks when blocking if they will execute in user space immediately after being rescheduled. Because Taos implements the blocking component of mutex and condition variables in the kernel, this is an important optimization for threads blocked on user-level events. In addition, the cross-address space RPC path, implemented in assembly language, does a stack handoff. (SRC's RPC path is so heavily optimized that the stack contains no useful context because all values are kept in registers.) Even with the optimizations for these two cases, there are still many places in the Taos kernel where threads block using the process model and consume a kernel stack.

Measurements from a five processor, 96 megabyte Firefly at DEC SRC, for example, showed that 886 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).

Continuation-based control transfer generalizes the optimizations found in Taos. Continuations allow kernel stacks to be discarded in more situations, and the stack management functionality in the control transfer interface allows machine-independent code to take advantage of stack handoff. In addition, continuations enable the general-purpose latency-reducing technique of continuation recognition.

2.2.3 LRPC: Direct Cross-Address Space Control Transfer

LRPC [Bershad *et al.* 90; Bershad 90], a light-weight cross-address space procedure call technique prototyped in the context of the Taos operating system, supplies a very direct control-transfer path between client and server address spaces. LRPC bypasses all scheduling and context-switch overhead in the kernel. Threads using LRPC have user-mode execution stacks cached for their use in the address spaces that they frequently visit. When a thread makes a cross-address space call, it traps into the kernel and returns to user-mode in the server address space with an upcall running on its execution stack there. The return path from server address space to client is equally direct.

A continuation-based cross-address space procedure call can achieve a very similar direct control-transfer path. Instead of having execution stacks waiting in server address spaces, the server has a pool of server threads. Because the server threads are blocked with a continuation function and do not have kernel stacks, they really exist for their user-mode stacks. When a client thread traps into the kernel, a stack handoff to a waiting server thread avoids scheduling and context-switch overhead and continuation recognition avoids leaving the fast path to call a general-purpose continuation function. Because this leaves the client thread blocked with a continuation function, the return path operates equally efficiently. The major difference occurs in the entry to the server address space: this is a return from a system call instead of upcall as with LRPC. However, support for upcalls could remove even this difference.

Furthermore, the continuation-based approach generalizes easily to optimize situations other than cross-address space procedure call. The same efficient stack handoff control-transfer path with continuation recognition can be used in situations such as exception and page-fault processing. Continuation-based RPC also 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].

2.2.4 Windows NT: Deferred Procedure Calls

Windows NT [Custer 93], Microsoft’s general-purpose commercial operating system, uses a process-based organization. To reduce the amount of code executed at interrupt level and improve the system’s real-time characteristics, interrupt handlers in Windows NT defer much of their processing with a Deferred Procedure Call (DPC) mechanism. In essence, the DPC mechanism allows an interrupt handler to block with a continuation function.

Interrupt handlers in Windows NT use DPC objects to defer processing until after the processor’s interrupt request level (IRQL) is reduced. Interrupt handlers execute with an elevated IRQL, which blocks out interrupts of equal and lower priority. If an interrupt handler performs extended computations at an elevated IRQL, the system’s responsiveness suffers. To avoid this, an interrupt handler can initialize a DPC object with a function and queue the DPC. When the processor’s IRQL lowers to allow interrupts, the Windows NT kernel automatically executes any queued DPC functions. Because they borrow the kernel stack of the current thread, like interrupt handlers, DPC functions execute in a similarly constrained execution environment.

The DPC mechanism effectively allows an interrupt handler to block with a continuation function. However, DPC functions do not execute as independent threads. With a more powerful continuation-based organization, Windows NT could execute DPC functions in interrupt threads and remove the current restrictions on their execution environment without losing performance.

2.2.5 Ada: Rendezvous Optimization

The Ada programming language contains built-in multiprocessing primitives that can sometimes benefit from control transfer optimizations. One such optimization for the Ada rendezvous operation [Habermann & Nassi 80] amounts to a stack handoff to a virtual server thread. In contrast, continuation-based control transfer not only makes explicit the notion of stack handoff but allows the use of stack handoff in much more general situations.

In Ada's execution model, a process consists of a set of threads[2] that communicate via a rendezvous mechanism. A server thread exports some number of entry statements. Client threads call entries using the normal procedure call syntax. Server threads execute `accept` statements for entries; in the body of the `accept` statement formal parameters are bound to the client's actual arguments. If the entry call happens before the `accept` then the client waits and if the `accept` occurs before an entry call then the server waits. After the rendezvous, or execution of the body of the `accept`, the client and server threads resume normal concurrent execution.

Although Ada semantics have the server thread execute the body of the `accept` statement, the compiler can arrange for the client thread to execute the `accept` body with identical results. If the server thread does nothing outside the `accept`—after the rendezvous it loops back and reenters the `accept`—then the need for the server thread as a distinct executable entity vanishes. With this optimization, entry calls effectively become monitor calls.

When the server thread has been optimized away, the rendezvous operation can be viewed as a stack handoff to a virtual server thread. To start the rendezvous, the client thread does a stack handoff to the virtual server thread. The virtual server thread executes the body of the `accept` on the client's stack. Finally, the virtual server thread does a stack handoff back to the client thread.

2.2.6 Other Optimizations

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* 92] can be implemented by keeping a pool of blocked threads in the kernel, each with a default "return-to-user-level"

[2]The Ada terminology is *task*.

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.

2.3 Control Transfer in Programming Languages

The notion of a continuation arose in the study of programming languages [Strachey & Wadsworth 74]. Theoreticians invented the continuation abstraction as a way of expressing the semantics of control transfer. A continuation represents the control state of a computation; it answers the question "What will happen next?" Some modern programming languages support first-class continuations, a programming construct that allows the programmer to "capture" and manipulate the continuation of an arbitrary expression in the language. Using first-class continuations, one can implement multiprocessing [Wand 80], coroutines [Haynes *et al.* 84; Haynes *et al.* 86], timed preemption [Haynes & Friedman 84; Haynes & Friedman 87; Dybvig & Hieb 89], and user-level threads [Cooper & Morrisett 90]. Recent work of interest to operating systems in the areas of control delimiters [Sitaram & Felleisen 90] and functional continuations [Felleisen *et al.* 88] addresses short-comings of the usual first-class continuation formulations in the areas of modularity and embedding.

2.3.1 Abstract Continuations

Denotational semantics [Milne & Strachey 76] ascribes meanings to programs in such a way that the meaning of a program fragment depends only on the meanings of its constituents. It uses valuations, or functions mapping program fragments to values (meanings) in domains, which are recursively-defined sets of functions on values, locations, stores, environments, *etc*. The valuation functions themselves are defined recursively on the structure of program fragments such as expressions and commands.

Denotational semantics uses continuations to express control transfer in expressions and commands. For example, consider a compound command C_0; C_1. The meaning of the compound command is a function defined in terms of the meanings of the two constituent commands C_0 and C_1. However, a simple composition can not express the idea that C_0 may contain a non-local jump or exit and under some circumstances C_1 will never be executed. To handle this possibility, the function that is the meaning of a command takes as one argument a command continuation; the command continuation is itself a function that represents the subsequent execution of the program. (In Milne's formulation, a command continuation is a function from stores to answers.) Then the meaning of a compound statement like C_0; C_1 can be defined by applying the meaning of C_0 (a function) to an argument, an intermediate command continuation that is defined by applying the meaning of C_1 to the command continuation of the entire compound statement.

In the same way, denotational semantics uses expression continuations to define the valuation function for expressions. An expression continuation is a function from expression values to command continuations. The expression value input is needed because expressions, unlike commands, return values and hence the continuation for an expression must accept a value.

2.3.2 First-Class Continuations

Some modern programming languages, such as Scheme [Steele 78; Clinger & Rees 92] and ML [Appel & Jim 89], provide support for first-class continuations. Programmatic continuations give the user direct access to the abstract continuations that underlie the semantics of the programming language. A programmatic continuation is “first-class” if it has indefinite extent and lifetime: it may be passed as an argument, returned as a function’s value, stored in a global data structure, and invoked multiple times. This flexibility makes first-class continuations a control transfer primitive of great power.

In Scheme support for first-class continuations takes the form of a built-in function, `call-with-current-continuation` (abbreviated here `call/cc`). Scheme packages a continuation as a functional object that may be called with a single argument. However, the call of a continuation does not return. Instead, it restores the control context that existed at the time that the continuation was created with `call/cc`.

For example,

```
(define example
   (lambda (x)
      (call/cc (lambda (k)
         (+ 1 (* 2 (k x)))))))

(+ 5 (example 3)) ⇒ 8
```

defines `example` to be a function of one argument `x`. The application of `call/cc` packages its own continuation as a functional object and then calls its argument, passing it the continuation, which in this case is bound to `k`. When `k` is called, its application control context `(… (* 2 …))` is discarded and control transfers to the context in which the continuation was created: `(+ 5 …)`.

The representation of continuations in Scheme and ML makes them opaque objects. These languages do not supply any operations to examine, manipulate, or modify a continuation.

Smalltalk and MPL

Smalltalk [Goldberg & Robson 83] supports a simple form of continuations. The Smalltalk interpreter stores its state in Contexts; the execution of a non-primitive message creates a new Context. A Context includes an instruction pointer, a stack and stack pointer, and a pointer to a parent Context. Smalltalk Contexts are first-class objects; they support methods that manipulate their contents and are garbage-collected. However, the interpreter prevents a Context from being returned from more than once.

The Modular Programming Language, a precursor of Mesa [Geschke *et al.* 77], supported similar execution contexts with a general `transfer` primitive [Lampson *et al.* 74]. Lampson demonstrated that higher-level control disciplines could be expressed using contexts and `transfer`.

2.3.3 Continuation-Based Multiprocessing

First-class continuations can implement multiprocessing, timed preemption with engines, and user-level threads. These techniques all use continuations to represent the state of a computation; `call/cc` saves the state of the current computation and invoking a continuation context-switches to another computation.

Wand [Wand 80] describes continuation-based multiprocessing. With his technique, the “kernel” consists of a function that manipulates a ready queue hidden inside a closure (and hence protected from the rest of the program). The kernel supports two operations: enqueue a process on the ready queue, and dequeue and dispatch a process. A process consists of a command continuation; because Scheme does not support command continuations directly the kernel uses a thunk that invokes a continuation from `call/cc` with a specified value. To dispatch a process, the kernel calls the thunk which in turn invokes a continuation.

A coroutine implementation in terms of continuations uses a similar technique [Haynes *et al.* 84; Haynes *et al.* 86]. A coroutine can suspend itself and transfer control to another coroutine. The transfer operation takes a value that is returned as the result of the resuming coroutine’s suspended transfer operation. Implementing this with continuations, a coroutine consists of a function that manipulates a saved continuation hidden inside a closure. The continuation represents the suspended execution state of the coroutine. This resembles Wand’s multiprocessing, except that instead of a kernel with a queue of continuations each coroutine maintains its own private execution state.

Engines [Haynes & Friedman 84; Haynes & Friedman 87], a control construct that provides timed preemption, can also be implemented with continuations [Dybvig & Hieb 89]. An engine is a functional wrapper around some code. When an engine is invoked, it is given some number of “ticks” that limit the code’s execution. If the code finishes executing, then the engine returns the code’s return value and the number of unused ticks. If the code does not finish, then the engine returns a new engine that may be used to continue executing the code. An implementation with continuations represents the code inside an engine with a continuation. To limit the code’s execution, the implementation needs preemption via a continuation-based interrupt facility: when an interrupt occurs, the handler has access to the interrupted code in the form of a continuation. Combined with Wand’s technique, engines can implement a time-shared multiprocessing system.

A user-level threads implementation in ML [Cooper & Morrisett 90] demonstrates that the continuation-based multiprocessing and preemption techniques described above can implement a traditional threads interface. The ML user-level threads interface provides facilities similar to Mach’s C-Threads [Cooper & Draves 88]: a fork operation to create a new thread, an exit operation to allow a thread to terminate itself, and mutex and condition variables for synchronization among threads.

2.3.4 Functional Continuations

Functional continuations and control delimiters supply an alternative formulation for first-class continuations that dynamically restricts continuation-based control transfer [Felleisen *et al.* 88]. This scoping mechanism provides the isolation that one desires in an operating system. In fact, the OS6 operating system and its implementation language BCPL used a similar dynamic scoping for control transfer [Stoy & Strachey 72].

A functional continuation differs from the abortive continuations produced by `call/cc` in that the invocation of a functional continuation does not discard the current evaluation context. Instead, the functional continuation can return a value to its caller. However, the `control` operation that produces functional continuations does discard its current evaluation context, unlike `call/cc`, so the functional continuation must be used to make `control` “return.”

For example,

```
(define example2
   (lambda (x)
      (control (lambda (k)
         (+ 1 (* 2 (k x))))))

(+ 5 (example2 3)) ⇒ 17
```

because `k` gets bound to the functional continuation `(lambda (x) (+ 5 x))`, then `(k 3)` evaluates to `8`, the body of control evaluates to `17`, and this is the final result because `control` discards its evaluation context.

The control delimiter `run` evaluates its argument in an independent control context. An application of `control` inside `run` only discards its evaluation context back to the boundary created by `run`, and the created functional continuation only reaches back to that boundary. This means that there is no way for code executing inside an application of `run` to escape, or transfer outside of the `run` application.

The OS6 operating system, an early experimental single-user, single-address space, single-threaded system written in BCPL, contains a very similar `run` primitive for executing programs. Programs in OS6 are just BCPL thunks (functions without arguments). The `run` primitive constrains the execution of a subprogram so that no matter what happens, the subprogram returns properly to its parent's `run` invocation. This isolates the parent from the vagaries of the child's execution.

2.4 Conclusions

Given that continuations were invented as a technique for expressing control transfer in a very general setting, it should not be surprising that continuations can generalize a number of specific operating system control transfer optimizations.

From an operating systems perspective, my framework for using continuations most strongly resembles a combination of the stack-less threads of Taos with the finish-up functions of V, although neither of those systems uses continuation recognition. By allowing a thread to block with either an explicit continuation function or an implicit continuation consisting of saved stack frames and register state, continuations combine the software engineering advantages of a multi-threaded process-based kernel organization with the performance advantages of an event-driven interrupt-based organization.

From a programming language perspective, this technique most strongly resembles functional command continuations. In those terms, the user-kernel boundary forms a natural control delimiter. Blocking with a continuation function discards the current execution context in the same way that the `control` primitive discards the current evaluation context back to the last control delimiter. The major inconvenience with a traditional systems programming language such as C is that the programmer must explicitly supply functional continuations.

Chapter 3

Control Transfer Management

The efficient, portable use of continuations in an operating system kernel requires special attention to the low-level management of control transfer. By this, I mean the code that implements control transfer and manages the supporting data structures, principally threads, register context save areas, and kernel stacks. The control transfer paths that are of interest here include transitions between threads inside the kernel and transitions between threads running at user-level and kernel code. The chapter concludes with a description of an interface for control transfer management that provides additional flexibility and opportunities for optimization in support of continuations. This development proceeds as follows:

- I start with a review of current practice. Most systems use a kernel stack per thread and have a single context-switch primitive to switch between threads.

- Next, I examine in detail the control transfer needs of continuations. Continuations can be implemented using fixed kernel stacks and a context-switch primitive, but some important optimizations require more flexible control transfer support.

- From these and other considerations, I synthesize requirements for an interface for control transfer management.

- Finally, I present an interface that meets these requirements. Like the pmap interface in Mach [Rashid *et al.* 87; Tevanian 87], the design of this interface derives from the requirements of the machine-independent code that uses it; the interface does not attempt to abstract or generalize hardware support for control transfer.

I also include some examples of the interface's use and a discussion of its portability. I defer to Appendix B a complete description of the details of the interface.

3.1 Current Practice

Most current operating system kernels allocate a kernel stack for each kernel-supported thread. Figure 3-1 illustrates the situation. (Section 2.1.2 discusses alternatives to this design.)

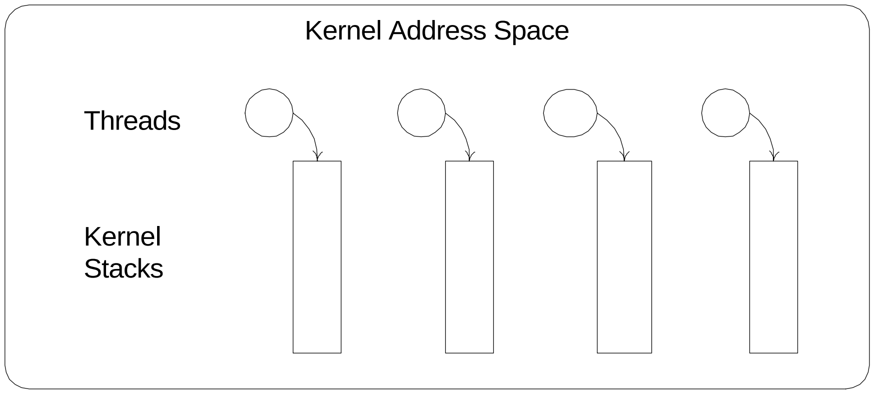

Figure 3-1: One Kernel Stack Per Thread

This arrangement requires a context-switch for the kernel to switch from running one thread to running another. That is, the kernel must save the register context of the currently running thread and restore the saved register context of another thread. The register context can be pushed on the kernel stack, with a pointer to it kept in the thread structure, or it can be saved directly in the thread structure. The kernel must also manipulate the MMU to switch to a new address space if the two threads belong to different processes.

Because each thread has its own kernel stack and register context save area, it is in principle always possible to suspend the currently executing thread and switch to another thread. In practice, this level of concurrency may be undesirable. However, if a thread running inside the kernel wishes to block because memory is temporarily unavailable, or because an IO operation (perhaps due to a page fault) has not completed, or because a lock that it wishes to acquire is held by another thread, the thread can block and its state will be saved and restored properly.

The details of the context-switch operation, which saves and restores register context and changes the MMU context, depend on the hardware architecture. The following subsections review how several recent operating systems package the context-switch functionality internally.[1] All of these systems attempt to hide the machine-dependent details while exposing the important functionality, because they have been ported to multiple CPU architectures. Broadly speaking, they offer similar levels of functionality—virtual memory with multiple address spaces, file systems, networking, *etc*.

3.1.1 BSD Unix

BSD4.3 [Leffler *et al.* 89] is a version of Unix [Ritchie & Thompson 78] developed at the University of California at Berkeley and used in both research and production environments. For its context-switch primitive, BSD4.3 uses `swtch()`. This operation selects a new process and context-switches to it. One drawback of this specification is that the code to select the next process to run is duplicated in each machine-dependent implementation of `swtch()`.

Because BSD4.3 does not run on multiprocessors and its scheduling code disables interrupts, `swtch()`does not address concurrency issues.

A more recent version, BSD4.4, uses essentially the same interface with the name changed to `cpu_swtch()`. The `swtch()` function, which calls `cpu_swtch()`, handles in addition the machine-independent calculation of CPU usage.

3.1.2 Windows NT

Windows NT [Custer 93] is a commercial operating system from Microsoft. It uses `KiSwapContext(new-thread, ready-flag)` to implement context-switch. Two things distinguish `KiSwapContext` from BSD's `swtch`. First, KiSwapContext has less knowledge of the machine-independent scheduling data structures. Second, the operation provides for multiprocessor operation.

`KiSwapContext` receives the next thread to run as an argument, so it doesn't need any knowledge of the kernel's scheduling policy. Before doing anything else, `KiSwapContext` puts the current thread back in the ready queues if `ready-flag` is `TRUE`, but it uses an upcall to the machine-independent `KiReadyThread` to do this.

A single lock controls access to the scheduling data structures in Windows NT. `KiSwapContext` drops the lock during its execution, after it has switched to the new thread's stack but before the new thread's register context is restored. This prevents another processor from picking up the

[1]This discussion relies on my examination of the source code for each of these systems.

previous thread and starting to run it while its stack is still in use, but it has the drawback that `KiSwapContext` "knows" the locking strategy for the machine-independent data structures.

3.1.3 Sprite

Sprite [Ousterhout *et al.* 88] is a research operating system from the University of California at Berkeley. Sprite implements context-switch with `Mach_ContextSwitch(old-process, new-process)`. Unlike `KiSwapContext`, Sprite's global scheduling lock is held throughout a call to `Mach_ContextSwitch`. This means that the machine-dependent implementation of `Mach_ContextSwitch` has no knowledge of the machine-independent data structures or locking strategies. The disadvantage of this approach is that the global scheduling lock is held throughout a potentially lengthy operation.

3.1.4 Mach 2.5

Mach 2.5 [Accetta *et al.* 86], an early version of the Mach operating system from Carnegie Mellon University, uses `switch_thread_context(old-thread, new-thread)`. This interface takes a different approach to handling multiprocessor concurrency. After switching to the new thread's stack, but before restoring its register context, `switch_thread_context` makes an upcall to the machine-independent `thread_continue(old-thread)`. This function adjusts the scheduling state of the old thread, putting it back in the ready queues if necessary. This technique avoids holding any locks across the call to `switch_thread_context`.

Unlike BSD's `swtch`, NT's `KiSwapContext`, and Sprite's `Mach_ContextSwitch`, `switch_thread_context` does not change address spaces. Machine-independent scheduling code uses separate calls to Mach's pmap module (responsible for MMU management) to change address spaces before calling `switch_thread_context`.

3.2 Continuations

In this section I examine the control transfer needs of continuations. First I arrive at an example of continuation-using code, and then I consider possible implementations. Continuations could be implemented using a kernel stack per thread and a context-switch primitive, but some important optimizations—stack discarding and stack handoff—require more flexible control transfer support.

3.2.1 Simple Example

Continuations give the kernel programmer a choice of thread state representations. When a thread blocks voluntarily, the programmer can choose the natural machine-dependent representation, saving the thread's register context and stack frames, or the programmer can choose a compact machine-independent representation, saving a continuation function and the important state variables. When a blocked thread has a continuation function, it resumes by executing the function.

To make this work, the programmer must have blocking primitives that provide this choice. Examples of such primitives include `sleep`, which blocks the calling thread for a specified period of time, `wait`, which blocks the calling thread until another thread calls `wakeup` on it, and `yield`, which blocks the calling thread and moves it to the rear of the ready queue. The support for continuations could take the form of alternative primitives, such as `sleep_with_continuation` and `wait_with_continuation`, but it is simpler to add a continuation argument to `sleep` and `wait` and `yield` and other such primitives and specify that a null value for the continuation means that the thread blocks in the normal manner, saving register context and stack frames.

Another requirement for making this work is that the programmer must have a way to return or exit from a continuation function. For example, if the implementation of a system call blocks with a continuation function, then the continuation function needs a way to reach the system call exit path, the machine-dependent instructions that restore the user's saved register context and return the system call's status code. One obvious possibility is that the control transfer mechanism that calls the continuation function could set up the stack frame so that a return from the continuation function branched to the system call exit path, with the return value from the continuation becoming the system call's status code.

Unfortunately, most kernels have at least two exit paths, one for system calls and one for exceptions and interrupts. The entry and exit paths for asynchronous entrances into the kernel, such as page faults, device interrupts, and arithmetic exceptions, save and restore the machine's entire user register context. In contrast, the system call entry and exit paths typically save and restore fewer registers, and the system call exit path returns a status code.

The best way to handle the possibility of multiple exit paths from a continuation function is to provide multiple exit primitives, such as `return_from_system_call` and `return_from_-exception`, that the continuation function can call. By convention then, continuation functions never return directly. (They don't have a valid return address!) Instead, they make calls to other functions that also never return. Eventually, an exit primitive transfers control back out of the kernel.

Putting this all together, Figure 3-2 shows a system call implementation that uses continuations. The system call `example` takes two arguments. It uses the first argument and then calls `wait`, supplying a continuation function, after saving away the second argument. The call to `wait` does not return. Instead, when the thread wakes up the continuation function, `example_cont`, executes. The continuation function retrieves and uses the second argument and then returns a status code to the user with `return_from_system_call`.

```
example(arg1, arg2) {
    use arg1;
    current_thread()->save_arg = arg2;
    wait(example_cont);
    // NOTREACHED
}
example_cont () {
    arg2 = current_thread()->save_arg;
    use arg2;
    return_from_system_call(SUCCESS);
    // NOTREACHED
}
```

Figure 3-2: System Call with Continuation

3.2.2 Implementation

A traditional control transfer framework, with a dedicated kernel stack per thread and a context-switch primitive, would suffice to implement continuations. Although in practice this would not be desirable, it is nevertheless useful to consider the possibility to establish a base for subsequent optimizations.

Looking again at Figure 3-2, there are three operations to consider:

1. Context-switching *away* from a thread blocking with a continuation.

2. Context-switching *to* a thread blocked with a continuation.

3. Returning from a continuation back to user-level.

A control transfer interface like Windows NT's `KiSwapContext` and the other interfaces discussed in Section 3.1 could be tweaked to implement these operations. Figure 3-3 shows how this would work.

Context-switching away from a thread blocking with a continuation is easy, because the context-switch does not have to save the blocking thread's register state. Instead, the context-switch just saves the continuation function in the thread structure.

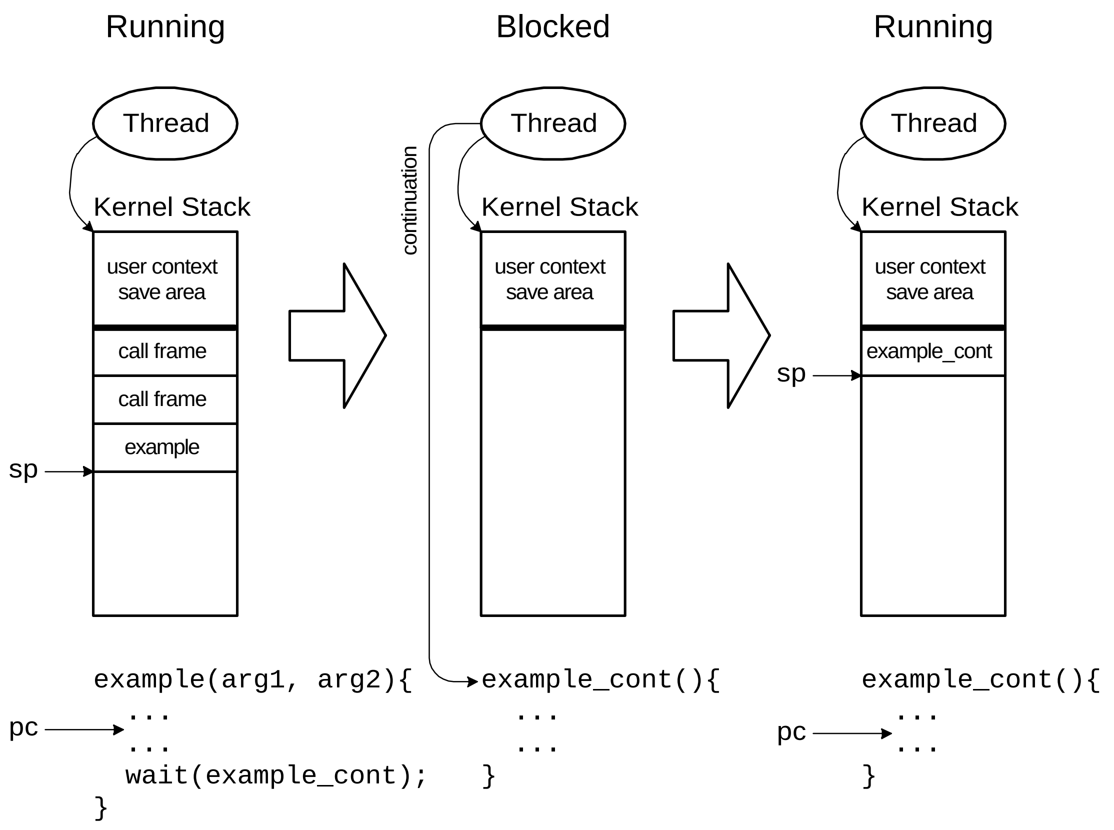

Figure 3-3: System Call Implementation with Continuation

Context-switching to a thread blocked with a continuation is also easy, because the context-switch does not have to restore saved register state. If the saved continuation function is not null, then the context-switch knows the thread blocked with a continuation. Instead of restoring register state, it builds a stack frame for the continuation function at the base of the thread’s kernel stack, and just branches to the continuation function. The return address in the stack frame can be left null or set to a debugging function, because the continuation function should never return.

Finally, returning from a continuation to user-level is easy. The return primitives can just branch to the normal kernel exit paths. If user register state is saved at the base of the kernel stack, then the return primitive will need to reset the stack pointer so that the exit path finds the saved register state.

One subtle complication occurs in the system call entry and exit paths. Current calling conventions divide the register set into caller-saved and callee-saved registers. The system call entry and exit paths do not save or restore the caller-saved registers; if they should be saved they are the responsibility of the user. (For complete safety, the exit path should zero the caller-saved registers, to prevent sensitive values from leaking out of the kernel.) The entry and exit paths may also avoid saving and restoring the callee-saved registers, because the compiler will naturally save and restore them as the system call executes. This means part of the user's register context is sprinkled through stack frames on the kernel stack. Unfortunately, when a continuation function executes this state will be lost.

One simple solution to this problem defines a variation of the system call entry and exit paths, for continuation-using system calls, that explicitly saves and restores callee-saved registers.

3.2.3 Optimizations

Although it is possible to implement continuations within a control transfer framework that dedicates a kernel stack to each thread, in practice this would not be satisfactory. Implemented properly, continuations can achieve the performance benefits of having a single kernel stack per processor. This requires support for stack discarding and stack handoff optimizations.

Stack discarding occurs when a thread blocks with a continuation function, as in Figure 3-3. While the thread is blocked in this manner, its kernel stack is not needed and can be deallocated. Of course, before the thread can run again it needs a new kernel stack. Stack discarding represents one example of a space-time tradeoff—the kernel recovers the memory consumed by idle kernel stacks, at the expense of frequent stack allocation and deallocation operations.

Stack handoff improves upon stack discarding by eliminating the stack allocation and deallocation operations, in most cases. Stack handoff occurs when a thread blocks with a continuation function and the next thread to run is blocked with a continuation. In this case, the blocking thread has a stack that it no longer needs and the resuming thread is in need of a stack. Stack handoff transfers the blocking thread's kernel stack directly to the resuming thread.

3.3 Interface Requirements

In this section, I discuss requirements for an interface for control transfer management. These requirements stem from the previous two sections, in which I reviewed current practice and examined the control transfer needs of continuations. This discussion leads directly to the interface design presented in Section 3.4.

Portability and efficiency are important overall requirements. The control transfer interface must be implementable on a very wide variety of processor hardware, including multiprocessors, and implementations for current and foreseeable RISC processors must be very efficient. My goal for portability is that the control transfer interface must be at least as portable as current practice, a simple context-switch primitive.

More specifically, the requirements fall into the following categories:

- Stack Management. The control transfer interface must support stack discarding and stack handoff. Stack allocation and deadlock prevention considerations complicate this requirement.

- Context-Switch. The control transfer interface must support transfers between threads. Multiprocessor considerations complicate this requirement. The control transfer operations must support multiple multi-threaded address spaces.

- Kernel Entry and Exit. The control transfer interface must support multiple kernel entry and exit paths.

The following subsections discuss these requirements in more detail, and Section 3.3.5 summarizes with a final detailed list of requirements.

3.3.1 Portability

I've implicitly assumed in this chapter that the best way to structure control transfer management for portability divides the implementation into two pieces, machine-independent code and machine-dependent code, with a control transfer interface between them. Done properly, this approach has the advantage that porting to a new machine requires no changes to machine-independent code, but only another implementation of the control transfer interface. In addition, the machine-independent algorithms and data structures can change without coordinated modifications to the many machine-dependent implementations. These software-engineering benefits are tremendously important when different groups control different ports of the software.

However, another approach to portability has been very successful and must also be considered. For example, GNU Emacs, a widely-used text editor and programming environment, does *not* define an interface to the operating system that ports of GNU Emacs must implement.[2] Instead Emacs uses many preprocessor symbols to control conditional compilation in a unified source tree. Ideally, these preprocessor symbols each control Emacs' adaptation to one "feature" of the underlying system. Some examples include `BROKEN_TIOCGETC`, `DONT_DEFINE_SIGNAL`, and `HAVE_GETPAGESIZE`. In this environment, porting becomes a matter of creating a configuration

[2] This discussion relies on my examination of the source for GNU Emacs version 18.59 as maintained at CMU SCS.

header file that defines the right subset of these preprocessor symbols. Using this approach, the Emacs 18.59 source tree supports 46 operating systems and 74 hardware platforms.

The Emacs approach has some advantages. Porting can be easy if the existing set of preprocessor symbols suffices to define the new environment. A configuration tool can examine the new environment and automatically produce a configuration header file. Furthermore, by defining new preprocessor symbols and modifying the Emacs source code, a port can customize Emacs arbitrarily.

The Emacs approach also has some very significant disadvantages. First, the software slowly grows in complexity as it accumulates ports. Emacs 18.59 uses more than 450 preprocessor symbols, with roughly 1500 preprocessor #if statements. This means that many preprocessor symbols have very few uses. The relatively clean “feature” preprocessor symbols unfortunately constitute a minority. Furthermore, the density of preprocessor #if statements is high—roughly one in every twenty lines of code. This is more than twice the density of #if statements in the machine-independent Mach 3.0 source. (This comparison is generous to Emacs, because almost all of the #if statements in the Mach 3.0 source control features unrelated to porting.) Finally, because the Emacs approach does not cleanly separate machine-dependent and machine-independent code, it requires continual effort on the part of Emacs developers to incorporate and maintain modifications made for new ports.

Given these considerations, I believe that the best approach to portability starts with a clean interface that separates machine-independent and machine-dependent code.[3] A few judiciously-chosen “feature” preprocessor symbols can make an interface more flexible, but relying on conditional compilation for portability ultimately results in too much complexity and maintenance difficulties.

3.3.2 Stack Management

Stack allocation, stack discarding, and stack handoff determine the most interesting requirements for a control transfer interface that can support continuations effectively. Several issues surround kernel stack allocation, including special requirements for kernel stack memory and what happens when kernel memory for stacks is not immediately available. Stack handoff is an obvious optimization to avoid stack discarding and allocation operations. Stack handoff becomes more

[3]One could argue that the Emacs implementors started with exactly this approach, using the Unix system call interface as their basis for portability. If so, variations among versions of Unix and ports to non-Unix platforms have since obscured this design.

interesting when one considers the possibility of exposing this functionality directly, with its own semantics.

In fact, stack discarding and stack handoff could be implemented under the covers of a context-switch primitive like NT's `KiSwapContext`. `KiSwapContext` would have to free the current kernel stack if the thread was blocking with a continuation, and allocate a new kernel stack if the new thread lacked one because it was blocked with a continuation. This means that `KiSwapContext` would have to be cognizant of kernel stack allocation. `KiSwapContext` would have to report failure if it couldn't allocate a needed stack. It could transparently optimize discarding one stack and allocating another by reusing the current kernel stack.

The following examination of the issues surrounding stack allocation and stack handoff demonstrates that this possibility is unattractive; it pushes too much complexity into the multiple machine-dependent implementations of context-switch and it misses an opportunity to expose useful functionality with stack handoff.

3.3.2.1 Controlling Stack Allocation

The control transfer interface should give machine-dependent code the *option* of full stack management, with control of stack allocation and deallocation. The interface should not *require* that machine-dependent code assume this burden. In most cases, a standard machine-independent stack-management implementation is quite satisfactory. The standard kernel stack management uses normal kernel virtual memory primitives to allocate and free stacks.

Some CPU architectures need special stack management. This happens when kernel stack memory requires special attributes or address ranges. The usual incentive in these cases is better memory system performance. Examples include the MIPS architecture, where special kernel stack allocation can reduce TLB misses, and multiprocessor 88K systems, where special kernel stack allocation can reduce memory cache overhead.

Reducing TLB Overhead

The MIPS architecture [Kane 88] uses a software-managed TLB. Some kernel stack memory references occur when TLB misses are not permitted; this happens during TLB miss handling and when changing the current address space. This constraint has two solutions:

- Kernel stacks can be allocated in normal, mapped kernel virtual memory (the kseg2 region of the address space). The current kernel stack must be wired in the TLB, using one or more of the eight reserved TLB entries. Context-switches must unwire the old stack and wire the new stack.

- Kernel stacks can be allocated in non-mapped physical memory (the kseg0 region). With this alternative, kernel stack accesses do not require TLB entries. This is more efficient, because more TLB entries are available for other uses and context-switching is faster, but in most implementations it restricts kernel stack size to one page (4K bytes) because the kernel manages physical memory with page-level granularity.

Many operating systems for the MIPS architecture (*e.g.*, Ultrix, Windows NT, Sprite, and Mach 2.5) allocate kernel stacks in mapped kseg2 memory because they have multi-page stacks. However, some systems can take advantage of unmapped kseg0 memory. For example, Mach 3.0 can operate with 4K kernel stacks, because it is a microkernel system that runs less code in the kernel address space. To realize the performance benefit of allocating stacks in unmapped kseg0 memory, the kernel needs to avoid using the normal kernel virtual memory primitives for stack allocation.

The Alpha architecture [Sites 92] also uses a software-managed TLB. As in the MIPS case, the ability to bypass the normal stack allocation mechanism yields improved performance.

Reducing Cache Overhead

A multiprocessor system based on the 88K architecture [Motorola 90a] provides another example. The MC88200 memory management unit [Motorola 90b] allows memory to be non-coherent, or private to a processor, via a bit in TLB entries. Because accesses to such memory bypass the normal cache-coherency protocols, cache misses are handled two cycles faster. In addition, the cache-coherency protocol potentially stalls other processors for one cycle while their cache tags are accessed. With this architecture, it is advantageous to allocate kernel stacks in non-coherent memory and use explicit cache operations to synchronize stack memory only when necessary. Two things make this possible:

- Kernel stacks are used by only one processor at a time; there is no concurrent sharing of kernel stack memory. This is true of most operating system implementations, not just Mach 3.0.

- Kernel stacks are normally a per-processor resource, with one stack allocated for the use of each processor. This is due to the extreme prevalence of stack-handoff operations when continuations are used. (See Section 5.2 for performance numbers characterizing this effect in

Mach 3.0.) When a relatively rare cross-processor stack migration occurs, explicit synchronization can be performed.

Therefore, a multiprocessor 88K implementation like the Luna 88K will also wish to bypass a machine-independent stack allocation mechanism.

Summary

Implementations of the control transfer interface for some CPU architectures will want to assert control over stack allocation, but these will be in the minority. Most implementations would be happy to use standard machine-independent stack management code. How can the control transfer interface provide this flexibility for some machine-dependent code without burdening all machine-dependent code?

One solution defines two versions of the control transfer interface. The complete interface assumes machine-dependent control of stack memory allocation and deallocation and a subset interface assumes a machine-independent implementation. Client code uses the complete interface, without worrying about the implementation of stack allocation. Those architectures that don’t actually need this control implement only the subset, and standard machine-independent code fills in the rest.

3.3.2.2 Non-Blocking Stack Allocation

The control transfer interface should provide for non-blocking stack allocation. This subsection presents the need for non-blocking allocation and outlines a recovery mechanism, based on a stack-alloc thread, that the kernel can use when the non-blocking allocation fails to allocate a stack.

Stack handoff optimizes away most stack allocation operations, but occasionally the context-switch path must allocate a new kernel stack. This occurs during a switch from a thread that must hold onto its stack (because the thread is *not* blocking with a continuation) to a thread that does not have a stack (because it *is* blocked with a continuation). In this case, a stack must be allocated for the new thread’s use. However, the context-switch path can’t use a normal stack allocation primitive, because it might make use of the VM subsystem to allocate memory:

- The VM subsystem might block if there is no memory available. This would lead to a call back into the context-switch path, and infinite recursion would result.

- The stack allocation probably occurs at a point in the context-switch path that holds scheduling or thread locks, because it is modifying the state of a thread. From a locking hierarchy and performance point of view, calling out to virtual memory code that will be taking its own locks is unwise.

For these reasons, the context-switch code needs a way to allocate stacks that won't block and won't take any virtual memory locks. However, this means that the stack allocation call might fail. Therefore the kernel also needs a mechanism to cope with the temporary inability to allocate a stack.

The solution to this problem comes in two parts. The first part, a requirement for the control transfer interface, is a non-blocking alternative for stack allocation that can report failure. The second part, which comes into play when stack allocation fails, is a kernel thread dedicated to making possibly-blocking stack allocation calls.

One easy way to implement a non-blocking stack allocation primitive uses a small cache of unused kernel stacks. Then stack allocation only fails when the cache is empty. (Of course, a "correct" implementation of non-blocking stack allocation could *always* fail. This would work, but it would produce suboptimal performance.)

Because of locking considerations it is also desirable that stack deallocation avoid calling into the VM subsystem or other foreign modules and taking random locks. With a stack-cache implementation of stack allocation, this is easy: deallocating a stack just adds the stack to the cache.

This analysis exposes another requirement for the control transfer interface. For the reasons outlined above, as well as the performance benefit of avoiding VM allocation and deallocation, implementations of the control transfer interface will likely keep a cache of unused stacks. This means that the control transfer interface should expose some way of hastening the recovery of this unused memory when the kernel virtual memory system runs out of physical memory. With this addition, caching page-sized (or larger) kernel stacks will not cause memory availability problems.

As a consequence of non-blocking stack allocation, it must be possible to defer stack allocation to a context that can make blocking stack allocation calls. Strictly speaking, this mechanism is not part of the control transfer interface, but it is so closely related that it should be discussed here.

An obvious method for deferring stack allocation uses a distinguished stack-allocation thread, whose sole purpose is to perform blocking stack allocation calls. When the scheduling code's non-blocking stack allocation call fails, it can put the stack-less thread on a queue for the stack allocation thread and pick another thread, which hopefully won't need a stack to run.

This mechanism is not optimal, because while a stack-less thread is sitting in the stack-allocation thread's work queue more stacks may become available (because some other threads block and give up stacks). Meanwhile, if the stack allocation thread is blocked in the VM subsystem, the stack-less thread will be left unrunnable for longer than necessary. However, this situation should be quite rare.

3.3.2.3 Avoiding Deadlock

The description of the stack-allocation thread in the previous subsection left unanswered an obvious question: who allocates the stack-allocation thread's stack? The control transfer interface must address this problem with some mechanism that will ensure that certain "stack-privileged" threads always have a stack available to them.

Stack-Privileged Threads

In fact, there are several threads for which a stack allocation attempt should always succeed. In addition to the stack-allocation thread, the idle threads (if the scheduler uses idle threads) and other internal threads in the virtual memory system's pageout path must be stack-privileged.

The stack-allocation thread is the most obvious example—it is very likely that when the scheduling code selects the stack-allocation thread to run, a stack handoff to the stack-allocation thread will not be possible. (Because the stack-allocation thread is made runnable only when stacks need to be allocated, and more stacks are only needed when stack handoff is not possible because the current thread is blocking and keeping its stack.) At this point, if there is no stack for the stack-allocation thread, then it will be added to its own work queue of threads waiting for a stack and deadlock with itself.

Idle threads, assuming the scheduler uses them, are another example. (Typically there is one idle thread per processor in a multi-processor system.) When the scheduling code fails to allocate a stack for the thread it wishes to run next, it must back off and select another runnable thread. If it selects the idle thread, this means that there are no other runnable threads. The scheduler can't select another thread at this point, so a context-switch to the idle thread must succeed. This implies that a stack allocation attempt for the idle thread must succeed.

All threads involved in creating more free pages when the system is running out of available memory also fall into this category. In the case of Mach 3.0, these are the pageout daemon thread, the threads in the default pager that service "internal" memory objects [Young 89, p. 34, p. 64], and the device-reply thread, which is used when the default pager writes out dirty pages. These are the same threads that are "VM-privileged" and can dip into the virtual memory system's reserved memory pool [Young 89, p. 80; Draves 91, p. 209]. If any of these threads were denied a stack and consequently placed on the stack-allocation thread's work queue, the system could deadlock because the stack-allocation thread might block waiting for free pages and a thread necessary to produce those free pages would be waiting for the stack-allocation thread. (The stack-allocation thread itself can not be VM-privileged, or a burst of threads needing stacks could exhaust the reserved memory pool. Only threads necessary to the pageout of internal memory objects by the default pager can be VM-privileged.)

Implementing Stack-Privilege

The control transfer interface must guarantee that stack-privileged threads always have a stack available to them. One good way to accomplish this reserves a stack for each stack-privileged thread.

Actually, disallowing stack discarding for stack-privileged threads would be a very simple solution. It would not impose any other constraints on the control transfer interface. However, context-switches to and from idle threads and possibly other stack-privileged threads are relatively frequent. For performance reasons, it is desirable that the control transfer interface support stack-privileged threads with a mechanism that allows them to participate in stack handoff as much as possible, while still preserving correctness.

A better solution reserves a stack for each stack-privileged thread, but allows the stack-privileged threads to participate in stack handoff and use non-reserved stacks in normal operation. This is easy to manage; a field in the thread structure holds a pointer to the reserved stack, or null if the thread is not stack-privileged.[4] The scheduling code has the responsibility of not using stack handoff when the current thread is stack-privileged *and* using its reserved stack. Otherwise the stack-privileged thread would lose access to its reserved stack. However, stack-privileged threads can participate in stack handoff using other stacks.

This solution creates two requirements for the control transfer interface. First, if the non-blocking stack allocation does not have any cached stacks available, then it should use the reserved stack if the thread is stack-privileged. Second, the stack discarding operation should not actually free a reserved stack from a stack-privileged thread. (In this case, the stack discarding operation should perform any other cleanup necessary to break the normal links between the stack-privileged thread and the reserved stack that it is using.)

[4]An equivalent but less convenient formulation would set aside a reserved pool of kernel stacks, without specifically assigning each reserved stack to a VM-privileged thread.

This stack-privilege algorithm could be improved to permit a stack-privileged thread to handoff its reserved stack to another stack-privileged thread. Consider two stack-privileged threads TA and TB and their respective reserved stacks SA and SB. Before the stack handoff, TA is running on SA and TB is blocked with a continuation, so SB is not in use. To preserve the invariant that both threads have a reserved stack available to them, the stack handoff need only swap the reserved stacks. After the stack handoff, TA is blocked with a continuation and is not using its reserved stack, which is now SB, and TB is running on its reserved stack, which is now SA. However, this situation arises infrequently enough that the optimization does not seem to be worth the increased implementation complexity.

Another minor improvement would permit a stack-privileged thread to handoff its reserved stack to another thread, if a replacement reserve stack could be allocated out of the cache of unused stacks. Again, this is probably not worth implementing.

3.3.2.4 Exposing Stack Handoff

Stack handoff is most useful if it does not happen under the covers of a context-switch. So far, I’ve presented stack handoff as an optimization in context-switch for stack discarding and subsequent allocation. This is an important optimization; without it continuations would not be very useful. However, stack handoff can be more than just an optimization.

A stack-handoff primitive that changes address spaces and the “current thread” *without* calling the new thread’s continuation effectively lets a computation cross address space boundaries. For example,

```
// thread A is running
thread B = get waiting thread;
stack-handoff(thread A, thread B);
// now thread B is running
```

The code now has the option of calling thread B’s continuation or going off to do something else entirely. Section 4.2.5 and Section 4.2.6 examine the application of this form of stack handoff to achieve continuation recognition, another valuable optimization. In brief, a single optimized function can replace two more general functions, one in the blocking thread and one in the resuming thread.

3.3.2.5 Avoiding Stack Overflow

Exposing the stack handoff operation creates another requirement for the control transfer interface, to avoid stack overflow. This section discusses how stack overflow could occur and how it can be prevented.

A sufficiently long sequence of stack handoff operations, if uninterrupted by a return to user mode, will result in stack overflow. Actually, the real problem isn't stack handoff; it is the continuation calls associated with stack handoff. Consider what could happen: thread A blocks with a continuation, the scheduler selects thread B to run and performs a stack handoff to B, and then calls thread B's continuation. Thread B runs inside the kernel and blocks again with a continuation, resulting in a stack handoff to thread C and a call to thread C's continuation. Thread C runs inside the kernel and blocks again with a continuation, resulting in a stack handoff to thread D and a call to its continuation, *etc*. Because the continuations do not return, each thread's continuation function uses up more of the stack that is passed from thread to thread. No matter how big a stack the kernel uses, a sufficiently long sequence of continuation calls will overflow the stack.

To prevent this problem, the control transfer interface should contain an operation to call a continuation function. This primitive should call its continuation argument, *after* resetting the current stack pointer to the base of the stack. In other words, it should create the same stack frame for the continuation function as the stack-allocation operation would.

3.3.3 Context Switch

The control transfer interface should include some variant of the traditional context-switch primitive, for use when threads block without a continuation and stack handoff is not possible. Two issues complicate this requirement. First, multiprocessor machines are vulnerable to a race condition that can result in a thread running simultaneously on two processors. Second, some provision for changing address spaces must be included, either as a separate primitive or as part of context-switch.

3.3.3.1 Multiprocessors

On multiprocessors, there is a window of vulnerability between when a thread decides to context-switch and when the new thread is running. This creates a requirement for the control transfer interface, to allow machine-independent code to context-switch safely on multiprocessors without undue dependencies between machine-independent and machine-dependent code.

The obvious context-switch code for a uniprocessor, when moved to a multiprocessor, opens up a window in which two processors can be using the same stack. This occurs when a wakeup happens on a thread which is just in the process of blocking, and another processor starts running the thread before it has finished blocking on the first processor.

For example, a naive implementation might look like:

```
swtch() {
   if (!setjmp(self)) {
      add self to run queues;
   danger-point:
      get new thread from run queues
      longjmp(new thread);
      /*NOTREACHED*/
   }
}
```

where `setjmp` saves registers and returns zero, and `longjmp` restores registers that were saved by a previous `setjmp`, making the `setjmp` return again but with a non-zero value.

On a multiprocessor, this code is not correct. At the label `danger-point`, another processor might pick the current thread out of the run queues and `longjmp` to it. This would result in the thread executing on two processors at once, with the likelihood that the processors would clash in their use of the stack.

There are many possible solutions to this problem, all of which defer some change in scheduling state until after the switch to the next stack. For example:

- Sprite's `Mach_ContextSwitch` assumes that the current thread will take a lock on the scheduling data structures, which is released by the next thread after the context-switch. This has the advantage that `Mach_ContextSwitch` knows nothing about the machine-independent data structures and algorithms.

- Windows NT's `KiSwapContext` addresses this problem by holding a global lock on the scheduling data structures that is released after the switch to the next stack but before restoring the next thread's register context.[5]

- Mach 2.5's `switch_thread_context` uses an upcall to the machine-independent scheduling code, after switching to the next stack but before restoring the next thread's register context, to put the old thread on the run queues. This has the advantage that scheduling locks are not held through the context-switch path.

Another possibility effectively combines the Sprite and Mach 2.5 techniques. The context-switch operation can make available to the new thread a pointer to the previously running thread. At this

[5]In fact, to take NT's global scheduling lock one must first prevent interrupts and then acquire a spin-lock. The spin-lock is released after switching to the next stack but before restoring the next thread's register context. The next thread then "lowers IRQL" to allow interrupts again. This is all hidden inside `KiSwapContext`; it only makes a difference when the next thread did not block with `KiSwapContext`.

point the new thread can safely put the previous thread back in the run queues, or otherwise finalize the previous thread's scheduling state. This insulates the context-switch code from the machine-independent scheduling implementation; the scheduling code may or may not hold a lock across the context-switch code, depending on its needs.

3.3.3.2 Multiple Address Spaces

In addition to context-switching between two threads, the control transfer interface must also support switching between address spaces. The context-switch operation and the stack handoff operation should both automatically change the current address space when the two threads involved belong to different address spaces.

Mach 2.5 takes a different approach, in which address space switches use a primitive in a different interface. This approach cleanly separates the interfaces to two machine-dependent modules, control transfer and MMU management.

However, putting all machine-dependent components of a context switch or stack handoff into one operation allows them to be optimized together with a minimum of procedure call overhead. This is especially important for stack handoff, because in practice it occurs much more frequently than context-switch. This doesn't preclude separate operations to manipulate the MMU context.

3.3.4 Kernel Entry and Exit

The control transfer interface must provide for entry to the kernel from user processes and exit back to a user process. Of course, all operating system kernels support entry and exit. The interesting aspect of these features concerns their interaction with continuations and kernel stack management.

3.3.4.1 Multiple Exit Paths

As I discussed in Section 3.2.1, the programmer must have a way to return or exit from a continuation function. For example, if the implementation of a system call blocks with a continuation function, then the continuation function needs a way to reach the system call exit path, the machine-dependent instructions that restore the user's saved register context and return the system call's status code.

For good reason, most kernels have at least two entry/exit paths, one for synchronous entrances, such as system calls, and one for asynchronous or unexpected entrances, such exceptions and interrupts. In the case of synchronous entrances, the kernel typically saves and restores a subset of the machine's user register context. A system call need only save callee-saved registers, and even some of this responsibility may be delegated to the system call stub linked with the user program or to the compiler-generated function prologues in the kernel. In addition the system call exit path returns a status code. In contrast, the asynchronous paths typically save and restore the entire user register context, which may be significantly more expensive.

System call, exception, and interrupt handling may all result in control transfers that benefit from continuations, so the control transfer interface must support exits from a continuation function in each of these contexts.

3.3.4.2 Saving User Context

The stack management functionality interacts with the saving and restoring of user register context. This does not affect the control transfer interface *per se*, but it does create a requirement for implementations of the control transfer interface: user register context can not be tightly tied to the kernel stack. When the thread discards its kernel stack, it should not discard saved user register context.

For best performance, this means that the trap handler should write user register values into a save area associated with the thread, instead of writing them to the base of the thread's kernel stack. If this is inconvenient or impossible (perhaps the hardware "helpfully" saves register context in the wrong place), the implementation can always copy the user register context off the stack during a stack discard or stack handoff operation.

A related issue concerns callee-saved registers during a system call. As I discussed in Section 3.2.2, a common optimization avoids saving and restoring these registers explicitly in the system call entry and exit paths because the C compiler saves and restores these registers automatically as the kernel executes. However, this optimization is incompatible with stack discarding, so any system calls that may possibly make use of stack discarding must use longer entry and exit paths.

3.3.5 Summary

The requirements for a control transfer interface that will support continuations, stack discarding, and stack handoff can be summarized as follows:

- Machine-dependent code should have the option of controlling stack memory allocation, but not be required to supply this functionality.

- A variant of the stack allocation operation should allow a non-blocking allocation attempt that doesn’t try to acquire locks in other modules, such as the virtual memory subsystem. Stack deallocation also should avoid calling out to other modules.
- The interface should support an operation that garbage-collects any unused but cached stacks.
- The stack allocation and discarding operations should support the notion of stack-privileged threads and reserved stacks.
- Stack handoff should be directly exposed, with an operation that merely changes the current address space and thread before returning.
- The interface should supply an operation that calls a continuation function while resetting the stack, to prevent overflow due to stack handoff.
- The context-switch operation should support multiprocessors without creating dependencies between machine-independent and machine-dependent code.
- The context-switch and stack handoff operations should automatically change address spaces when the two threads involved belong to different processes.
- The control transfer interface must support multiple kernel entry and exit paths.

These requirements lead directly to the actual interface specification.

3.4 Control Transfer Interface

I have developed a control transfer interface that provides additional flexibility and opportunities for optimization in support of continuations by meeting the requirements of the previous section. In doing this, I have adopted Tevanian’s philosophy for developing Mach’s pmap interface for virtual memory functionality:

> “The pmap interface is not an attempt to abstract all of the functionality provided by popular MMU designs. Instead, it is an interface to the page-based functionality needed by the machine-independent algorithms. Each primitive in the interface provides a specific function as required by machine-independent code.” [Tevanian 87, p. 43]

I’ve applied this approach to control transfer. Rather than abstract hardware support for processes and context-switching, I’ve specified the operations that the machine-independent code actually needs, and left it up to the machine-dependent implementation to map those operations onto the hardware. The result is a conceptually simple yet powerful interface.

`stack_attach(thread, stack, cont)`	give thread a new stack
`stack_detach(thread)`	remove and return thread's stack
`stack_alloc(thread, cont)`	allocate and attach thread's stack
`stack_alloc_try(thread, cont)`	non-blocking `stack_alloc`
`stack_free(thread)`	detach and free thread's stack
`stack_privilege(thread)`	make the thread stack-privileged
`stack_collect()`	garbage-collect unused stacks
`stack_handoff(curthd, nextthd)`	switch to a new thread on same stack
`switch_context(curthd, cont, nextthd)`	switch to a new thread and stack
`call_continuation(cont)`	call the continuation function
`current_thread()`	return pointer to current thread
`thread_syscall_return(retval)`	exit kernel from system call
`thread_exception_return()`	exit from exception or interrupt

Table 3-1: Control Transfer Interface

The new interface allows machine-independent thread management and inter-process communication code to change address spaces, to manage the relationship of kernel stacks and threads, and to create and call continuations. Table 3-1 lists the operations. (Table 3-1 omits some details. See Appendix B for a complete description of the interface.)

The interface does not include any functions for examining a blocked thread's continuation function. 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 control transfer interface uses a compile-time "feature" conditional to give implementations the option of controlling stack management. Machine-dependent implementations can choose between two variants of the interface:

- Machine-dependent code can take over full responsibility for stack management by defining `MACHINE_STACKS` in a header file. In this case, machine-dependent code must define `stack_alloc`, `stack_alloc_try`, `stack_free`, and `stack_collect`.

- If machine-dependent code does not define `MACHINE_STACKS`, then it must define the `stack_attach` and `stack_detach` functions. In this case, machine-independent code defines `stack_alloc`, `stack_alloc_try`, `stack_free`, and `stack_collect` using the kernel's standard virtual memory interface and the `stack_attach` and `stack_detach` operations.

This is the only use of the `MACHINE_STACKS` conditional in machine-independent code; all clients of the control transfer interface rely solely on the `stack_alloc`/`stack_free` version of the interface.

The following subsections further describe the interface, by giving example implementations of two basic scheduling primitives, and discuss the interface's portability.

3.4.1 Using the Interface

The implementation of Mach's higher-level thread management operations makes a nice example of the use of the control transfer interface. The thread management operations manipulate the scheduling data structures—the run queues, the wait queues, state information associated with each thread—in addition to using the interface for actual control transfers. Table 3-2 presents the Mach 3.0 thread management operations that are relevant here.

The following subsections describe these operations and sketch their implementation in terms of the control transfer interface primitives. These subsections omit some unimportant details of the actual implementation of `thread_handoff` and `thread_block`.

`thread_handoff(oldthd, cont, nextthd)`	handoff to a thread, if possible
`thread_block(cont)`	selects and runs a new thread

Table 3-2: Thread Management Operations

3.4.1.1 thread_block

The `thread_block` function blocks the current thread and chooses another thread for execution. By default, the current thread is left runnable. It is also possible for the current thread to enter a waiting state, but this example does not describe that mechanism.

The caller has the option of specifying a continuation function. `thread_block`'s behavior depends on the value of this argument:

- If a continuation is specified, then the current thread is left blocked without a stack and the `thread_block` call does not return. Instead, when the thread resumes the continuation is called. In this case, `thread_block` uses the more efficient `stack_handoff` path if possible but sometimes it is forced to use `switch_context`, which changes stacks.

- If the continuation argument is `NULL`, when the thread resumes its `thread_block` call returns. In this case, `thread_block` always uses `switch_context`.

```
thread_continue(prev_thread) {
    cur_thread = current_thread();
    thread_dispatch(prev_thread);
    // call_continuation not necessary here
    (*cur_thread->cont)();
    /*NOTREACHED*/
}
thread_dispatch(prev_thread) {
    if (prev_thread->cont) {
        // detach and free thread's stack
        stack_free(thread);
    }
    if (prev_thread->state == RUNNING) {
        // return old_thread to run queue
        thread_setrun(prev_thread);
    }
}
```

Figure 3-4: thread_block helper functions

Statically, most calls to `thread_block` do not supply a continuation. Dynamically, however, most calls to `thread_block` do use a continuation and `thread_block` does make use of `stack_handoff`—see Section 5.2.

```
thread_block(cont) {
    old_thread = current_thread();
retry:
    // select a runnable thread from the ready queue
    next_thread = thread_select();
    if (next_thread->cont) {
        if (cont &&
            old_thread not using reserved stack) {
            // handoff changes current_thread()
            stack_handoff(next_thread);
            // now current_thread() == next_thread
            old_thread->cont = cont;
            if (old_thread->state == RUNNING)
                // return to ready queue
                thread_setrun(old_thread);
            call_continuation(next_thread->cont);
            /*NOTREACHED*/
        } else {
            // allocate a new stack
            if (!stack_alloc_try(next_thread,
                                 thread_continue)) {
                add next_thread to stack-alloc queue;
                wakeup stack-alloc thread;
                goto retry;
            }
        }
    }
    // go to sleep - returns only if cont is NULL
    prev_thread = switch_context(old_thread, cont,
                                 next_thread);
    // old_thread is running again
    thread_dispatch(prev_thread);
}
```

Figure 3-5: Implementing thread_block

Figure 3-5 shows the implementation of `thread_block`, and Figure 3-4 depicts its helper functions. The `thread_continue` function executes when a thread blocked without a stack resumes execution on a new stack. The `thread_dispatch` function, called from `thread_block` and `thread_continue` after a context switch, disposes of the previously running thread. This implementation illustrates several aspects of the control transfer interface.

Stack Handoff

The `stack_handoff` function changes the identity of the currently executing thread. That is, `current_thread()` returns a different value after `stack_handoff`. In addition, `stack_handoff` changes the current address space if the current thread and the next thread reside in different address spaces. Because `stack_handoff` does not call the next thread's continuation function directly, `thread_block` must do this with `call_continuation`.

The `thread_block` implementation can only make use of `stack_handoff` if the next thread blocked with a continuation and the current thread is also blocking with a continuation function. Furthermore, `thread_block` must check that the current thread is not running on a reserved stack.

Calling Continuations

The `call_continuation` function calls its function pointer argument. The usage

```
call_continuation(continuation);
```

is equivalent to

```
(*continuation)();
```

except that `call_continuation` also resets the stack pointer to the base of the stack.

When `thread_block` calls the new thread's continuation after `stack_handoff`, it must use `call_continuation` to prevent stack overflow. Otherwise the continuation function could chew up some stack space and eventually call thread_block again, resulting in another stack handoff and more stack space consumption, repeating until eventually the kernel stack would overflow.

In `thread_continue`, on the other hand, `call_continuation` is not necessary: `thread_-continue` is already executing at the base of the current stack, because the stack was just attached to the thread via `stack_alloc_try`.

Stack Management

The `stack_alloc_try` function takes a thread and a continuation as arguments. It makes a non-blocking attempt to allocate a kernel stack and attach it to the thread, returning a boolean value to indicate success or failure. If successful, the thread and stack are initialized such that when the thread resumes (via `switch_context`), the thread executes the continuation function on its new stack. The continuation function takes as an argument the previously executing thread.

In `thread_block`'s implementation, the continuation function is always `thread_continue`. `thread_continue` in turn calls the thread's "true" continuation function, after disposing of the previously executing thread.

The `stack_free` function takes a thread as an argument. It discards the thread's stack. In this example, the `thread_dispatch` function calls `stack_free` on the previously executing thread if that thread blocked with a continuation function, and hence doesn't need its stack.

Although the `thread_block` implementation doesn't use them directly, the control transfer interface contains two more stack management operations, `stack_alloc` and `stack_collect`. When `stack_alloc_try` fails to allocate a stack, `thread_block` puts the thread on a work queue for a special internal stack-allocation thread to pick up. This thread can use the potentially blocking `stack_alloc` call to allocate a stack. The virtual memory system calls `stack_collect` when the kernel needs more free physical memory; `stack_collect` should free any unused stacks cached in the stack management implementation.

Context-Switch

The `switch_context` function performs a context-switch to another thread, changing address spaces if the current thread and the next thread belong to different address spaces. In addition to the current and next threads, `switch_context` takes as an argument the continuation function for the current thread. If this is not `NULL`, then `switch_context` need not save register state for the blocking thread. Otherwise `switch_context` must save the callee-saved registers in preparation for a future `switch_context` that will resume the blocking thread.

The only tricky aspect of this concerns the treatment of the previously running thread. `switch_context` returns a pointer to the previously running thread. The `thread_block` implementation calls `thread_dispatch` on the previous thread, to discard the thread's stack (if it blocked with a continuation function) and otherwise finalize its scheduling state. The other possibility is that `switch_context` doesn't return; in this case, when the thread resumes it executes a continuation function (the argument to `stack_alloc` or `stack_alloc_try`) on a new stack and this continuation function receives the previously running thread as an argument.

This technique is very similar to the way first-class continuations work in programming languages such as Scheme [Steele 78; Clinger & Rees 92]. In Scheme, `call-with-current-continuation` captures a continuation that represents the future execution of the current expression. The expression can either return a value normally or call the continuation with a return value as an argument. In our case, the return value is the previously running thread, the current expression is a call to `switch_context`, and the continuation is either the return context of `switch_context` or an explicit continuation function.

3.4.1.2 thread_handoff

The `thread_handoff` function blocks the current thread with a continuation and hands control to a specific next thread. Figure 3-6 presents the `thread_handoff` implementation. It essentially just a stack handoff plus the appropriate updating of scheduling state. There are two important points to note:

- `thread_handoff` may fail (return `FALSE`). This can happen when the next thread is in an inappropriate scheduling state (for example, another processor just started running the next thread, or the next thread is not allowed to run on the current processor due to processor set constraints) or when the current thread is stack-privileged and running on its reserved stack (see Section 3.3.2.3).

- If successful, `thread_handoff` returns normally except that the value of `current_thread()` has changed, and the current address space has changed if the threads are in different tasks. It does *not* call the new thread's continuation.

```
boolean_t
thread_handoff(old_thread, cont, next_thread) {
    // current_thread() == old_thread

    if (old_thread using reserved stack ||
        next_thread can't run)
        return FALSE;
    next_thread->state = RUNNING;

    // stack_handoff changes current_thread()
    stack_handoff(next_thread);
    // now current_thread() == next_thread

    old_thread->state = WAITING;
    old_thread->cont = cont;
    return TRUE;
}
```

Figure 3-6: Implementing thread_handoff

Because `thread_handoff` returns without calling the new thread's continuation, `thread_handoff`'s caller can perform continuation recognition—it can check the continuation and decide if it wants to call it or take a faster, specialized path instead. See Section 4.2.5 for an example of this optimization.

3.4.2 Portability

In practice, the control transfer interface has met my requirement for portability, that it be at least as portable as a simple context-switch primitive. Two things validate this claim. First, implementations for a substantial number of architectures exist as part of Mach 3.0. Second, there is a relatively straightforward (albeit inefficient) way of transforming an existing context-switch implementation to a full implementation of the control transfer interface.

The control transfer interface has been ported to a variety of CPU architectures. I originally did implementations for the Intel 80386/80486 and MIPS R3000 processors. Subsequent implementations by other people include the Intel 860, Motorola 88100, Motorola 68030/68040, Sun Sparc, DEC VAX, DEC Alpha, and National Semiconductor 32532 processors. Currently, only the MIPS and Alpha implementations define the `MACHINE_STACKS` conditional and implement custom stack management.

Given an existing context-switch implementation, it is normally fairly easy to produce a full implementation of the control transfer interface. Potential difficulties arise in two places. First, the existing trap handling code might save user register context on the thread's kernel stack. The `stack_attach`, `stack_detach`, and `stack_handoff` functions can work around this problem by copying the user register context between the kernel stack and another save area in the thread structure. Second, the processor may have a special context-switch instruction that must be used to change address spaces. The `stack_handoff` operation can work around this by constructing a dummy context, identical to the current context, and performing a hardware context-switch internally. These techniques result in undesirable performance for `stack_handoff`, but they are available when one desires a quick implementation of the control transfer interface.

Chapter 4

Using Continuations

After one has in place a framework for using continuations in an operating system kernel, some important questions remain: “When should they be used?” and “How are they used?” Using MockIPC, an extended example based on Mach IPC, I discuss five general-purpose techniques for using continuations. I also briefly review the use of continuations in the Mach 3.0 kernel. This experience leads me to conclude that with relatively little effort one can modify an operating system kernel to make productive use of continuations.

The question of when to use continuations is best answered via an analysis of cost versus benefit. In an operating system kernel written in a traditional systems programming language such as C there is a software engineering cost associated with using continuations. Because of the lack of language support, using continuations requires some extra effort on the part of the kernel programmer. It is important to identify those areas that will benefit sufficiently to make continuations an appropriate optimization. The answers can vary, depending on whether the performance goal is decreasing the latency of important control transfer paths or minimizing the space overhead in the operating system kernel.

I present five general-purpose techniques for using continuations, along with advice on when to apply these techniques. Three of these techniques provide different styles for using continuations. These techniques offer a range of tradeoffs between performance, modularity, and code impact. The remaining two techniques concern continuation recognition, an optimization that reduces the latency of dynamically frequent control transfer paths. One approach to continuation recognition relies on stack handoff and is appropriate in synchronous, RPC-like settings. The other approach applies more broadly, allowing one thread to optimize another thread's execution.

Finally, in this chapter I review how I modified the Mach 3.0 kernel to use continuations. Because of Mach's message-passing microkernel structure, interprocess communication was the first candidate for continuations. In all, I modified six areas of the Mach kernel using the techniques described here; most of the Mach kernel source code remained unaffected.

4.1 When to Use Continuations

The following steps should drive the process of deciding whether continuations are an appropriate optimization for a particular operating system kernel:

- *Identify performance objectives.* If performance is not a concern, then continuations are probably not appropriate. Continuations can reduce the space consumed by kernel stacks and reduce the latency of selected control transfer paths.

- *Find candidate code paths.* In general, the "most frequent" code paths are appropriate candidates for continuations, but a more detailed answer depends on the performance objectives.

- *Evaluate software engineering costs.* Two factors contribute to the software engineering cost: implementing the control transfer interface as discussed in the previous chapter, and converting or rewriting the candidate code paths to use continuations. The latter factor largely depends on how those code paths fit into the modular organization of the kernel.

Once continuations have been used to achieve a performance improvement, it often becomes worthwhile to use continuations in more marginal situations. For example, converting internal kernel "worker" threads to use continuations requires almost no additional effort to achieve an incremental performance improvement. In these cases, the marginal benefit of using continuations exceeds the marginal cost, because the one-time cost of implementing the control transfer interface has already been paid.

4.1.1 Candidate Code Paths

The appropriate candidates for continuations and continuation recognition are the "most frequent" code paths in the operating system kernel. If the kernel contains no "most frequent" paths, then continuations are probably not an appropriate optimization.

The definition of “most frequent” varies depending upon the performance objective and the application load. If the objective is reducing memory consumption, then the important control transfer paths are those in which most threads are blocked most of the time. If the objective is reducing control-transfer latency with stack handoff and continuation recognition, then the important paths are those which occur frequently at run-time and dominate the kernel’s control transfer execution overhead.

These two definitions can produce different candidate code paths. For example, suppose almost all threads spend large amounts of time blocked in path A, while the few remaining threads context-switch furiously amongst themselves via path B. If memory savings is the objective, then one would focus attention on path A and the consequent savings from stack discarding. If latency is the objective, then one would focus attention on path B to realize the benefits of stack handoff and continuation recognition.

Continuation recognition, which reduces control transfer latency, has prerequisites in addition to dynamic frequency of the control transfer path. Handoff control transfer paths can most easily take advantage of continuation recognition. In these cases, a thread A that is blocking with a continuation specifies a direct control transfer to a resuming thread B that blocked with a continuation, and then goes on to execute an optimized code path instead of calling thread B’s continuation. However, handoff control transfer is not necessary for continuation recognition.

More generally, continuation recognition is possible whenever one thread’s activity impinges on a blocked thread, and the executing thread A can take advantage of precise knowledge of the blocked thread B’s state. As a simple example, thread A might be retrieving or changing thread B’s user register state, which it can access directly if thread B is in certain known states. In other cases, the executing thread A might have some knowledge or information about the state of the system that is important to the blocked thread B but wasn’t available to it when it blocked. Then continuation recognition allows thread A to modify thread B’s continuation in light of the new system state.

4.1.2 Software Engineering Costs

Two contributions to software engineering cost must be evaluated when one considers continuations. First, one must implement the control transfer interface that makes using continuations possible. Section 3.4 discusses this one-time effort; I won’t consider it further here. Second, one must convert or rewrite candidate code paths using the techniques of Section 4.2.

Ill-considered use of continuations can lead to the following problems:

- *Excessive code duplication.* Continuation functions can duplicate fragments of code found in other functions.

- *Inter-module implementation dependencies.* Continuation functions are easiest to write given global knowledge of the kernel implementation, and without care they can introduce dependencies between unrelated modules.

- *Loss of generality*. Continuation-using functions are easiest to write when they make assumptions about the context in which they will be called. Adding continuations can turn a general-purpose function into a function optimized for a specific purpose. This hampers future code reuse.

Continuations work best when the blocking thread needs to save only a few words of state information. With continuations, any state needed across a blocking operation must be manually saved and restored. This is both burdensome for the programmer and expensive in terms of performance. The thread structure must be larger to accommodate the state (or an annex structure must be allocated and deallocated) and unnecessary cycles may be spent saving the state in some cases.

Continuations also work best when the blocking kernel operation occurs entirely inside a single code module. Control transfers that occur inside deeply-nested call chains can be very awkward. The continuation must replace or execute pieces of multiple functions. This is always ugly, and it is especially difficult if the call chain spans multiple code modules.

On the other hand, the techniques explored in the next section make blocking with a continuation inside shallow call chains quite acceptable. In performance-critical control transfer paths, deeply-nested call chains are undesirable in any case because of the function call overhead.

4.1.3 Microkernel Operating Systems

Continuations can be a particularly effective optimization for “microkernel” operating systems. In these systems, the kernel exports a small interface and implements only a few abstractions. This maximizes the performance benefits and minimizes the software engineering costs of using continuations. Although continuations can also improve the performance of “monolithic” operating systems, in such systems it is more difficult for a focused optimization to achieve across-the-board improvements.

Microkernel operating systems do well with continuations because it is easy to identify a stable set of “most frequent” control transfer paths. For example, in a communication-oriented microkernel most application loads will stress the interprocess communication paths in the kernel. In the case of Mach 3.0, there are roughly 60 different points where a thread can block in the kernel, and most of those points have many different continuations. However, over 99% of the blocking operations occur in only six ways, *and* these “hot spots” are mostly independent of the work-load.

In contrast, “monolithic” operating systems such as Mach 2.5 don’t benefit as much from continuations. There are over 180 places in Mach 2.5 where a thread can block, and there are no real hot spots. X Window System [Scheifler & Gettys 90] applications stress the socket control transfer paths, file system applications block in the buffer cache, terminal programs exercise the character IO and serial line code, *etc*. For many particular application loads, like a network file server, one can find “most frequent” control transfer paths that might justify the use of continuations. However, across a broad spectrum of applications such hot spots are harder to find in a monolithic kernel.

4.2 How to Use Continuations

In this section, I use an extended example to explain how to actually use continuations. I present five techniques, including three alternatives for using continuations and taking advantage of stack discarding, and two methods for using continuation recognition. Although for didactic simplicity the presentation uses variations on a single example derived from Mach 3.0’s interprocess communication code, the techniques discussed here have general applicability.

The three alternatives for using continuations offer a range of tradeoffs. The first technique uses continuations as “structured gotos.” The result can actually be quite elegant and efficient, but it tends to produce very specialized functions optimized for a single calling context. The second technique adds some arguments and glue code to a blocking function, so that it can also block with a continuation. This produces somewhat ugly code, but it is a simple modification that preserves the original functionality and organization of the code while adding support for continuations. The third technique produces the most “modular” code. It allows a function’s callers to specify a continuation, without requiring that they have knowledge of the function’s implementation or that the function have knowledge of its callers.

The remaining techniques offer two different approaches to using continuation recognition. The fourth technique combines stack handoff and continuation recognition to achieve very efficient handoff control transfer paths. This combination blurs the distinction between the blocking thread and the resuming thread, with the result resembling a control transfer path with a single “migrating” thread. The fifth technique uses continuation recognition without an explicit stack handoff. With this technique, an executing thread examines and modifies the continuation of a blocked thread to optimize the blocked thread's subsequent execution.

4.2.1 MockIPC Introduction

The background for this section is a message-passing interprocess communication system that I call MockIPC. In semantics and implementation, MockIPC resembles Mach 3.0 IPC, but it is greatly simplified to omit distracting and unimportant details. The resemblance to Mach IPC makes MockIPC more realistic, but this resemblance does not limit the generality of the techniques described here. For example, Section 4.3 describes how these techniques were applicable in areas of the Mach kernel other than interprocess communication.

4.2.1.1 Overview

In MockIPC, messages are sent to and received from ports, which are protected message queues. A single system call, `mock_msg`, can send a message to a port, or receive a message from a port, or send a message to one port and receive a message from a different port. In fact, this is the most common usage of `mock_msg`: client programs send a request message and receive a reply message; server programs send a reply message and receive another request message. Both the sending and receiving operations can block (the sending operation if the message queue is full, the receiving operation if the message queue is empty), but the receiving operation blocks much more frequently than the sending operation. At any given time, most threads are blocked in a receive operation.

The following characteristics of MockIPC are important here:

- The blocking receive operation is a good candidate for continuations. Because most threads are blocked most of the time waiting for a message, stack discarding can reduce the kernel's memory requirements.

- Because of its dynamic frequency, a message handoff from a sending thread to a waiting receiving thread is a good candidate for continuation recognition. Continuation recognition can reduce the latency of this operation, because the sending thread can modify the receiver's continuation to avoid redundant or unnecessary effort.

- In the implementation, the receive operation is not too deeply nested. (It happens in `ipc_mqueue_receive`, called from `mock_msg_receive`, called from `mock_msg`.) This means that the software engineering cost of using continuations in MockIPC is modest.

The blocking message send operation is similarly nested, but it would be slightly harder to code with continuations, because the continuation for a send operation might include a receive operation, in which case more state variables would have to be saved and restored. In any case, if blocking send operations are infrequent there is little reason to optimize them with continuations.

The MockIPC code presented here resembles Mach 3.0's IPC implementation. Mach IPC has functions with the same or similar names (`mach_msg`, `ipc_mqueue_receive`, *etc*) that do very similar things. MockIPC has been simplified from Mach IPC in the following ways:

- Omitted IPC options and features. (See Appendix A for a description of Mach IPC.)
- Simplified locking, with no concern for deadlock.
- Limited error handling.
- No concern for the termination or deactivation of threads and ports.
- No reference counting or garbage collection.

Probably the most relevant difference is that MockIPC uses different synchronization primitives. To simplify the example, MockIPC uses mutexes and condition variables instead of the equivalent Mach kernel functionality.

`mock_msg`	System call entry point. Sends and/or receives a message.
`mock_msg_send`	The message-send half of a system call.
`mock_msg_receive`	The message-receive half of a system call.
`ipc_mqueue_send`	Internal primitive for sending a message. The destination port is specified in the message itself.
`ipc_mqueue_receive`	Internal primitive for receiving a message from a message queue. The message queue is initially locked and is unlocked upon return.
`ipc_mqueue_copyin`	Internal primitive for converting a user's handle into a message queue, which is returned locked.

Table 4-1: MockIPC Functions

The important functions in MockIPC, whose implementation I explore, are listed in Table 4-1. `mock_msg` is the system call entry point; `mock_msg_send` and `mock_msg_receive` are helper functions that are only called from `mock_msg`. In contrast, `ipc_mqueue_send`, `ipc_mqueue_receive`, and `ipc_mqueue_copyin` are also called from other parts of the kernel to manipulate message queues. Their use in varied and unpredictable contexts helps keep MockIPC from being a trivial example.

The unimportant functions in MockIPC, whose implementation I do not explore, are listed in Table 4-2. For our purposes, these functions are just placeholders that indicate code motion and prevent variables from being “dead.”

`ipc_kmsg_get`	Allocate a kernel message structure and copy into it a message from the current user address space.
`ipc_kmsg_copyin`	Translate port handles in the message to pointers to the actual port data structures.
`ipc_kmsg_copyout`	Translate port pointers in the message to handles.
`ipc_kmsg_put`	Copy a message to the current user address space and deallocate the kernel message structure.
`ipc_kmsg_destroy`	Deallocate a kernel message structure.
`ipc_entry_lookup`	Lookup a port handle.

Table 4-2: MockIPC Helper Functions

The synchronization functions, which are listed in Table 4-3, are a slight variation on the traditional mutex and condition variable primitives for concurrent programming [Birrell 89; Nelson 91, chapter 4; Cooper & Draves 88]. The variations occur in the functions for blocking on a condition, `twait` and `thandoff`. Both functions take a `timeout` argument that limits the potential duration of the wait. If the wait terminates, either because of a `signal` or the timeout expiring, the `twait`/`thandoff` call returns the time remaining in the timeout. (This is zero if the timeout expired.) As is usual, the thread can’t make any assumptions about why the wait terminated. It must always recheck the shared state before taking action.

`lock`	Lock a mutex, if necessary waiting until the mutex is available.
`unlock`	Unlock a mutex.
`signal`	Wakeup a thread waiting on the condition variable.
`twait`	Wait with timeout on the condition variable, with the mutex released while blocked. Returns the time remaining.
`thandoff`	Combined `signal`/`twait` operation, to block on a condition while waking up a specified thread.

Table 4-3: MockIPC Synchronization Functions

`thandoff` adds the additional wrinkle of a directed context-switch. It allows a thread to wait on a condition variable and specify a particular thread to wake up, effecting a direct transfer of control to the specified thread. For this example, condition variables are just queues of threads, and the queue primitives (`empty`, `first`, `dequeue`, `enqueue`) can operate on condition variables while the lock governing the condition is held. `thandoff` is used by dequeueing a thread from one condition variable and then at a later time handing off to that thread while waiting on another condition variable.

4.2.1.2 Before Continuations

Before delving into the various ways of using continuations in MockIPC, it will be useful to establish a baseline for comparison. Figures 4-1 and 4-2 give the implementation, without continuations, of `mock_msg`, `mock_msg_send`, `mock_msg_receive`, `ipc_mqueue_send`, `ipc_mqueue_receive`, and `ipc_mqueue_copyin`.

```
mock_msg(msg, option, snd_size, rcv_size, rcv_name, timeout) {
    if (option & SEND) {
        rc = mock_msg_send(msg, snd_size, timeout);
        if (rc != SUCCESS)
            return rc;
    }
    if (option & RECEIVE) {
        rc = mock_msg_receive(msg, rcv_size, rcv_name, timeout);
        if (rc != SUCCESS)
            return rc;
    }
    return SUCCESS;
}
mock_msg_send(msg, snd_size, timeout) {
    kmsg = ipc_kmsg_get(msg, snd_size);
    ipc_kmsg_copyin(kmsg);
    rc = ipc_mqueue_send(kmsg, timeout);
    return rc;
}
mock_msg_receive(msg, rcv_size, rcv_name, timeout) {
    mqueue = ipc_mqueue_copyin(rcv_name);
    // mqueue is locked
    kmsg = ipc_mqueue_receive(mqueue, timeout);
    // mqueue is unlocked
    if (kmsg == NULL)
        return TIMED_OUT;
    ipc_kmsg_copyout(kmsg);
    ipc_kmsg_put(kmsg, msg, rcv_size);
    return SUCCESS;
}
```

Figure 4-1: MockIPC Send and Receive

The implementation is quite straightforward. A few points to note:

- Although it isn't coded quite this way, `mock_msg` really just returns `mock_msg_receive`'s return code directly. That is, `mock_msg`'s call of `mock_msg_receive` is "tail recursive" and `mock_msg_receive` has the same continuation (in the language sense) as `mock_msg`. This means that the apparent three-level call chain (`mock_msg` to `mock_msg_receive` to `ipc_mqueue_receive`) has the complexity of a two-level call chain; the call `mock_msg` to `mock_msg_receive` adds no complexity.

- `ipc_mqueue_send` always consumes the `kmsg` that it is handed, either by enqueueing it or destroying it.

- The locking in `mock_msg_receive` is a little unorthodox. `ipc_mqueue_copyin` returns the message queue in the locked state, and `ipc_mqueue_receive` takes a locked message queue and returns with it unlocked.

- When `ipc_mqueue_send` and `ipc_mqueue_receive` block with `twait`, they can make no assumptions about the state of the message queue—whether it is empty, full, or partly full—when they wakeup. This is why they both use `twait` inside a while loop.

```
ipc_mqueue_send(kmsg, timeout) {
    mqueue = kmsg->dest_port->mqueue;
    lock(mqueue->lock);

    while (mqueue->count >= mqueue->limit) {
        if (timeout == 0) {
            unlock(mqueue->lock);
            ipc_kmsg_destroy(kmsg);
            return TIMED_OUT;
        }

        timeout = twait(mqueue->lock, mqueue->full, timeout);
    }

    mqueue->count++;
    enqueue(mqueue->queue, kmsg);
    signal(mqueue->empty);

    unlock(mqueue->lock);
    return SUCCESS;
}

ipc_mqueue_copyin(rcv_name) {
    table = current_task()->name_table;

    lock(table->lock);
    mqueue = ipc_entry_lookup(table, rcv_name);
    lock(mqueue->lock);
    unlock(table->lock);
    return mqueue;
}

ipc_mqueue_receive(mqueue, timeout) {
    while (empty(mqueue->queue)) {
        if (timeout == 0) {
            unlock(mqueue->lock);
            return NULL;
        }

        timeout = twait(mqueue->lock, mqueue->empty, timeout);
    }

    kmsg = dequeue(mqueue->queue);
    mqueue->count--;
    signal(mqueue->full);

    unlock(mqueue->lock);
    return kmsg;
}
```

Figure 4-2: MockIPC Queue Primitives

4.2.1.3 With Handoff

Without using continuations or continuation recognition, one approach to optimizing MockIPC uses thread handoff. A handoff is a directed control transfer, or a control transfer in which the blocking thread transfers to a specific resuming thread. Handoffs can take advantage of optimized scheduling code [Black 90a]. A handoff MockIPC implementation provides an appropriate comparison baseline for the use of continuation recognition.

```
mock_msg(msg, option, snd_size, rcv_size, rcv_name, timeout) {
    if (option == (SEND|RECEIVE)) {
        // RPC path - attempts handoff

        kmsg = ipc_kmsg_get(msg, snd_size);
        ipc_kmsg_copyin(kmsg);

        rcv_mqueue = ipc_mqueue_copyin(rcv_name);
        // rcv_mqueue is locked

        snd_mqueue = kmsg->dest_port->mqueue;
        lock(snd_mqueue->lock);

        // check handoff conditions
        //  - there is a receiving thread to wakeup
        //  - we block because our queue is empty

        if (empty(snd_mqueue->empty) ||
            !empty(rcv_mqueue->queue) ||
            (timeout == 0)) {
            // can't handoff - abort

            unlock(snd_mqueue->lock);
            unlock(rcv_mqueue->lock);

            rc = ipc_mqueue_send(kmsg, timeout);
            goto after_msg_send;
        }

        // queue the message

        snd_mqueue->count++;
        enqueue(snd_mqueue->queue, kmsg);

        // dequeue the receiving thread
        rcv_thread = dequeue(snd_mqueue->empty);
        unlock(snd_mqueue->lock);

        // do scheduling handoff; combined wakeup and twait
        timeout = thandoff(rcv_thread,
                           rcv_mqueue->lock, rcv_mqueue->empty,
                           timeout);
        // rcv_mqueue is locked again after wakeup

        kmsg = ipc_mqueue_receive(rcv_mqueue, timeout);
        if (kmsg == NULL)
            return TIMED_OUT;

        ipc_kmsg_copyout(kmsg);
        ipc_kmsg_put(kmsg, msg, rcv_size);
        return SUCCESS;
    }

    if (option & SEND) {
        rc = mock_msg_send(msg, snd_size, timeout);
    after_msg_send:
        if (rc != SUCCESS)
            return rc;
    }

    if (option & RECEIVE) {
        rc = mock_msg_receive(msg, rcv_size, rcv_name, timeout);
        if (rc != SUCCESS)
            return rc;
    }

    return SUCCESS;
}
```

Figure 4-3: MockIPC Handoff

Figure 4-3 shows the use of handoff without continuations in MockIPC. In this example, the handoff code exists as an optimized, inline path in `mock_msg`. The bulk of the MockIPC code, in `mock_msg_send` and `mock_msg_receive`, does not depend on the existence of this optimized path; the handoff code could be removed if it proved troublesome and MockIPC would continue to function. On the other hand, the handoff path does depend on the specific implementation of `mock_msg_send` and `mock_msg_receive` and in fact the handoff path reproduces fragments of that implementation.

Handoff is possible when the following conditions hold:

- `mock_msg` is used to both send and receive a message.
- A thread is waiting to receive the message sent in the first half of `mock_msg`.
- The receive half of the `mock_msg` will block.

In this situation, `mock_msg` uses `thandoff` to perform the directed control transfer. If the proper conditions do *not* hold, then the handoff path cleans up and jumps to the generic `mock_msg` code.

Note that the handoff code performs exactly the same message-queue operations as the separate send and receive functions, although in a different order. In particular, before blocking the handoff path performs the following message-queue operations:

1. locks `rcv_mqueue` (in `ipc_mqueue_copyin`)
2. locks `snd_mqueue`
3. checks the state of `snd_mqueue`
4. checks the state of `rcv_mqueue`
5. increments the message count in `snd_mqueue`
6. enqueues a message in `snd_mqueue`
7. dequeues a thread from `snd_mqueue`
8. unlocks `snd_mqueue`
9. enqueues a thread in `rcv_mqueue` (in `thandoff`)
10. unlocks `rcv_mqueue` (in `thandoff`)

After unblocking, the handoff path:

11. locks `rcv_mqueue` (in `thandoff`)
12. checks the state of `rcv_mqueue`
13. dequeues a message from `rcv_mqueue`
14. decrements the message count in `rcv_mqueue`
15. unlocks `rcv_mqueue`

These operations are all required in this version of the handoff path, to maintain the consistency of the message-queue data structures. (They are unlocked during `thandoff`.) As you will see, continuation recognition can eliminate about half of these operations.

It's worth noting that handoff in MockIPC could be implemented in other ways. For example, `mock_msg_send` and `ipc_mqueue_send` could be modified not to wakeup any threads, but instead dequeue from a condition variable and return a thread that should be woken up. `mock_msg_receive` and `ipc_mqueue_receive` would then take this thread as an argument that if non-`NULL` would allow handoff to occur inside `ipc_mqueue_receive`.

However, such an implementation would not be as amenable to the handoff-style continuation recognition of Section 4.2.5. It would keep the sending code and the receiving code segregated, without the opportunity to optimize them together as a unified code path.

4.2.2 First Technique: Continuations as Structured Gotos

The first method that I'll present for converting a blocking operation to use continuations assumes that the functions involved are only used in a single context. In the case of MockIPC, the assumption is that `mock_msg`, `mock_msg_receive`, and `ipc_mqueue_receive` are only used in the context of a system call. The drawback stemming from this assumption is that the transformed functions lose their general-purpose nature, because they can no longer be called in other contexts. The good news is that this method produces relatively efficient, clean code.

The basic idea is to break up the functions along the control transfer path into pieces, with the splitting points being function calls, returns, and blocking operations. If function A calls function B which blocks, then four functions result: A-before-call-to-B, B-before-block, B-after-block, and A-after-call-to-B. Function calls replace function returns. More complicated situations produce a network of functions, with function calls performing state transitions.

Figure 4-4 demonstrates this technique, as applied to MockIPC's receive operation. Here the original code (see Figure 4-1) has `mock_msg_receive` which calls `ipc_mqueue_receive` which can block. The transformed code has `mock_msg_receive` which calls `ipc_mqueue_receive`, which can block and resume in `ipc_mqueue_receive_continue`, which calls `mock_msg_receive_finish`. (Actually, `ipc_mqueue_receive_continue` first calls back to `ipc_mqueue_receive`, which calls `mock_msg_receive_finish`. This implements the loop that is necessary when using `twait`.) Some points to note:

```
mock_msg_receive(msg, rcv_size, rcv_name, timeout) {
    mqueue = ipc_mqueue_copyin(rcv_name);
    // mqueue is locked

    save msg, rcv_size;

    ipc_mqueue_receive(mqueue, timeout);
    // NOTREACHED
}

ipc_mqueue_receive(mqueue, timeout) {
    if (empty(mqueue->queue)) {
        if (timeout == 0) {
            unlock(mqueue->lock);
            thread_syscall_return(TIMED_OUT);
            // NOTREACHED
        }

        save mqueue;

        (void) twait(mqueue->lock, mqueue->empty, timeout,
                     ipc_mqueue_receive_continue);
        // NOTREACHED
    }

    kmsg = dequeue(mqueue->queue);
    mqueue->count--;
    signal(mqueue->full);

    unlock(mqueue->lock);
    mock_msg_receive_finish(kmsg);
    // NOTREACHED
}

ipc_mqueue_receive_continue(timeout) {
    // mqueue is locked
    retrieve mqueue;
    ipc_mqueue_receive(mqueue, timeout);
    // NOTREACHED
}

mock_msg_receive_finish(kmsg) {
    retrieve msg, rcv_size;

    ipc_kmsg_copyout(kmsg);
    ipc_kmsg_put(kmsg, msg, rcv_size);

    thread_syscall_return(SUCCESS);
    // NOTREACHED
}
```

Figure 4-4: First Technique: MockIPC Receive with Continuations

- None of the transformed functions return. Instead they call other functions, which also don't return. Ultimately `thread_syscall_return` is called, and it returns to user level.

- `twait` has acquired another argument, a continuation. If it is non-null, then when the wait terminates and the blocking thread resumes, the thread resumes by executing the continuation function. The `twait` continuation function takes as an argument the time-remaining value that `twait` would normally return.

- `ipc_mqueue_receive` can directly call `thread_syscall_return` when a time-out occurs, because it "knows" that it is called in a system call context. This means that the `mock_msg_receive_finish` code need not check for a null `kmsg`, unlike the corresponding code (before continuations) in Figure 4-1.

A good compiler could take advantage of special declarations or pragmas telling it that these functions never return. With this knowledge, it could avoid saving and restoring registers across the function calls—the machine code generated for these function calls would become simple jumps.

4.2.3 Second Technique: Minimizing Code Impact

The second method for converting blocking code to use continuations preserves the generality of the code—it is useable in multiple contexts—but the functions involved still have implementation dependencies on each other. In the case of MockIPC, this means that the transformed `ipc_mqueue_receive` remains useful outside of a system call context, but `mock_msg_receive` still has knowledge of `ipc_mqueue_receive`'s implementation. In effect, this method provides a compromise between the implementation and context dependencies of the first technique (in Section 4.2.2) and the modularity and encapsulation of the third technique (in Section 4.2.4). The transformed code's complexity also falls between these other two techniques.

The technique adds `resume` and `continue` arguments to the blocking function (in the case of MockIPC, `ipc_mqueue_receive`). When called with a `FALSE resume` and `NULL continue`, the function retains its pre-continuations behavior. This preserves its generality. A non-`NULL continue` argument requests that the function use continuations, and in fact directly specifies the continuation with which it should block. (This is the source of the implementation dependency; the function's caller must have knowledge of the function's implementation to supply a correct continuation function.) The `resume` argument is not required, but it lets the continuation function "get back inside" the blocking function to finish up. The `resume` argument avoids code duplication while minimizing the changes to the blocking function.

Figure 4-5 demonstrates this technique with `mock_msg_receive` and `ipc_mqueue_receive`. `mock_msg_receive` passes a continuation function, `mock_msg_receive_continue`, to `ipc_mqueue_receive`. `mock_msg_receive_continue` also calls `ipc_mqueue_receive`, with `resume` being `TRUE`, to resume the computation after the blocking `twait`. Some points to note:

- Because it is a continuation for `twait`, `mock_msg_receive_continue` accepts a `timeout` argument. This is an implementation dependency. If `ipc_mqueue_receive` were to block in some other way, or change its treatment of the `timeout` value from `twait`, then `ipc_mqueue_receive`'s callers would need to change.

- Because of the positioning of `twait` and the while loop in `ipc_mqueue_receive`, its `resume` argument is not actually necessary—re-entering `ipc_mqueue_receive` from the top effectively resumes it. I included the `resume` argument here because in more general situations it is a useful technique.

```
mock_msg_receive(msg, rcv_size, rcv_name, timeout) {
    mqueue = ipc_mqueue_copyin(rcv_name);
    // mqueue is locked

    save mqueue, msg, rcv_size;

    kmsg = ipc_mqueue_receive(mqueue, timeout,
                              FALSE, mock_msg_receive_continue);

    mock_msg_receive_finish(kmsg);
    // NOTREACHED
}

mock_msg_receive_continue(timeout) {
    // mqueue is locked
    retrieve mqueue;

    kmsg = ipc_mqueue_receive(mqueue, timeout,
                              TRUE, mock_msg_receive_continue);

    mock_msg_receive_finish(kmsg);
    // NOTREACHED
}

mock_msg_receive_finish(kmsg) {
    if (kmsg == NULL) {
        thread_syscall_return(TIMED_OUT);
        // NOTREACHED
    }

    retrieve msg, rcv_size;

    ipc_kmsg_copyout(kmsg);
    ipc_kmsg_put(kmsg, msg, rcv_size);

    thread_syscall_return(SUCCESS);
    // NOTREACHED
}

ipc_mqueue_receive(mqueue, timeout, resume, continue) {
    if (resume)
        goto after_block;

    while (empty(mqueue->queue)) {
        if (timeout == 0) {
            unlock(mqueue->lock);
            return NULL;
        }

        timeout = twait(mqueue->lock, mqueue->empty, timeout, continue);
    after_block:
    }

    kmsg = dequeue(mqueue->queue);
    mqueue->count--;
    signal(mqueue->full);

    unlock(mqueue->lock);
    return kmsg;
}
```

Figure 4-5: Second Technique: MockIPC Receive with Continuations

4.2.4 Third Technique: Modular Continuations

My third method for converting a blocking operation to use continuations encapsulates the transformed function's implementation. The transformed function can be used in multiple contexts and it doesn't expose any implementation details or dependencies to its callers. The disadvantage of this method is that the transformed code can be slightly cumbersome. This method is appropriate when the blocking function resides in a module separate from its callers.[1]

The method adds a `continue` argument to the blocking function (in the case of MockIPC, this is `ipc_mqueue_receive`). This is superficially similar to the technique of Section 4.2.3, which also adds a `continue` argument. The difference is that here, the `continue` argument controls the continuation of the blocking function itself, instead of the blocking operation inside the blocking function. If the blocking function returns a value normally, then the `continue` function takes that value as an argument. Another difference with respect to Section 4.2.3 is that here the blocking function does not expose a `resume` argument, although it may use one internally. Both of these differences hide the internal implementation details of the blocking function from its callers.

Figure 4-6 demonstrates this technique with `mock_msg_receive` and `ipc_mqueue_receive`. The post-`ipc_mqueue_receive` code in `mock_msg_receive` moves into `mock_msg_receive_continue`. `mock_msg_receive_continue` takes a `kmsg` as an argument, because that is what `ipc_mqueue_receive` returns. Internally, `ipc_mqueue_receive` uses `ipc_mqueue_receive_helper` and `ipc_mqueue_receive_continue`, but these details are hidden from `mock_msg_receive`. Some points to note, comparing this code with the previous example in Figure 4-5:

- This technique requires one more function. `ipc_mqueue_receive_helper` hides the `resume` argument implementation from `ipc_mqueue_receive`'s callers.

- This technique requires one more saved state variable, the `continue` argument to `ipc_mqueue_receive`.

- `mock_msg_receive` and `ipc_mqueue_receive` are each responsible for saving and restoring their own state: `msg` and `rcv_size` in `mock_msg_receive`, `mqueue` and `continue` in `ipc_mqueue_receive_helper`.

4.2.5 Fourth Technique: Stack Handoff and Continuation Recognition

In this section, I demonstrate the use of continuation recognition for optimizing "handoff" control transfer paths. A handoff is a directed control transfer, or a control transfer in which the blocking thread transfers to a specific resuming thread. Handoffs can take advantage of optimized scheduling code [Black 90a]. Continuation recognition allows another level of optimization in

---

[1]In fact, the `twait` function itself uses this technique to hide its internal implementation.

handoff situations, because it creates an opportunity for the blocking thread to know what code the resuming thread would execute.

```
mock_msg_receive(msg, rcv_size, rcv_name, timeout) {
    mqueue = ipc_mqueue_copyin(rcv_name);
    // mqueue is locked

    save msg, rcv_size;

    (void) ipc_mqueue_receive(mqueue, timeout,
                              mock_msg_receive_continue);
    // NOTREACHED
}

mock_msg_receive_continue(kmsg) {
    if (kmsg == NULL) {
        thread_syscall_return(TIMED_OUT);
        // NOTREACHED
    }

    retrieve msg, rcv_size;

    ipc_kmsg_copyout(kmsg);
    ipc_kmsg_put(kmsg, msg, rcv_size);

    thread_syscall_return(SUCCESS);
    // NOTREACHED
}

ipc_mqueue_receive(mqueue, timeout, continue) {
    return ipc_mqueue_receive_helper(mqueue, timeout,
                                     FALSE, continue);
}

ipc_mqueue_receive_helper(mqueue, timeout, resume, continue) {
    if (resume)
        goto after_block;

    while (empty(mqueue->queue)) {
        if (timeout == 0) {
            kmsg = NULL;
            goto exit;
        }

        if (continue != 0) {
            save mqueue, continue;

            (void) twait(mqueue->lock, mqueue->empty, timeout,
                         ipc_mqueue_receive_continue);
            // NOTREACHED
        after_block:
        } else {
            timeout = twait(mqueue->lock, mqueue->empty, timeout, NULL);
        }
    }

    kmsg = dequeue(mqueue->queue);
    mqueue->count--;
    signal(mqueue->full);

exit:
    unlock(mqueue->lock);

    if (continue != 0) {
        (*continue)(kmsg);
        // NOTREACHED
    } else {
        return kmsg;
    }
}

ipc_mqueue_receive_continue(timeout) {
    // mqueue is locked
    retrieve mqueue, continue;

    ipc_mqueue_receive_helper(mqueue, timeout, TRUE, continue);
    // NOTREACHED
}
```

Figure 4-6: Third Technique: MockIPC Receive with Continuations

With continuation recognition, a running thread can examine a blocked thread’s continuation, and based on the results of this examination the running thread can perform some optimization. In this case, the optimization changes the code that the blocked thread executes when it resumes via a stack handoff. This is possible for a thread blocked with a continuation function, because the continuation and other relevant state of the blocked thread are easily accessible in the thread structure, instead of being hidden in opaque compiler-dependent stack frames.

Putting this idea into practice, there are two things to keep in mind. First, it is best to check for handoff and recognition as soon as possible; this maximizes the opportunities for optimization. Second, the running thread changes (and optimizes) the blocked thread’s execution by performing a stack handoff to it and then *not* calling its continuation. This technique joins the two halves of the control transfer path, the blocking code and the resuming code, and lets the control transfer path be optimized as a whole inside one function.

Figure 4-7 demonstrates this for MockIPC. The handoff path checks that the waiting thread is blocked with `mock_msg_receive_continue`; this type of check is the defining feature of continuation recognition. Once this is known, the code enters an optimized path that in fact never calls `mock_msg_receive_continue`. Note that the position of the `stack_handoff` call is not very important—it could be placed anywhere after recognition and before the protecting message-queue locks are released.

In this case, the continuation function `mock_msg_receive_continue` (which is not shown here) is taken from Figure 4-5 in Section 4.2.3. Continuation functions drawn from Figure 4-4 or Figure 4-6 could have been used instead.

The complete handoff path performs the following eight message-queue operations:

1. locks `rcv_mqueue` (in `ipc_mqueue_copyin`)
2. locks `snd_mqueue`
3. checks the state of `snd_mqueue`
4. checks the state of `rcv_mqueue`
5. enqueues a thread in `rcv_mqueue`
6. dequeues a thread from `snd_mqueue`
7. unlocks `snd_mqueue`
8. unlocks `rcv_mqueue`

```
mock_msg(msg, option, snd_size, rcv_size, rcv_name, timeout) {
    if ((option & (SEND|RECEIVE)) == (SEND|RECEIVE)) {
        // RPC path - attempts handoff

        kmsg = ipc_kmsg_get(msg, snd_size);
        ipc_kmsg_copyin(kmsg);

        rcv_mqueue = ipc_mqueue_copyin(rcv_name);
        // rcv_mqueue is locked

        snd_mqueue = kmsg->dest_port->mqueue;
        lock(snd_mqueue->lock);

        // check handoff conditions
        //  - there is a receiving thread to wakeup
        //    (and it is blocked with the proper continuation)
        //  - we block because our queue is empty

        if (empty(snd_mqueue->empty) ||
            (first(snd_mqueue->empty)->cont != mock_msg_receive_continue) ||
            !empty(rcv_mqueue->queue) ||
            (timeout == 0)) {
            // can't handoff - abort

            unlock(snd_mqueue->lock);
            unlock(rcv_mqueue->lock);

            rc = ipc_mqueue_send(kmsg, timeout);
            goto after_msg_send;
        }

        // prepare current thread for blocking

        snd_thread = current_thread();
        save rcv_mqueue, msg, rcv_size;
        snd_thread->cont = mock_msg_receive_continue;

        snd_thread->timeout = timeout;
        enqueue(rcv_mqueue->empty, snd_thread);

        // stack handoff to receiving thread

        rcv_thread = dequeue(snd_mqueue->empty);
        stack_handoff(snd_thread, rcv_thread);

        unlock(snd_mqueue->lock);
        unlock(rcv_mqueue->lock);

        retrieve msg, rcv_size;

        ipc_kmsg_copyout(kmsg);
        ipc_kmsg_put(kmsg, msg, rcv_size);
        thread_syscall_return(SUCCESS);
        // NOTREACHED
    }
    if (option & SEND) {
        rc = mock_msg_send(msg, snd_size, timeout);
    after_msg_send:
        if (rc != SUCCESS)
            return rc;
    }
    if (option & RECEIVE) {
        rc = mock_msg_receive(msg, rcv_size, rcv_name, timeout);
        if (rc != SUCCESS)
            return rc;
    }

    return SUCCESS;
}
```

Figure 4-7: Fourth Technique: MockIPC Handoff with Continuation Recognition

Comparing this with the 15 message-queue operations for the equivalent handoff path without continuations and continuation recognition (listed in Section 4.2.1.3 on page 64), it is apparent that continuation recognition eliminated 7 redundant operations. (Those numbered 5, 6, 10, 11, 12, 13, and 14.) The main reason for this is that with continuation recognition, the message-queue data structures never go through an intermediate state in which a message is queued.

4.2.6 Fifth Technique: Asynchronous Continuation Recognition

Continuation recognition without an explicit stack handoff is also a valuable technique. With this method, an executing thread examines and modifies the continuation of a blocked thread to optimize the blocked thread's subsequent execution.

Figure 4-8 demonstrates this in the case of MockIPC. The `ipc_mqueue_send` function checks the queue of threads waiting to receive a message. If it finds a waiting thread, and that thread has the continuation `mock_msg_receive_continue`, then `ipc_mqueue_send` performs a message handoff. Instead of queuing the message, it hands it directly to the receiving thread and changes the receiver's continuation to `mock_msg_receive_continue_fast`. This alternate continuation knows that it doesn't have to dequeue a message, or otherwise look at the message queue.

```
ipc_mqueue_send(kmsg, timeout) {
    mqueue = kmsg->dest_port->mqueue;
    lock(mqueue->lock);

    if (empty(mqueue->empty) ||
        (first(mqueue->empty)->cont != mock_msg_receive_continue)) {

        while (mqueue->count >= mqueue->limit) {
            if (timeout == 0) {
                unlock(mqueue->lock);
                ipc_kmsg_destroy(kmsg);
                return TIMED_OUT;
            }

            timeout = twait(mqueue->lock, mqueue->full, timeout);
        }

        mqueue->count++;
        enqueue(mqueue->queue, kmsg);
        signal(mqueue->empty);
    } else {

        receiver = dequeue(mqueue->empty);
        // optimize receiver's continuation
        receiver->cont = mock_msg_receive_continue_fast;
        receiver->kmsg = kmsg;
        wakeup(receiver);
    }

    unlock(mqueue->lock);
    return SUCCESS;
}

mock_msg_receive_continue_fast() {
    retrieve kmsg, msg, rcv_size;

    ipc_kmsg_copyout(kmsg);
    ipc_kmsg_put(kmsg, msg, rcv_size);

    thread_syscall_return(SUCCESS);
    // NOTREACHED
}
```

Figure 4-8: Fifth Technique: Continuation Recognition without Handoff

The end result is that a complete send-receive pair performs the same eight message queue operations listed in the previous section for the handoff path optimized with continuation recognition. The main difference is that this code doesn't take advantage of handoff scheduling. In addition, because this technique splits the optimized path between two functions (`ipc_mqueue_send` and `mock_msg_receive_continue_fast` in this example) some low-level optimizations such as common subexpression elimination may be inhibited. On the other hand, this use of continuation recognition is more general, because it is not restricted to send-receive pairs. For example, the performance of asynchronous message passing would improve with this method, while the previous technique only improves synchronous RPC performance.

4.3 Using Continuations in Mach

This section reviews the use of continuations in the Mach 3.0 kernel. For each code path that I modified, I summarize the cost/benefit analysis that led to the use of continuations in that code path and I briefly discuss the implementation in terms of the techniques of the previous section. In all I discuss six areas that I modified and one that I could have tackled but chose not to.

The original motivation for developing continuations as a control transfer technique was to reduce the memory footprint of the Mach kernel by discarding the kernel stacks of threads waiting to receive a message. This attempt led to continuations and the stack discarding optimization. My subsequent efforts to improve the performance and portability of the technique led to stack handoff, continuation recognition, and the control transfer interface.

Unfortunately, some historical factors have somewhat distorted the resulting implementation. In particular, it took two years before support for the new control transfer interface existed for all the CPU architectures to which Mach has been ported. In this interim period, machine-independent code used continuations only under a `KEEP_STACKS` compile-time conditional. This limited the extent to which the machine-independent code could be reorganized or rewritten to take advantage of continuations. It favored the technique of Section 4.2.3 because of its minimal code impact. Today the `KEEP_STACKS` conditional no longer exists, but its effects linger.

In retrospect, the following two areas of the Mach kernel offered performance improvements sufficiently compelling to prompt the initial effort of using continuations:

- Interprocess communication. Because most threads are blocked most of the time waiting for a message, optimizing IPC with stack discarding produced a significant reduction in space overhead. The important RPC path was already optimized for low latency, so only a small latency improvement was possible.

- Exception handling. The exception handling path contains some complexity that continuation recognition can bypass. Because it was a relatively unoptimized part of the kernel, applying the handoff continuation recognition technique to exception handling produced a significant latency improvement.

Once I'd already developed a framework for using continuations, the following areas offered small performance improvements for very modest software engineering costs:

- Internal kernel threads. The kernel contains a number of internal "worker" threads that can easily use a continuation when they wait for more work.
- Scheduling. When the scheduling system preempts a thread, or a thread voluntarily yields the processor, it doesn't have any kernel-level state and can easily use a continuation.
- Thread halting. In situations where the Mach kernel interface allows one thread to clobber another thread, the victim thread must be in an appropriate "halted" state. Continuation recognition can easily eliminate two context-switches from the synchronization normally necessary to put a victim thread into the halted state.

The virtual memory system presented two possible uses for continuations:

- Page fault handling. The page fault handler can use continuations and stack discarding while waiting for a page fault to be satisfied. This prevents the possibility of a positive feedback that would cause the kernel to allocate more memory for kernel stacks precisely when the kernel has little available physical memory.
- Extern pager interactions. The external pager interface, like the exception-handling interface, is relatively unoptimized and has a number of complexities and inefficiencies that continuation recognition could address.

In the following subsections, I first present a very brief overview of the Mach kernel abstractions and implementation. I then explore each of the above seven areas in more detail.

4.3.1 Mach Kernel Overview

The Mach kernel provides a small set of abstractions that reflects the underlying hardware facilities: memory, processors, and IO devices. User programs access kernel objects via a message-oriented communication interface, in the same way they would access objects provided by user-level servers. The kernel abstractions and interfaces are sufficiently general that user-level subsystems can emulate other operating system interfaces, such as Unix [Ritchie & Thompson 78; Golub *et al.* 90], MS DOS [Schulman 93; Malan *et al.* 91], and OS/2 [Letwin 88; Phelan *et al.* 93], fairly efficiently.

The kernel supports the following abstractions:

- Ports and messages. A port is a protected message queue. User programs manipulate ports with opaque handles or references known as port rights.
- Tasks. The task is the unit of protection and resource allocation. Tasks have a virtual address space. The kernel virtual memory primitives provide a great deal of flexibility for sparse allocation and mapping, control of protection, and sharing.
- Threads. Threads provide schedulable execution contexts. Threads belong to tasks.
- Memory objects. User "external pager" servers can provide memory objects that may be mapped into multiple address spaces. The kernel manages physical memory as a cache of memory object contents. The kernel communicates with the appropriate server to perform page-in and page-out operations on a memory object.
- Devices. The kernel represents IO devices as device objects with a common open/read/write/ioctl/close message interface. For network interfaces, the kernel converts incoming packets to messages that it sends asynchronously to ports supplied by user programs.

The kernel interfaces rely heavily on the message-oriented interprocess communication facility:

- User programs manipulate all kernel objects (with the exception of ports and messages themselves) through the IPC interface, by sending a request message to the object and waiting for a reply message.
- The kernel communicates with external pagers (servers for memory objects) via the IPC interface. For example, it sends page-in requests to the memory object and the external pager for the memory object responds with a message containing a page of data.
- The kernel handles exceptions such as an illegal memory reference with the IPC interface. Tasks and threads can have exception servers. The faulting thread sends an exception request message to its exception server and then waits for a reply message.

The kernel acts much like an ordinary server for kernel objects. However, in the kernel threads service their own requests. That is, when a thread requests a service of the kernel (sends a message to a kernel object, page faults, takes an exception), the thread enters the kernel address space and handles its own request. This is unlike user-level servers, which must have a pool of threads to handle incoming requests. The kernel does have internal "worker" threads, but they do not service requests from user programs.

Device interrupts do not execute in a thread context. Instead, a device interrupt handler "borrows" the kernel stack of the currently executing thread. Because of this, interrupt handlers suffer from many restrictions—they can't do much other than wakeup a thread.

The kernel implementation supports multiprocessor architectures with fine-grained locking. Typically each kernel object has one or more locks for its state. The kernel uses both simple spin locks and reader-writer blocking locks. Spin locks are a feasible technique because the scheduler does not preempt threads while they are inside the kernel, and threads holding a spin lock do not access pageable memory or perform other possibly-blocking operations. The spin lock implementation uses conditional compilation to compile-out spin lock overhead on uniprocessor architectures.

Internally the kernel uses dynamic memory allocation for most of its needs, including the structures that represent ports, messages, and other kernel objects. The memory allocation primitives inside the kernel may block while the virtual memory system recovers physical memory.

Kernel code and most kernel data, including the structures that represent kernel objects, are not pageable. However, threads executing inside the kernel may access pageable user memory.

Threads executing user code can not block directly; they must enter the kernel address first. This may happen directly when the thread requests a service of the kernel, or indirectly as a result of a clock interrupt and subsequent scheduling preemption. Inside the kernel, threads can block in many ways: trying to acquire a reader-writer lock, allocating memory, page fault, or otherwise waiting for some data structure to change state.

Appendix A documents Mach's IPC interface. For further information on Mach's interfaces and implementation, see also [Tevanian 87; Young 89; Black 90b].

4.3.2 Message Receive

The IPC implementation uses continuations in two ways: continuations and stack discarding when blocking to wait for a message, and in addition continuation recognition during cross-address space RPC. The stack discarding was the original motivation for developing a framework for using continuations; it provides a substantial space savings in this case. The IPC implementation supports cross-address space RPC as a special, highly optimized case of message-oriented communication. Stack handoff and continuation recognition were developed to recover the performance that was lost to stack discarding.

The message receive path is an attractive target for continuations and stack discarding because most threads spend most of their time blocked waiting for a message. The performance numbers in the next chapter verify this, but here I would like to consider why this is the case in Mach. (This analysis applies to other communication-oriented microkernel systems.) Message receive operations are prevalent because there are really only a few fundamental reasons for a thread to wait for a long period of time, and these all manifest themselves as waiting for a message:

- Waiting for slow IO, such as a tty read.
- Sleeping until a specified time.
- Waiting in `select`-like operations, for one of several things to happen.
- Waiting in a server for a request from a client.

In the case of Unix APIs, the Unix process waits for a reply message from the Unix emulation subsystem. Emulation subsystems and other native Mach applications use IPC to perform all of the above actions.

The Mach IPC implementation uses two of the techniques explored in the MockIPC example. The message-receive operation uses the minimal-code-impact technique of Section 4.2.3, and the optimized RPC path uses the handoff continuation recognition technique of Section 4.2.5.

4.3.3 Exceptions

The Mach exception-handling interface uses Mach IPC messages to report exceptions to user-level servers. When a thread running in a user address space encounters an exception condition, such as a divide-by-zero or other integer and floating point exception, an invalid instruction, or a reference to an invalid or protected memory address, then the kernel invokes the exception interface.

The exception interface allows an exception port to be associated with each thread and task. If a thread has an exception port, then it is given a chance to handle the exception. If the thread’s exception server fails to handle the exception, or if the thread doesn’t have an exception port, then the kernel tries the task’s exception port. If the task’s exception server fails to handle the exception or the task doesn’t have an exception port, then the kernel terminates the task. If an exception server does handle the exception then the thread continues execution. (Presumably the exception server altered the thread’s register state in the course of handling the exception so that the thread does not immediately retake the same exception.)

This means that a single exception might result in zero, one, or two upcalls to exception servers. To perform an exception upcall, the kernel sends a message to the exception port. The message contains three integers describing the type of exception and two ports representing the thread taking the exception and the thread's task. The thread then waits for a reply message. Eventually the exception server sends a reply message containing a status code that indicates whether it handled the exception.

Several things make exception handling a good candidate for continuations and continuation recognition:

- Exception-handling latency is very important to some operating system emulation environments. For example, a DOS emulation needs fast exception handling because DOS applications access DOS services with "interrupt" instructions, access privileged registers, and use privileged IO instructions, all of which result in control transfers to the DOS emulation subsystem via the exception interface.

- There is very little state that needs to be saved while a thread waits for an exception reply message. The only requirement is that the exception type information (three integers) must be saved across an upcall to the thread's exception port, because this information might be needed again for a subsequent upcall to the task's exception port.

- Continuation recognition has potential in this situation. First, it allows exception handling to participate directly in the fast handoff RPC path, because that path can recognize the special continuation for a thread blocked waiting for an exception reply message. Second, it avoids some time-consuming processing of the exception request message, because it contains references to two ports.

I developed the "structured gotos" technique of Section 4.2.2 while modifying the exception handling code to use continuations. Internally, the kernel has an `exception` function that the machine-dependent trap-handling code calls to initiate exception processing for the current thread. Because `exception` is always called in an exception-handling context, `exception` can always use `thread_exception_return` to resume user-level execution. This predictability makes the "structured gotos" technique appropriate.

Exception handling takes advantage of continuation recognition as follows. The `exception_raise` primitive delivers an exception request message to a particular exception port. Before it even constructs the message, `exception_raise` locks the exception port and attempts a stack handoff to a server thread waiting with the `mach_msg_receive_continue` continuation.

If this succeeds, then `exception_raise` synthesizes the exception request message directly in the external format needed by the exception server and copies the message out to the server's user-space buffer. If continuation recognition fails, `exception_raise` falls back to a slower path that constructs the message in internal format and queues it to the exception port. Continuation recognition bypasses the message parsing and translating that would normally be needed to receive an exception request message. Because exception request messages carry two ports, they qualify as "complex" messages (see Section A.3) that normally require some processing overhead.

The exception path leaves the thread blocked with an `exception_raise_continue` continuation. When the exception server uses `mach_msg` to send its reply message and wait for another request message, then the optimized handoff RPC path in `mach_msg` checks for `exception_raise_continue` (as well as `mach_msg_receive_continue`) and resumes execution of the fast exception-handling path. This use of continuation recognition bypasses the normal `ipc_mqueue_receive` processing that `exception_raise_continue` must normally perform to handle various exceptional conditions, but the request path savings are more significant.

4.3.4 Internal Kernel Threads

The Mach kernel contains a large number of internal "worker" threads—eight or more in some configurations. These threads offer a good example of an opportunity to use continuations in which the marginal benefit, although small, exceeds the marginal cost:

The worker threads, all of which use continuations, are:

- The idle thread. The Mach kernel contains one idle thread per processor; they execute as the lowest priority threads in the system.

- The pageout daemon. The kernel wakes up the pageout daemon when the list of free physical pages gets small. The pageout daemon creates new free pages, by reclaiming clean pages and sending dirty pages out to external pagers to be cleaned [Draves 91].

- The stack-alloc thread. As described in Section 3.3.2.2, the stack-alloc thread allocates new kernel stacks for other threads.

- The io-done thread. Because of locking considerations, device interrupts can not directly send IPC reply messages when a requested device operation finishes. Instead, the device interrupt wakes up the io-done thread and it sends the reply message.

- The net-io thread. The net-io thread handles incoming network packets. It runs the packets through registered packet filters [Mogul *et al.* 87; Yuhara *et al.* 94] and when a packet filter indicates interest, sends packets (wrapped inside asynchronous IPC messages) out to applications.

- The reaper thread. When a thread terminates itself, there are some technical problems with letting the thread deallocate some of its own resources, such as its thread structure. Instead, the thread queues itself and wakes up the reaper thread to perform the final rites.

- The action thread. Again, for technical reasons some processor and processor set operations require the assistance of a third party. The action thread helps shut down processors and reassign processors to processor sets.

- The sched thread. The algorithm that the scheduler uses to adjust thread execution priorities has some undesirable properties. The sched thread runs periodically to examine and “fix” the priorities of runnable threads.

For all of these threads, blocking with a continuation results in a small latency improvement when they wakeup and block, because they can then participate in stack handoffs instead of context-switches. For example, in one common scenario when a disk read finishes, the kernel transfers from the idle thread to the io-done thread to an application thread. Using continuations for these helper threads improves the latency of this path.

For some of these threads, using continuations results in a small space savings because they no longer need dedicated kernel stacks. The idle thread, the pageout daemon, the io-done thread, and the stack-alloc thread do not qualify, because they must be “stack-privileged” (see Section 3.3.2.3) and have reserved kernel stacks, although this doesn’t prevent them from using continuations and stack handoff. For direct or indirect reasons (because they might be needed to create free physical pages), these threads must have a kernel stack available to them or the kernel could deadlock.

These marginal performance improvements came almost for free, with very little coding effort and no long-term software engineering drawbacks. The worker threads all have no state to save when they block waiting for more work, and their “work loop” functions are all special-purpose and not called in any other context. This makes them trivial candidates for the “structured gotos” technique for using continuations.

For example, Figure 4-9 depicts the changes necessary to create the stack-alloc thread from its predecessor. (In previous versions of Mach, threads could be “swapped out” by having their kernel stacks made pageable. The swapin thread reversed this process.) The `thread_block` function blocks the calling thread until another thread performs a wakeup on its wait condition (the argument to `assert_wait`). The only real change was the replacement of an explicit loop with iteration via the continuation supplied to `thread_block`.

Before

```
void swapin_thread() {
    for (;;) {
        lock queue;
        while (queue not empty) {
            dequeue thread;
            unlock queue;
            swapin_stack(thread);
            lock queue;
        }
        assert_wait(&queue);
        unlock queue;
        thread_block();
    }
}
```

After

```
void stack_alloc_thread() {
    stack_privilege(current_thread());
    stack_alloc_thread_continue();
    /*NOTREACHED*/
}
void stack_alloc_thread_continue() {
    lock queue;
    while (queue not empty) {
        dequeue thread;
        unlock queue;
        stack_alloc(thread, thread_continue);
        lock queue;
    }
    assert_wait(&queue);
    unlock queue;
    thread_block(stack_alloc_thread_continue);
    /*NOTREACHED*/
}
```

Figure 4-9: The stack-alloc thread

4.3.5 Scheduling

Blocking operations that originate in the Mach scheduler provide another example where continuations made an incremental performance improvement possible in return for little effort.

The Mach scheduler sometimes blocks a thread and leaves it in a runnable state. This can happen involuntarily, when a thread is preempted because its scheduling quantum expired, or voluntarily, because the thread made a `thread_switch` system call to yield the processor.

In practice, these scheduling operations are not very important to system performance. The scheduling quantum is 100 milliseconds, very large in comparison to any latency improvements from stack handoff. Stack discarding in these cases can save memory, but a machine without much memory probably only has one user and very few simultaneously runnable threads. Most applications do not use the `thread_switch` system call.

However, I decided to implement scheduling preemptions and `thread_switch` with continuations because some conceivable application loads would benefit and using continuations required almost no effort. When the scheduler is entered via a clock interrupt and decides to preempt the current thread, the thread has no relevant kernel-level state that must be preserved. This means that the scheduler can simply use `thread_exception_return` as the thread's continuation. When the scheduler next runs the thread, `thread_exception_return` automatically does the right thing, restoring the user-level register state that the clock interrupt handler saved and hence resuming the thread's execution. This was a one-line change.

Modifying `thread_switch` took only a bit more effort. `thread_switch` can't use `thread_syscall_return` directly as its continuation argument to `thread_block`, because `thread_syscall_return` requires a status code argument. Instead, when `thread_switch` calls `thread_block` to yield the processor, it supplies `thread_switch_continue` as its continuation. This function in turn calls `thread_syscall_return` with a success status code argument.

4.3.6 Thread Halting

When one thread terminates another thread, the victim thread must normally be given an opportunity to put itself into a "clean" state. This operation proved to be a good opportunistic use of continuation recognition.

In the Mach implementation, thread termination is actually just one example of halting another thread at a "clean point." The kernel calls that get and set the user register context of a thread also halt the target thread in this manner, so that they can manipulate a complete snapshot of the target thread's register state. In the case of termination, the victim thread must not be removed from the system while it holds any resources or is otherwise occupied inside the kernel.

Without continuations and continuation recognition, the clean point mechanism involves an explicit synchronization with two context-switches between the threads. The requesting thread asks the victim thread to halt itself, and then context-switches to it. The victim thread executes until it gets to a clean point, such as just before returning to user space, and then marks itself as halted and context-switches back to the requesting thread.

In most cases, this clean point mechanism has only a minor performance impact. The Unix emulation subsystem uses set-context when creating a Unix process, and it uses thread termination when destroying a Unix process. The get-context and set-context kernel calls are also used when delivering Unix signals and when debugging. However, the overhead of the two context-switches for halting at a clean point makes a relatively small contribution to the overall performance of these activities.

Although the performance improvement for most application loads is relatively small, continuation recognition makes it sufficiently easy to eliminate the context-switch overhead of halting at a clean point that this optimization becomes feasible. This situation is also interesting because it is currently the only place in the Mach kernel that uses continuation recognition *without* a stack handoff, the technique described in Section 4.2.6.

The clean point mechanism checks the target thread, to see if it is blocked with one of a few common continuations such as `thread_exception_return` and `mach_msg_receive_-continue`. In these cases, the mechanism bypasses the synchronization and the context-switches that normally put the target thread in a halted state. If the target thread is blocked with `thread_exception_return`, it can be directly marked as halted. If the target thread is blocked with `mach_msg_receive_continue`, then the clean point code calls a cleanup function exported by the IPC module to put the target thread into a halted state and change its continuation to `thread_exception_return`.

4.3.7 VM Faults

In the Mach virtual memory system, threads can block to wait for a physical page to become available or to wait for a physical page to finish a paging transition, such as page-in. These blocking operations, in the context of a user page fault, would appear to be good candidates for continuations because intrinsically very little state must be saved. In fact, the structure of the virtual memory system made it a little unpleasant to use continuations in these cases, but I did it anyway. The motivation for this was *not* the usual performance improvement; it was to prevent pathological behavior in memory-poor environments.

The potential problem is a positive feedback that would tend to push the system into a poor performance regime. When the system has few free physical pages, the kernel initiates paging and threads will block more frequently in the virtual memory system. If these blocking operations did not use continuations, then the kernel would have to allocate more kernel stacks at this point, because the blocking threads would hold on to their kernel stacks instead of handing them off to other threads. This would increase the demand for physical memory precisely at a time when it was in short supply. In the worst possible scenario, this feedback could lead to unstable behavior, with the system oscillating between two semi-stable states: one with a working set that just fits in physical memory, little paging, and fairly good performance, and the other with a working set (augmented with kernel stack pages) that doesn’t fit in physical memory, much paging, and poor performance.

In practice, I never observed anything like this. Nevertheless, I modified the virtual memory system to use continuations to prevent such problems. Because of the structure and complexity of the page-fault handler, the result is a little messy but it was very easy to implement.

The major functions involved in page-fault handling are `vm_fault` and `vm_fault_page`. The `vm_fault_page` helper function does most of the work, including blocking. The two functions contained about 1200 lines of C (including comments), with 12 arguments and 18 local variables between them. Using the techniques of Section 4.2.3 and Section 4.2.4, it took about 100 lines of additional "boilerplate" code, most of it to explicitly save and restore state variables in several places, to add `resume` and `continue` arguments to `vm_fault` and `vm_fault_page`. `vm_fault` uses the "modular" technique so that its callers are not dependent on its implementation, but `vm_fault` does make assumptions about `vm_fault_page`'s implementation. The worst aspect of the implementation is that these blocking operations must save 60 bytes of state, more than will fit in the normal 28 byte save area in the thread structure, so `vm_fault` must allocate and free an "annex" structure to preserve state for `vm_fault_continue`.

4.3.8 External Pager Interactions

The Mach kernel's interactions with external pagers normally resemble RPCs and could take advantage of continuation recognition to reduce latency. I would expect the improvement to be similar to the exception-handling path's improvement—perhaps a factor of two or three over the current system. However, I chose *not* to use continuations in this area. This section describes how external pager interactions could take advantage of continuation recognition and why they currently do not.

The external pager interface really consists of two interfaces: the kernel can send asynchronous messages to an external pager, a normal process running outside the kernel, and the external pager can send asynchronous messages to the kernel. However, in most cases the kernel initiates an RPC-like interaction by sending a "request" message to the external pager and the external pager responds with a "reply" message. As with the exception-handling interface, the messages from the kernel carry a port argument. Some messages in both directions carry out-of-line memory to move pages between the kernel and the external pager.

Continuation recognition could become an interesting optimization for application loads that made heavy use of these RPC-like interactions. Because of the complexity of the messages in the external pager interfaces and the complexity of the virtual memory system, there is much room for optimization. So far, the Mach project has focused on achieving good virtual memory performance by effective caching of data pages in memory, reducing the frequency of external pager interactions. But for some application scenarios, external pager performance would be very relevant.

The interactions most likely to affect performance for some applications are:

`memory_object_data_request—memory_object_data_provided`

The kernel sends `memory_object_data_request` to request pages in a memory object. The external pager responds with `memory_object_data_provided` to deliver the data pages. Optimizing this interaction would improve the performance of normal paging and mapped-file IO.

`memory_object_data_request—memory_object_data_unavailable`

The external pager can also respond with `memory_object_data_unavailable`, to indicate that the memory object does not contain the requested data. For copy-on-write memory operations, the kernel creates temporary memory objects that contain the modified pages; when searching for a page it first checks the temporary object before moving down the “shadow chain” towards the base memory object. The external pager for temporary objects uses `memory_object_data_unavailable` to indicate that a temporary object does not contain a paged-out modified page. This interaction affects the performance of applications that use copy-on-write, sometimes modify copy-on-write pages, and are not entirely memory-resident.

External pagers for normal memory objects can also use `memory_object_data_-unavailable` to indicate that the kernel should zero-fill the requested pages.

`memory_object_data_unlock—memory_object_lock_request`

The kernel sends `memory_object_data_unlock` when a thread attempts to write a page that the external pager has “locked.” The external pager sends `memory_object_-lock_request` to grant write access to a previously “locked” page. This is a common scenario in some distributed shared memory and garbage collection algorithms [Appel & Li 91].

Some other external pager interactions have an RPC-like flavor, but are not as likely to affect performance:

`memory_object_init—memory_object_set_attributes`

The kernel sends `memory_object_init` when a memory object is first mapped into an address space by the kernel. The external pager should respond with a `memory_object_set_attributes` message. This interaction would only be a performance issue for applications that created, mapped, and destroyed memory objects at a high rate. Furthermore, it would be difficult to use continuations in this scenario because the kernel sends `memory_object_init` and then waits for the memory object to change state while deep inside the `vm_map` kernel call. The page-fault handler initiates the other external pager interactions discussed here.

`memory_object_data_request`—`memory_object_data_error`

An external pager can respond to `memory_object_data_request` with a `memory_object_data_error` message, to indicate that the pager can not supply data that it should possess. This interaction is normally quite rare but it might be fall out naturally if the other `memory_object_data_request` interactions were modified to use continuation recognition.

Most external pager interactions do use continuations. As described in the previous section, when a thread waits for a physical page state transition inside a user page fault it blocks with the `vm_fault_continue` continuation. This includes waiting for responses to `memory_object_data_request` and `memory_object_data_unlock` messages. On the other side of the interaction, threads in an external pager wait for messages from the kernel inside the `mach_msg` system call, with `mach_msg_receive_continue` as their continuation.

However, several factors make these control transfer paths challenging targets for continuation recognition.

- First, the complexity of the virtual memory system and the page-fault handling algorithm in particular make it a difficult job. Avoiding wholesale duplication of already very complex and large functions would require a thorough restructuring.

- Second, the external pager interactions only approximate RPC control transfers. When an external pager sends `memory_object_data_provided`, it might wake up zero, one, or more threads. Getting the efficiency of scheduling handoffs in the common case of an interaction between one faulting thread and one external pager thread, while preserving correctness in the general case, would be a challenge. One approach would use the stack handoff-based continuation recognition of Section 4.2.5 in the common case, augmented with the continuation-modifying technique of Section 4.2.6 for the general case.

- Third, the implementation of external pagers would have to change to get the best performance. Currently external pager threads use one `mach_msg` system call to send a message such as `memory_object_data_provided`, and then later make a second `mach_msg` call to wait for further messages from the kernel. These would have to be combined into a single send/receive `mach_msg` to enable a scheduling handoff to a thread waiting in the page-fault handler.

Considering the size of the implementation effort, and the fact that current applications do not need great external pager performance, I decided to leave continuation recognition in the external pager system for future work.

Chapter 5

Performance

In this chapter I examine the effect that continuations have on the performance of Mach 3.0. In terms of space, I 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, I also show that most control transfers use continuation recognition. This reduces the latency of cross-address space communication and user-level exception handling. These performance results lead me to conclude that one can apply continuations to performance-critical paths and get good results with relatively little effort once the underlying control transfer interface has been implemented.

5.1 Experimental Environment

I 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 Chapter 4. Except for MK40's use of continuations, the MK32 and MK40 kernels measured here were identical.[1] Mach 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.

[1]The MK40 kernel also implements optimizations unrelated to continuations. I added the unrelated optimizations to the MK32 kernel reported here to ensure a fair comparison.

All three kernels were measured 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. The write buffer takes at least six cycles to process each write. It has a 16.67Mhz clock and executes one instruction per cycle, barring cache misses and write stalls. The 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. The 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 [Golub *et al.* 90]. I also measured an MS-DOS emulation environment on the Toshiba 5200 [Malan *et al.* 91]. The MS-DOS emulation uses the 80386's virtual-8086 mode. It implements privileged operations and MS-DOS (BIOS) system calls with a user-level exception handler that catches the faults resulting from native-mode operations. The exception handling thread runs in the address space of the emulated MS-DOS program.

5.2 Dynamic Frequency of Continuation Use

The value of continuations depends on the frequency with which they can be used. To determine this, I counted the number of blocking operations that used continuations in three tests run on the Toshiba 5200 running the MK40 kernel. The first test measured a short C compilation benchmark. This test consists of a shell script which compiles nine source files, all small and with few include files.[2] 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™, an interactive space combat simulation with animation and sound. 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 4-3 summarizes the results. (On the DS3100, the frequencies for the compile test and kernel build are similar. The DOS emulation 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 90b], 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

---

[2]This is the same compilation test reported in [Golub *et al.* 1990].

acquisition. MK40 implements these dynamically infrequent operations using the process model; while blocked, the thread retains its kernel stack.

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 5-1: Frequency of Stack Discarding with Continuations

Table 5-2 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.

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 5-2: Frequency of Continuation Recognition and Stack Handoff

5.3 Time Savings Due to Continuations

In this section I show that continuations improve the runtime performance of cross-address space RPCs and exception handling. My RPC test measures the round-trip time for a cross-address space "null" RPC, which sends the shortest possible message (a 24 byte message header) in each direction and executes a minimal amount of user code. My 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 client thread and it does not examine or change the state of the client thread, so the client thread retakes the exception. 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 5-3.

	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 5-3: RPC and Exception Times (in μsecs)

5.3.1 RPC Improvements

The RPC path in MK32 was already highly optimized relative to Mach 2.5 [Draves 90], so there was little room for further improvement. Although it uses one kernel 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 with a scheduling handoff. In contrast, Mach 2.5 queues messages and uses the general scheduling machinery to determine that the receiving thread is the next to run.

MK40, using continuation recognition, permits the client and server threads to share context during the transfer and achieves an additional 14% reduction in latency on the DS3100.

To examine the source of the improvement in RPC latency on the DS3100, I counted instructions, loads, and stores for each component of the total RPC path, as shown in Table 5-4. In this case, the performance gain comes from doing a stack handoff instead of a context-switch. Continuation recognition provides only enough performance benefit to offset the cost of saving and restoring state with a continuation.

Although the MK40 path uses 21% fewer instructions, it is only 14% faster. The reason for this discrepancy is that the R2000’s write buffer limits the performance of the MK40 path; its 212 stores (at 6 cycles per store) must take at least 1272 cycles.

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.[3]

[3]The Toshiba 5200's RPC latency increased slightly in MK40 because of a performance “bug” that has since been fixed. The trap handler on the 5200 saved 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 copied the current thread's state from the stack and copied the new thread's state onto the stack. In isolation, these copies cost approximately 50 μsecs per RPC.

Table 5-4 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.

	MK40			MK32		
	instrs	loads	stores	instrs	loads	stores
request path						
syscall entry	64	7	25	67	8	20
msg copyin	41	6	6	41	6	6
sender	180	50	28	185	47	26
stack handoff	83	22	18			
context switch				250	52	27
receiver	149	53	20	139	46	15
msg copyout	41	6	6	41	6	6
syscall exit	35	21	1	24	11	1
reply path						
syscall entry	64	7	25	67	8	20
msg copyin	41	6	6	41	6	6
sender	164	41	27	173	41	25
stack handoff	83	22	18			
context switch				250	52	27
receiver	105	40	15	96	34	10
msg copyout	41	6	6	41	6	6
syscall exit	35	21	1	24	11	1
user space						
client code	21	3	5	21	3	5
server code	20	4	5	20	4	5
total	1167	315	212	1480	341	206

Table 5-4: RPC Component Costs on the DS3100

5.3.2 Runtime Cost of Continuations

There is a small runtime cost associated with the use of continuations in Mach. As Table 5-4 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 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 5-4 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.

5.3.3 Exception Handling Improvements

As Table 5-3 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. The need for a fast but portable cross-address space RPC mechanism motivated me to develop a general interface for handling control transfer efficiently. Once I had that interface, I was able to apply it easily to the exception handling path. In less than three days of work, I saw a 2-3 fold improvement in the runtime performance of exception handling. This also realized a space savings due to stack discarding. Further, because these optimizations were implemented using machine-independent code, they only had to be done once. My experience with using continuations on other kernel paths has been similar.

5.4 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, a general purpose multi-user server at Carnegie Mellon University 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. (I measured the number of kernel threads and stacks used by sampling a counter at each clock interrupt, every 10 milliseconds.) 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. This thread has been moved outside the kernel in more recent versions of Mach 3.0 [Golub & Draves 91], leaving only one stack in use in the normal case. Rarely, more stacks were used due to the fact that some control transfers do not use continuations (see the bottom row in Table 4-3). In the worst of circumstances, I saw the compile test and MS-DOS emulation use 3 stacks, and the kernel build use 6. (I determined the maximum number of stacks in use by checking at every stack allocation, *not* by sampling at clock interrupts.)

Another way of evaluating the savings due to continuations is to consider the average amount of kernel memory consumed by each thread. Table 5-5 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.

	MK40			MK32		
	min	average	max	min	average	max
MI state	484	484	484	452	452	452
MD state	172	206	308	0	0	0
stack	0	0	0	336	4022	4432
total	656	690	792	788	4474	4884

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

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 a 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 kernel. The number of registers saved depends on whether or not a thread is using the floating point unit. On average, I have found that only about 1 in 4 threads uses floating point. 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 (4K bytes), 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. Even when kernel stacks are pageable, threads run often enough that their stacks remain in memory. With MK32, for example, I 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. 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.

Chapter 6

Conclusions

Control transfer is the fundamental activity in operating system kernels. Using a programming language abstraction, continuations, I have developed a framework for kernel control transfer that achieves increased flexibility and performance. An implementation in the context of the Mach 3.0 operating system from Carnegie Mellon University and performance measurements in that environment validate my framework. The resulting system has been successfully ported to more than ten different hardware architectures and is already in commercial use.

6.1 Contributions

The main contributions of this work are:

- An adaptation of continuations, a programming language abstraction, for use in a hostile environment, operating systems kernels written in conventional programming languages.

- A generalization of previous operating system control transfer optimizations.

- An interface for managing control transfer in the kernel, which provides significantly more functionality than existing such interfaces while still being portable and efficient.

- A methodology for using continuations, including converting existing code.

In this dissertation, I have shown how a programming language abstraction for control transfer can be adapted for use with conventional programming languages. Continuations, which represent the saved state of a suspended computation or thread, were originally developed as a mathematical abstraction for defining programming language control transfer semantics. Some modern programming languages support first-class continuations, which provide direct access to this abstraction. Unfortunately, the severe efficiency and concurrency requirements for operating system kernels have so far precluded the use of these modern programming languages; operating system kernels today are still written in systems programming languages such as C and C++. Operating in this conventional environment, my framework gives programmers a choice of continuation representations: when a thread blocks, the programmer can choose to represent the thread's state with saved register context and stack frames, or the programmer can choose to save a continuation function and any state variables that are deemed important. The flexibility of having a choice of thread state representations and the accessible nature of the machine-independent continuation function representation permits common control transfer situations to be optimized.

Continuations generalize many existing operating system control transfer optimizations. This should not be surprising, given that continuations were invented expressly as a generalization for all control transfer operations. To review several examples, recall LRPC [Bershad *et al.* 90], the V system [Cheriton 88], and Taos [McJones & Swart 89]. LRPC optimizes the control transfer in cross-address space communication; it provides a very direct path from client to server and back that eliminates unnecessary saving and restoring of register context. Continuations achieve this optimization in cross-address communication, but in a more general fashion that allows the same optimization to operate in other control transfer paths. The V system uses a single kernel stack per processor, but at the expense of disallowing context-switches inside the V kernel. Continuations achieve the performance characteristics of a single kernel stack per processor, but also allow traditional context-switches. This greatly simplifies the implementation of kernel facilities such as dynamic memory allocation, multiprocessor synchronization, and virtual memory. The Taos kernel optimizes kernel stack usage by discarding the kernel stack of threads that block in user mode, without any kernel context. Continuations achieve this thread management optimization, but also allow kernel stacks to be discarded in other situations when a continuation function can represent the kernel context.

In support of continuations, I have developed an interface for the machine-independent management of control transfer inside the kernel. The conventional interface for control transfer is a "ContextSwitch" primitive. In contrast, my interface provides the following functionality:

- Control of kernel stack allocation and deallocation.

- The ability to detach a kernel stack from a thread and later reattach a different stack.

- A stack-handoff primitive that changes the current thread without saving and restoring register context.
- A context-switch primitive that avoids multiprocessor synchronization problems.

In addition, I have developed a set of techniques for using continuations effectively. This includes advice for selecting the important control transfer paths, depending on the performance goals, and for selecting the appropriate coding techniques, depending on the structure of those control transfer paths. The advice and techniques apply both to converting existing kernel code to use continuations and writing new code with continuations.

Taken together, these contributions form a blueprint for putting continuations into practice.

6.2 Future Impact

The work described in this dissertation relates to several current trends in software and hardware evolution:

- Microkernels and other forms of modularized operating systems.
- Portable and consumer devices.
- High-latency memory hierarchies in high-performance computing.

Finding an appropriate balance between flexibility and performance is a continual challenge. Recent systems software, both academic and commercial, is reaching for more flexibility by moving system components into separate modules, which are often loaded in separate address spaces. For example, Windows NT [Custer 93] and OSF/1 MK [Roy 93] have moved their application programmer interfaces (Win32 and Unix, respectively) into address spaces outside the kernel. The X Window System [Scheifler & Gettys 90] places the display manager and the window manager functionality in their own address spaces. This trend increases the importance of control transfer performance.

Despite the exponential decrease in the cost of memory, memory capacity remains a concern in some increasingly important environments. With small portable computers, power consumption and packaging issues limit memory size. With consumer devices, such as cable set-top boxes, high-definition TVs, and personal digital assistants, the cost of goods must be kept as low as possible.

In some current designs memory represents a significant fraction of the total cost of goods.[1] This trend increases the importance of memory overhead for control transfer.

High-performance computing environments do not suffer from limited memory capacity. However, they must cope with deep memory hierarchies with correspondingly large latencies for accesses that must descend the hierarchy. This trend increases the importance of memory access locality in control transfer paths.

This dissertation addresses the issues of performance, memory overhead, and memory access locality in control transfer. The stack-handoff and continuation recognition optimizations improve the performance of important control transfer paths. The stack-discarding optimization reduces the memory overhead of control transfer; in most situations the kernel operates with one kernel stack per processor. This also improves memory access locality in the important control transfer paths, because these paths only reference one kernel stack instead of two.

6.3 Final Remarks

This dissertation has shown how continuations, a programming language abstraction for control transfer, can be adapted for use with conventional programming languages to achieve increased flexibility and performance in kernel control transfer. A programming language with support for first-class continuations would offer even greater flexibility. Supporting first-class continuations in a kernel environment, with its requirements for efficiency and concurrency, remains an interesting challenge. In addition, current formulations of first-class continuations do not allow continuation recognition, because first-class continuations are opaque objects. I hope that future research will make it possible to use first-class continuations in a kernel environment.

[1] For example, if a consumer device with a total cost of goods of $200 contains 4 megabytes of memory at $20/megabyte, then memory represents 40% of the total cost.

Appendix A

Mach IPC

Mach IPC plays multiple roles in this dissertation: motivation that originally prompted the work, example in the guise of MockIPC, and finally subject of measurement. This appendix describes the Mach 3.0 IPC system call interface. The interface allows programs to send messages to protected message queues known as ports. The messages can carry references to ports and regions of memory in addition to uninterpreted data.

In developing the Mach 3.0 IPC interface, I had two major goals [Draves 90]. First, I intended to fix a number of semantic problems. The original Mach interface did not give applications the tools needed to manage correctly their handles for ports. Second, I intended to develop a more efficient implementation, both in terms of latency for common operations and data structure size, and some aspects of the new interface assisted these performance goals. I also chose to preserve as much as possible the spirit of the Mach 2.5 interface and provided support for the old interface in the implementation.

The following features of the Mach 3.0 IPC interface relate to this dissertation:

- The `mach_msg` system call allows both the client and server sides of RPC-like message exchanges to wakeup another thread and block themselves with a single system call. This enables scheduling handoffs and the stack handoff form of continuation recognition in both the request and reply directions. See Section 4.3.2.

- Mach messages have internal structure that the kernel may need to interpret or parse. When the kernel sends a message with complex internal structure, continuation recognition can bypass the overhead of parsing a message that was just constructed. This occurs when the kernel sends exception messages; see Section 4.3.3. In a similar way, continuation recognition bypasses some of the overhead of supporting the more esoteric and little-used options to `mach_msg`.

The following sections discuss the IPC interface in detail.

A.1 Major Concepts

The Mach kernel provides message-oriented, capability-based interprocess communication. The interprocess communication (IPC) primitives efficiently support many different styles of interaction, including remote procedure calls, object-oriented distributed programming, streaming of data, and sending very large amounts of data in a single message.

The IPC primitives operate on three abstractions: messages, ports, and port sets. User tasks[1] access all other kernel services and abstractions via the IPC primitives.

The message primitives let tasks send and receive messages. Tasks send messages to ports. Messages sent to a port are delivered reliably (messages may not be lost) and are received in the order in which they were sent. Messages contain a fixed-size header and a variable amount of typed data following the header. The header describes the destination and size of the message.

The IPC implementation makes use of the virtual memory system to transfer efficiently large amounts of data. The message body can contain the address of a region in the sender's address space that should be transferred as part of the message. When a task receives a message containing such an "out-of-line" region of data, the data appears in an unused portion of the receiver's address space. This transmission of out-of-line data is optimized so that sender and receiver share the physical pages copy-on-write, and no actual data copy occurs unless the pages are written. Regions of memory up to the size of a full address space may be sent in this manner.

Ports hold a queue of messages. Tasks operate on a port to send and receive messages by exercising capabilities for the port. Multiple tasks can hold send capabilities, or rights, for a single port. Tasks can also hold send-once rights, which grant the ability to send a single message. Only one task can hold the receive capability, or receive right, for a port. Port rights can be transferred between tasks via messages. The sender of a message can specify in the message body via a special type specification that the message contains a port right. If a message contains a receive right for a port, then the receive right is removed from the sender of the message and the right is transferred to

[1]Mach terminology for processes.

the receiver of the message. While the receive right is in transit, tasks holding send rights can still send messages to the port, and they are queued until a task acquires the receive right and uses it to receive the messages.

Tasks can receive messages from port sets as well as ports. The port set abstraction allows a single thread to wait for a message from any of several ports. Tasks manipulate port sets with a capability, or port-set right, that is taken from the same name space as the port capabilities. The port-set right may not be transferred in a message. A port set holds receive rights, and a receive operation on a port set blocks waiting for a message sent to any of the constituent ports. A port may not belong to more than one port set, and if a port is a member of a port set, the holder of the receive right can’t receive directly from the port.

Port rights are a secure, location-independent way of naming ports. The port queue is a protected data structure, only accessible via the kernel’s exported message primitives. Rights are also protected by the kernel; there is no way for a malicious user task to guess a port name and send a message to a port to which it shouldn’t have access. Port rights do not carry any location information. When a receive right for a port moves from task to task, and even between tasks on different machines, the send rights for the port remain unchanged and continue to function.

A.2 Sending and Receiving Messages: mach_msg

The `mach_msg` system call sends and receives Mach messages. Mach messages contain typed data, which can include port rights and references to large regions of memory.

```
mach_msg_return_t
mach_msg(msg, option, send_size, rcv_size, rcv_name,
         timeout, notify)
   mach_msg_header_t *msg;
   mach_msg_option_t option;
   mach_msg_size_t send_size;
   mach_msg_size_t rcv_size;
   mach_port_t rcv_name;
   mach_msg_timeout_t timeout;
   mach_port_t notify;
```

If the `option` argument is `MACH_SEND_MSG`, it sends a message. The `send_size` argument specifies the size of the message to send. The `msgh_remote_port` field of the message header specifies the destination of the message.

If the `option` argument is `MACH_RCV_MSG`, it receives a message. The `rcv_size` argument specifies the size of the message buffer that will receive the message; messages larger than `rcv_size` are not received. (By default they are destroyed.) The `rcv_name` argument specifies the port or port set from which to receive.

If the `option` argument is `MACH_SEND_MSG|MACH_RCV_MSG`, then `mach_msg` does both send and receive operations. If the send operation encounters an error (any return code other than `MACH_MSG_SUCCESS`), then the call returns immediately without attempting the receive operation. Semantically the combined call is equivalent to separate send and receive calls, but it saves a system call and enables other internal optimizations.

If the `option` argument specifies neither `MACH_SEND_MSG` nor `MACH_RCV_MSG`, then `mach_msg` does nothing.

Some options, like `MACH_SEND_TIMEOUT` and `MACH_RCV_TIMEOUT`, share a supporting argument. If these options are used together, they make independent use of the supporting argument's value.

The arguments to `mach_msg` are:

`msg`
: The address of a message buffer in the caller's address space. Message buffers should be aligned on integer boundaries.

`option`
: Message options are bit values, combined with bitwise-or. One or both of `MACH_SEND_MSG` and `MACH_RCV_MSG` should be used. Other options act as modifiers.

`send_size`
: When sending a message, specifies the size of the message buffer. Otherwise zero should be supplied.

`rcv_size`
: When receiving a message, specifies the size of the message buffer. Otherwise zero should be supplied.

`rcv_name`
: When receiving a message, specifies the port or port set. Otherwise `MACH_PORT_NULL` should be supplied.

`timeout`
: When using the `MACH_SEND_TIMEOUT` and `MACH_RCV_TIMEOUT` options, specifies the time in milliseconds to wait before giving up. Otherwise `MACH_MSG_TIMEOUT_NONE` should be supplied.

`notify`

When using the `MACH_SEND_NOTIFY`, `MACH_SEND_CANCEL`, and `MACH_RCV_NOTIFY` options, specifies the port used for the notification. Otherwise `MACH_PORT_NULL` should be supplied.

A.3 Message Format

A Mach message consists of a fixed-size message header, a `mach_msg_header_t`, followed by zero or more data items. Data items are typed. Each item has a type descriptor followed by the actual data (or the address of the data, for out-of-line memory regions).

```
typedef unsigned int mach_port_t;

typedef unsigned int mach_port_seqno_t;

typedef unsigned int mach_msg_bits_t;
typedef unsigned int mach_msg_size_t;
typedef int mach_msg_id_t;

typedef struct {
   mach_msg_bits_t   msgh_bits;
   mach_msg_size_t   msgh_size;
   mach_port_t       msgh_remote_port;
   mach_port_t       msgh_local_port;
   mach_port_seqno_t msgh_seqno;
   mach_msg_id_t     msgh_id;
} mach_msg_header_t;
```

The `msgh_size` field in the header of a received message contains the message's size. The message size, a byte quantity, includes the message header, type descriptors, and in-line data. For out-of-line memory regions, the message size includes the size of the in-line address, not the size of the actual memory region. There are no arbitrary limits on the size of a Mach message, the number of data items in a message, or the size of the data items.

The `msgh_remote_port` field specifies the destination port of the message. The field must carry a legitimate send or send-once right for a port.

The `msgh_local_port` field specifies an auxiliary port right, which is conventionally used as a reply port by the recipient of the message. The field must carry a send right, a send-once right, `MACH_PORT_NULL`, or `MACH_PORT_DEAD`.

The `msgh_bits` field has the following bits defined:

```
#define MACH_MSGH_BITS_REMOTE_MASK  0x000000ff
#define MACH_MSGH_BITS_LOCAL_MASK   0x0000ff00
#define MACH_MSGH_BITS_COMPLEX      0x80000000

#define MACH_MSGH_BITS_REMOTE(bits)
#define MACH_MSGH_BITS_LOCAL(bits)
#define MACH_MSGH_BITS(remote, local)
```

The remote and local bits encode `mach_msg_type_name_t` values that specify the port rights in the `msgh_remote_port` and `msgh_local_port` fields. The remote value must specify a send or send-once right for the destination of the message. If the local value doesn't specify a send or send-once right for the message's reply port, it must be zero and `msgh_local_port` must be `MACH_PORT_NULL`. The complex bit must be specified if the message body contains port rights or out-of-line memory regions. If it is not specified, then the message body carries no port rights or memory, no matter what the type descriptors may seem to indicate.

The `MACH_MSGH_BITS_REMOTE` and `MACH_MSGH_BITS_LOCAL` macros return the appropriate `mach_msg_type_name_t` values, given a `msgh_bits` value. The `MACH_MSGH_BITS` macro constructs a value for `msgh_bits`, given two `mach_msg_type_name_t` values.

The `msgh_seqno` field provides a sequence number for the message. It is only valid in received messages; its value in sent messages is overwritten. Section A.7 discusses message sequence numbers.

The `mach_msg` call doesn't use the `msgh_id` field, but it conventionally conveys an operation or function id.

Each data item has a type descriptor, a `mach_msg_type_t` or a `mach_msg_type_long_t`. The `mach_msg_type_long_t` type descriptor allows larger values for some fields. The `msgtl_header` field in the long descriptor is only used for its inline, longform, and deallocate bits.

```
typedef unsigned int mach_msg_type_name_t;
typedef unsigned int mach_msg_type_size_t;
typedef unsigned int mach_msg_type_number_t;

typedef struct {
   unsigned int
      msgt_name : 8,
      msgt_size : 8,
      msgt_number : 12,
      msgt_inline : 1,
      msgt_longform : 1,
      msgt_deallocate : 1,
      msgt_unused : 1;
} mach_msg_type_t;

typedef struct {
   mach_msg_type_t   msgtl_header;
   unsigned short msgtl_name;
   unsigned short msgtl_size;
   unsigned int   msgtl_number;
} mach_msg_type_long_t;
```

The msgt_name (msgtl_name) field specifies the data's type. The following types are predefined:

```
MACH_MSG_TYPE_UNSTRUCTURED
MACH_MSG_TYPE_BIT
MACH_MSG_TYPE_BOOLEAN
MACH_MSG_TYPE_INTEGER_16
MACH_MSG_TYPE_INTEGER_32
MACH_MSG_TYPE_CHAR
MACH_MSG_TYPE_BYTE
MACH_MSG_TYPE_INTEGER_8
MACH_MSG_TYPE_REAL
MACH_MSG_TYPE_STRING
MACH_MSG_TYPE_STRING_C
MACH_MSG_TYPE_PORT_NAME

MACH_MSG_TYPE_MOVE_RECEIVE
MACH_MSG_TYPE_MOVE_SEND
MACH_MSG_TYPE_MOVE_SEND_ONCE
MACH_MSG_TYPE_COPY_SEND
MACH_MSG_TYPE_MAKE_SEND
MACH_MSG_TYPE_MAKE_SEND_ONCE
```

The last six types specify port rights, and receive special treatment. The next section discusses these types in detail. The type MACH_MSG_TYPE_PORT_NAME describes port right names, for use when no rights are being transferred, but just names. For this purpose, it should be used in preference to MACH_MSG_TYPE_INTEGER_32.

The msgt_size (msgtl_size) field specifies the size of each datum, in bits. For example, the msgt_size of MACH_MSG_TYPE_INTEGER_32 data is 32.

The msgt_number (msgtl_number) field specifies how many data elements comprise the data item. Zero is a legitimate number.

The total length specified by a type descriptor is (msgt_size * msgt_number), rounded up to an integral number of bytes. In-line data is then padded to an integral number of integers. This ensures that type descriptors always start on integer boundaries. It implies that message sizes are always an integral multiple of an integer's size.

The msgt_longform bit specifies, when TRUE, that this type descriptor is a mach_msg_type_long_t instead of a mach_msg_type_t. The msgt_name, msgt_size, and msgt_number fields should be zero. Instead, mach_msg uses the following msgtl_name, msgtl_size, and msgtl_number fields.

The msgt_inline bit specifies, when FALSE, that the data actually resides in an out-of-line region. The address of the memory region (a vm_offset_t) follows the type descriptor in the message body. The `msgt_name`, `msgt_size`, and `msgt_number` fields describe the memory region, not the address.

The `msgt_deallocate` bit is used with out-of-line regions. When `TRUE`, it specifies that the memory region should be deallocated from the sender's address space (as if with `vm_deallocate`) when the message is sent.

The `msgt_unused` bit should be zero.

A.4 Port Rights

Each task has its own name space of port rights. Port rights are named with unsigned integers. Except for the reserved values `MACH_PORT_NULL` (0) and `MACH_PORT_DEAD` (-1), this is a full 32-bit name space. When the kernel chooses a name for a new right, it is free to pick any unused name (one which denotes no right) in the space.

There are five basic kinds of rights: receive rights, send rights, send-once rights, port-set rights, and dead names. Dead names are not capabilities. They act as place-holders to prevent a name from being otherwise used.

A port is destroyed, or dies, when its receive right is deallocated. When a port dies, send and send-once rights for the port turn into dead names. Any messages queued at the port are destroyed, which deallocates the port rights and out-of-line memory in the messages.

Tasks may hold multiple "user-references" for send rights and dead names. When a task receives a send right which it already holds, the kernel increments the right's user-reference count. When a task deallocates a send right, the kernel decrements its user-reference count, and the task only loses the send right when the count goes to zero.

Send-once rights always have a user-reference count of one, although a port can have multiple send-once rights, because each send-once right held by a task has a different name. In contrast, when a task holds send rights or a receive right for a port, the rights share a single name.

A message body can carry port rights; the `msgt_name` (`msgtl_name`) field in a type descriptor specifies the type of port right and how the port right is to be extracted from the caller. The values `MACH_PORT_NULL` and `MACH_PORT_DEAD` are always valid in place of a port right in a message body. In a sent message, the following `msgt_name` values denote port rights:

`MACH_MSG_TYPE_MAKE_SEND`

The message will carry a send right, but the caller must supply a receive right. The send right is created from the receive right, and the receive right's make-send count is incremented.

`MACH_MSG_TYPE_COPY_SEND`

The message will carry a send right, and the caller should supply a send right. The user reference count for the supplied send right is not changed. The caller may also supply a dead name and the receiving task will get `MACH_PORT_DEAD`.

`MACH_MSG_TYPE_MOVE_SEND`

The message will carry a send right, and the caller should supply a send right. The user reference count for the supplied send right is decremented, and the right is destroyed if the count becomes zero. Unless a receive right remains, the name becomes available for recycling. The caller may also supply a dead name, which loses a user reference, and the receiving task will get `MACH_PORT_DEAD`.

`MACH_MSG_TYPE_MAKE_SEND_ONCE`

The message will carry a send-once right, but the caller must supply a receive right. The send-once right is created from the receive right.

`MACH_MSG_TYPE_MOVE_SEND_ONCE`

The message will carry a send-once right, and the caller should supply a send-once right. The caller loses the supplied send-once right. The caller may also supply a dead name, which loses a user reference, and the receiving task will get `MACH_PORT_DEAD`.

`MACH_MSG_TYPE_MOVE_RECEIVE`

The message will carry a receive right, and the caller should supply a receive right. The caller loses the supplied receive right, but retains any send rights with the same name.

If a message carries a send or send-once right, and the port dies while the message is in transit, then the receiving task will get `MACH_PORT_DEAD` instead of a right. The following `msgt_name` values in a received message indicate that it carries port rights:

`MACH_MSG_TYPE_PORT_SEND`

This is actually the same value as `MACH_MSG_TYPE_MOVE_SEND`. The message carried a send right. If the receiving task already has send and/or receive rights for the port, then that name for the port will be reused. Otherwise, the new right will have a new name. If the task already has send rights, it gains a user reference for the right (unless this would cause the user-reference count to overflow). Otherwise, it acquires the send right, with a user-reference count of one.

`MACH_MSG_TYPE_PORT_SEND_ONCE`

This is actually the same value as `MACH_MSG_TYPE_MOVE_SEND_ONCE`. The message carried a send-once right. The right will have a new name.

`MACH_MSG_TYPE_PORT_RECEIVE`

This is actually the same value as `MACH_MSG_TYPE_MOVE_RECEIVE`. The message carried a receive right. If the receiving task already has send rights for the port, then that name for the port will be reused. Otherwise, the right will have a new name. The make-send count of the receive right is reset to zero, but the port retains other attributes like queued messages, extant send and send-once rights, and requests for port-destroyed and no-senders notifications.

When the kernel chooses a new name for a port right, it can choose any name, other than `MACH_PORT_NULL` and `MACH_PORT_DEAD`, that is not currently being used for a port right or dead name. It might choose a name that at some previous time denoted a port right, but is currently unused.

A.5 Memory

A message body can contain the address of a region in the sender's address space that should be transferred as part of the message. The message carries a logical copy of the memory, but the kernel uses virtual memory techniques to defer any actual page copies. Unless the sender or the receiver modifies the data, the physical pages remain shared.

An out-of-line transfer occurs when the data's type descriptor specifies `msgt_inline` as `FALSE`. The address of the memory region (a `vm_offset_t`) should follow the type descriptor in the message body. The type descriptor and the address contribute to the message's size (`send_size`, `msgh_size`, `rcv_size`). The out-of-line region itself does not contribute to the message's size.

The name, size, and number fields in the type descriptor describe the type and length of the out-of-line data, not the in-line address. Out-of-line memory frequently requires long type descriptors (`mach_msg_type_long_t`), because the `msgt_number` field is too small to describe a page of 4K bytes.

Out-of-line memory arrives somewhere in the receiver's address space as new memory. It has the same inheritance and protection attributes as newly `vm_allocate`'d memory. The receiver has the responsibility of deallocating (with `vm_deallocate`) the memory when it is no longer needed. Security-conscious receivers should exercise caution when using out-of-line memory from untrustworthy sources, because the memory may be backed by an unreliable external pager.

Null out-of-line memory is legal. If the out-of-line region size is zero (for example, because `msgtl_number` is zero), then the region's specified address is ignored. A received null out-of-line memory region always has a zero address.

Unaligned addresses and region sizes that are not page multiples are legal. A received message can also contain memory with unaligned addresses and unusual sizes. In the general case, the first and last pages in the new memory region in the receiver do not contain only data from the sender, but are partly zero. (But see Section A.9.) The received address points to the start of the data in the first page. This possibility doesn't complicate deallocation, because `vm_deallocate` does the right thing, rounding the start address down and the end address up to deallocate all arrived pages.

Out-of-line memory has a deallocate option, controlled by the `msgt_deallocate` bit. If it is `TRUE` and the out-of-line memory region is not null, then the region is implicitly deallocated from the sender, as if by `vm_deallocate`. In particular, the start and end addresses are rounded so that every page overlapped by the memory region is deallocated. The use of `msgt_deallocate` effectively changes the memory copy into a memory movement. In a received message, `msgt_deallocate` is `TRUE` in type descriptors for out-of-line memory.

Out-of-line memory can carry port rights.

A.6 Message Send

The send operation queues a message to a port. The message carries a copy of the caller's data. After the send, the caller can freely modify the message buffer or the out-of-line memory regions and the message contents will remain unchanged.

Message delivery is reliable and sequenced. Messages are not lost, and messages sent to a port, from a single thread, are received in the order in which they were sent.

If the destination port's queue is full, then several things can happen. If the message is sent to a send-once right (`msgh_remote_port` carries a send-once right), then the kernel ignores the queue limit and delivers the message. Otherwise the caller blocks until there is room in the queue, unless the `MACH_SEND_TIMEOUT` or `MACH_SEND_NOTIFY` options are used. If a port has several blocked senders, then any of them may queue the next message when space in the queue becomes available, with the proviso that a blocked sender will not be indefinitely starved.

These options modify `MACH_SEND_MSG`. If `MACH_SEND_MSG` is not also specified, they are ignored.

`MACH_SEND_TIMEOUT`
: The `timeout` argument should specify a maximum time (in milliseconds) for the call to block before giving up. If the message can't be queued before the timeout interval elapses, then the call returns `MACH_SEND_TIMED_OUT`. A zero timeout is legitimate.

`MACH_SEND_NOTIFY`
: The `notify` argument should specify a receive right for a notify port. If the send were to block, then instead the message is queued, `MACH_SEND_WILL_NOTIFY` is returned, and a msg-accepted notification is requested. If `MACH_SEND_TIMEOUT` is also specified, then `MACH_SEND_NOTIFY` doesn't take effect until the timeout interval elapses.

  With `MACH_SEND_NOTIFY`, a task can forcibly queue to a send right one message at a time. A msg-accepted notification is sent to the notify port when another message can be forcibly queued. If an attempt is made to use `MACH_SEND_NOTIFY` before then, the call returns a `MACH_SEND_NOTIFY_IN_PROGRESS` error.

  The msg-accepted notification carries the name of the send right. If the send right is deallocated before the msg-accepted notification is generated, then the msg-accepted notification carries the value `MACH_PORT_NULL`. If the destination port is destroyed before the notification is generated, then a send-once notification is generated instead.

`MACH_SEND_INTERRUPT`
: If specified, the `mach_msg` call will return `MACH_SEND_INTERRUPTED` if a software interrupt aborts the call. Otherwise, the send operation will be retried.

`MACH_SEND_CANCEL`
: The notify argument should specify a receive right for a notify port. If the send operation removes the destination port right from the caller, and the removed right had a dead-name request registered for it, and notify is the notify port for the dead-name request, then the dead-name request may be silently canceled (instead of resulting in a port-deleted

notification). This option is typically used to cancel a dead-name request made with the `MACH_RCV_NOTIFY` option. It should only be used as an optimization.

The send operation can generate the following return codes.

These return codes imply that the call did nothing:

`MACH_SEND_MSG_TOO_SMALL`
: The specified `send_size` was smaller than the minimum size for a message.

`MACH_SEND_NO_BUFFER`
: A resource shortage prevented the kernel from allocating a message buffer.

`MACH_SEND_INVALID_DATA`
: The supplied message buffer was not readable.

`MACH_SEND_INVALID_HEADER`
: The `msgh_bits` value was invalid.

`MACH_SEND_INVALID_DEST`
: The `msgh_remote_port` value was invalid.

`MACH_SEND_INVALID_REPLY`
: The `msgh_local_port` value was invalid.

`MACH_SEND_INVALID_NOTIFY`
: When using `MACH_SEND_CANCEL`, the notify argument did not denote a valid receive right.

These return codes imply that some or all of the message was destroyed:

`MACH_SEND_INVALID_MEMORY`
: The message body specified out-of-line data that was not readable.

`MACH_SEND_INVALID_RIGHT`
: The message body specified a port right which the caller didn’t possess.

`MACH_SEND_INVALID_TYPE`
: A type descriptor was invalid.

`MACH_SEND_MSG_TOO_SMALL`
: The last data item in the message ran over the end of the message.

These return codes imply that the message was returned to the caller with a pseudo-receive operation:

`MACH_SEND_TIMED_OUT`
: The timeout interval expired.

`MACH_SEND_INTERRUPTED`
: A software interrupt occurred.

`MACH_SEND_INVALID_NOTIFY`
: When using `MACH_SEND_NOTIFY`, the notify argument did not denote a valid receive right.

`MACH_SEND_NO_NOTIFY`
: A resource shortage prevented the kernel from setting up a msg-accepted notification.

`MACH_SEND_NOTIFY_IN_PROGRESS`
: A msg-accepted notification was already requested, and hasn't yet been generated.

These return codes imply that the message was queued:

`MACH_SEND_WILL_NOTIFY`
: The message was forcibly queued, and a msg-accepted notification was requested.

`MACH_MSG_SUCCESS`
: The message was queued.

Some return codes, like `MACH_SEND_TIMED_OUT`, imply that the message was almost sent, but could not be queued. In these situations, the kernel tries to return the message contents to the caller with a pseudo-receive operation. This prevents the loss of port rights or memory that only exist in the message. For example, a receive right that was moved into the message, or out-of-line memory sent with the deallocate bit.

The pseudo-receive operation is very similar to a normal receive operation. The pseudo-receive handles the port rights in the message header as if they were in the message body. They are not swapped. After the pseudo-receive, the message is ready to be resent. If the message is not resent, note that out-of-line memory regions may have moved and some port rights may have changed names.

The pseudo-receive operation may encounter resource shortages. This is similar to a `MACH_RCV_BODY_ERROR` return code from a receive operation. When this happens, the normal send return codes are augmented with the `MACH_MSG_IPC_SPACE`, `MACH_MSG_VM_SPACE`, `MACH_MSG_IPC_KERNEL`, and `MACH_MSG_VM_KERNEL` bits to indicate the nature of the resource shortage.

The queuing of a message carrying receive rights may create a circular loop of receive rights and messages, which can never be received. For example, a message carrying a receive right can be sent to that receive right. This situation is not an error, but the kernel will garbage-collect such loops, destroying the messages and ports involved.

A.7 Message Receive

The receive operation dequeues a message from a port. The receiving task acquires the port rights and out-of-line memory regions carried in the message.

The `rcv_name` argument specifies a port or port set from which to receive. If a port is specified, the caller must possess the receive right for the port and the port must not be a member of a port set. If no message is present, then the call blocks, subject to the `MACH_RCV_TIMEOUT` option.

If a port set is specified, the call will receive a message sent to any of the constituent ports. It is permissible for the port set to have no constituent ports, and ports may be added and removed while a receive from the port set is in progress. The received message can come from any of the constituent ports that have messages, with the proviso that a constituent port with messages will not be indefinitely starved. The `msgh_local_port` field in the received message header specifies from which port in the port set the message came.

The `rcv_size` argument specifies the size of the caller's message buffer. The `mach_msg` call will not receive a message larger than `rcv_size`. Messages that are too large are destroyed, unless the `MACH_RCV_LARGE` option is used.

The destination and reply ports are reversed in a received message header. The `msgh_local_port` field names the destination port, from which the message was received, and the `msgh_remote_port` field names the reply port right. The bits in `msgh_bits` are also reversed. The `MACH_MSGH_BITS_LOCAL` bits have the value `MACH_MSG_TYPE_PORT_SEND` if the message was sent to a send right, and the value MACH_MSG_TYPE_PORT_SEND_ONCE if it was sent to a send-once right. The MACH_MSGH_BITS_REMOTE bits describe the reply port right.

A received message can contain port rights and out-of-line memory. The msgh_local_port field does not "receive" a port right although it does name the destination port; the act of receiving the message destroys the send or send-once right for the destination port. The msgh_remote_port field does potentially carry a received port right, the reply port right, and the message body can carry port rights and memory if MACH_MSGH_BITS_COMPLEX is present in msgh_bits. Received port rights and memory should be consumed or deallocated in some fashion.

In almost all cases, msgh_local_port will specify the name of a receive right, either rcv_name or if rcv_name is a port set, a constituent of rcv_name. If other threads are concurrently manipulating the receive right, the situation is more complicated. If the receive right is renamed during the call, then msgh_local_port specifies the right's new name. If the caller loses the receive right after the message was dequeued from it, then mach_msg will proceed instead of returning MACH_RCV_PORT_DIED. If the receive right was destroyed, then msgh_local_port specifies MACH_PORT_DEAD. If the receive right still exists, but isn't held by the caller, then msgh_local_port specifies MACH_PORT_NULL.

Received messages are stamped with a sequence number, taken from the port from which the message was received. (Messages received from a port set are stamped with a sequence number from the appropriate constituent port.) Newly created ports start with a zero sequence number, and the sequence number is reset to zero whenever the port's receive right moves between tasks. When a message is dequeued from the port, it is stamped with the port's sequence number and the port's sequence number is then incremented. The dequeue and increment operations are atomic, so that multiple threads receiving messages from a port can use the msgh_seqno field to reconstruct the original order of the messages.

These options modify MACH_RCV_MSG. If MACH_RCV_MSG is not also specified, they are ignored.

MACH_RCV_TIMEOUT

The timeout argument should specify a maximum time (in milliseconds) for the call to block before giving up. If no message arrives before the timeout interval elapses, then the call returns MACH_RCV_TIMED_OUT. A zero timeout is legitimate.

MACH_RCV_NOTIFY

The notify argument should specify a receive right for a notify port. If receiving the reply port creates a new port right in the caller, then the notify port is used to request a dead-name notification for the new port right.

`MACH_RCV_INTERRUPT`
: If specified, the `mach_msg` call will return `MACH_RCV_INTERRUPTED` if a software interrupt aborts the call. Otherwise, the receive operation will be retried.

`MACH_RCV_LARGE`
: If the message is larger than `rcv_size`, then the message remains queued instead of being destroyed. The call returns `MACH_RCV_TOO_LARGE` and the actual size of the message is returned in the `msgh_size` field of the message header.

The receive operation can generate the following return codes. These return codes imply that the call did not dequeue a message:

`MACH_RCV_INVALID_NAME`
: The specified `rcv_name` was invalid.

`MACH_RCV_IN_SET`
: The specified port was a member of a port set.

`MACH_RCV_TIMED_OUT`
: The timeout interval expired.

`MACH_RCV_INTERRUPTED`
: A software interrupt occurred.

`MACH_RCV_PORT_DIED`
: The caller lost the rights specified by `rcv_name`.

`MACH_RCV_PORT_CHANGED`
: `rcv_name` specified a receive right that was moved into a port set during the call.

`MACH_RCV_TOO_LARGE`
: When using `MACH_RCV_LARGE`, and the message was larger than `rcv_size`. The message is left queued, and its actual size is returned in the `msgh_size` field of the message buffer.

These return codes imply that a message was dequeued and destroyed:

`MACH_RCV_HEADER_ERROR`

A resource shortage prevented the reception of the port rights in the message header.

`MACH_RCV_INVALID_NOTIFY`

When using `MACH_RCV_NOTIFY`, the `notify` argument did not denote a valid receive right.

`MACH_RCV_TOO_LARGE`

When not using `MACH_RCV_LARGE`, a message larger than `rcv_size` was dequeued and destroyed.

In these situations, when a message is dequeued and then destroyed, the reply port and all port rights and memory in the message body are destroyed. However, the caller receives the message's header, with all fields correct, including the destination port but excepting the reply port, which is `MACH_PORT_NULL`.

These return codes imply that a message was received:

`MACH_RCV_BODY_ERROR`

A resource shortage prevented the reception of a port right or out-of-line memory region in the message body. The message header, including the reply port, is correct. The kernel attempts to transfer all port rights and memory regions in the body, and only destroys those that can't be transferred.

`MACH_RCV_INVALID_DATA`

The specified message buffer was not writable. The calling task did successfully receive the port rights and out-of-line memory regions in the message.

`MACH_MSG_SUCCESS`

A message was received.

Resource shortages can occur after a message is dequeued, while transferring port rights and out-of-line memory regions to the receiving task. The `mach_msg` call returns `MACH_RCV_HEADER_ERROR` or `MACH_RCV_BODY_ERROR` in this situation. These return codes always carry extra bits (bitwise-or'ed) that indicate the nature of the resource shortage:

`MACH_MSG_IPC_SPACE`

There was no room in the task's IPC name space for another port name.

`MACH_MSG_VM_SPACE`

There was no room in the task's VM address space for an out-of-line memory region.

`MACH_MSG_IPC_KERNEL`

A kernel resource shortage prevented the reception of a port right.

`MACH_MSG_VM_KERNEL`

A kernel resource shortage prevented the reception of an out-of-line memory region.

If a resource shortage prevents the reception of a port right, the port right is destroyed and the receiver sees the name `MACH_PORT_NULL`. If a resource shortage prevents the reception of an out-of-line memory region, the region is destroyed and the caller receives a zero address. In addition, the `msgt_size` (`msgtl_size`) field in the data's type descriptor is changed to zero. If a resource shortage prevents the reception of out-of-line memory carrying port rights, then the port rights are always destroyed if the memory region can not be received. A task never receives port rights or memory regions that it isn't told about.

A.8 Atomicity

The `mach_msg` call handles port rights in a message header atomically. Port rights and out-of-line memory in a message body do not enjoy this atomicity guarantee. The message body may be processed front-to-back, back-to-front, first out-of-line memory and then port rights, or in some random order.

For example, consider sending a message with the destination port specified as `MACH_MSG_-TYPE_MOVE_SEND` and the reply port specified as `MACH_MSG_TYPE_COPY_SEND`. The same send right, with one user-reference, is supplied for both the `msgh_remote_port` and `msgh_local_port` fields. Because `mach_msg` processes the message header atomically, this succeeds. If `msgh_remote_port` were processed before `msgh_local_port`, then `mach_msg` would return `MACH_SEND_INVALID_REPLY` in this situation.

On the other hand, suppose the destination and reply port are both specified as `MACH_MSG_-TYPE_MOVE_SEND`, and again the same send right with one user-reference is supplied for both. Now the send operation fails, but because it processes the header atomically, `mach_msg` can return either `MACH_SEND_INVALID_DEST` or `MACH_SEND_INVALID_REPLY`.

For example, consider receiving a message at the same time another thread is deallocating the destination receive right. Suppose the reply port field carries a send right for the destination port. If the deallocation happens before the dequeuing, then the receiver gets `MACH_RCV_PORT_DIED`. If the deallocation happens after the receive, then the `msgh_local_port` and the `msgh_remote_port` fields both specify the same right, which becomes a dead name when the receive right is deallocated. If the deallocation happens between the dequeue and the receive, then the `msgh_local_port` and `msgh_remote_port` fields both specify `MACH_PORT_DEAD`. Because the header is processed atomically, it is not possible for just one of the two fields to hold `MACH_PORT_DEAD`.

The `MACH_RCV_NOTIFY` option provides a more likely example. Suppose a message carrying a send-once right reply port is received with `MACH_RCV_NOTIFY` at the same time the reply port is destroyed. If the reply port is destroyed first, then `msgh_remote_port` specifies `MACH_PORT_DEAD` and the kernel does not generate a dead-name notification. If the reply port is destroyed after it is received, then `msgh_remote_port` specifies a dead name for which the kernel generates a dead-name notification. It is not possible to receive the reply port right and have it turn into a dead name before the dead-name notification is requested; as part of the message header the reply port is received atomically.

A.9 Caveats

Sending out-of-line memory with a non-page-aligned address, or a size which is not a page multiple, works but with a caveat. The extra bytes in the first and last page of the received memory are not zeroed, so the receiver can peek at more data than the sender intended to transfer. This might be a security problem for the sender.

If `MACH_RCV_TIMEOUT` is used without `MACH_RCV_INTERRUPT`, then the timeout duration might not be accurate. When the call is interrupted and automatically retried, the original timeout is used. If interrupts occur frequently enough, the timeout interval might never expire. `MACH_SEND_TIMEOUT` without `MACH_SEND_INTERRUPT` suffers from the same problem.

A.10 Bootstrapping

When a task is first created, it holds no port rights. The kernel provides a few system calls that let the task “bootstrap” itself and acquire initial port rights that in turn may be used in RPCs to acquire more port rights.

A.10.1 mach_reply_port

```
mach_port_t mach_reply_port();
```

The `mach_reply_port` system call allocates a port. The calling task acquires the receive right for the port.

A.10.2 mach_thread_self

```
mach_port_t mach_thread_self();
```

The `mach_thread_self` system call returns the calling thread's thread port. The thread can use the thread port to perform operations upon the kernel object that represents itself.

`mach_thread_self` has an effect equivalent to receiving a send right for the thread port. `mach_thread_self` returns the name of the send right. In particular, successive calls will increase the calling task's user-reference count for the send right.

A.10.3 mach_task_self

```
mach_port_t mach_task_self();
```

The `mach_task_self` system call returns the calling thread's task port. The thread can use the task port to perform operations upon the kernel object that represents its task.

`mach_task_self` has an effect equivalent to receiving a send right for the task port. `mach_task_self` returns the name of the send right. In particular, successive calls will increase the calling task's user-reference count for the send right.

However, the Mach runtime library retrieves the task port and stores it in a global variable. The standard header files define `mach_task_self()` to be a macro that retrieves this value. In this case, "calls" to `mach_task_self()` do not increment the calling task's user-reference count for the send right.

A.10.4 mach_host_self

```
mach_port_t mach_host_self();
```

The `mach_host_self` system call returns the calling thread's host port. The thread can use the host port to perform operations upon the kernel object that represents the current machine.

`mach_host_self` has an effect equivalent to receiving a send right for the host port. `mach_host_self` returns the name of the send right. In particular, successive calls will increase the calling task's user-reference count for the send right.

Appendix B

The Control Transfer Interface

This appendix describes in more detail the control transfer interface that was introduced in Section 3.4. The functions in the interface are divided into four functional groups: low-level stack management, high-level stack management, context-switch, and kernel exit.

Clients of the control transfer interface, such as interprocess communication and thread management code, only use the high-level stack management, context-switch, and kernel exit operations. Providers of the control transfer interface can choose to implement the high-level stack management operations directly, or to implement the two low-level primitives and take advantage of a standard machine-independent implementation of the high-level operations in terms of the low-level primitives. Machine-dependent code asserts full control over stack management by defining the `MACHINE_STACKS` conditional compilation symbol in a header file, thus disabling the standard machine-independent implementation.

B.1 Low-Level Stack Management

B.1.1 stack_attach

```
void stack_attach(thread, stack, continuation)
   thread_t thread;
   vm_offset_t stack;
   void (*continuation)(thread_t);
```

Attaches the kernel stack to the thread and initializes 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.

B.1.2 stack_detach

```
vm_offset_t stack_detach(thread)
   thread_t thread;
```

Detaches and returns the thread's kernel stack. This function may perform other machine-dependent finalization, such as copying user register context from the stack to a separate data structure.

B.2 High-Level Stack Management

B.2.1 stack_alloc

```
void stack_alloc(thread, continuation)
   thread_t thread;
   void (*continuation)(thread_t);
```

Allocates a new stack from kernel virtual memory and then calls `stack_attach(thread, stack, continuation)`.

B.2.2 stack_alloc_try

```
boolean_t stack_alloc_try(thread, continuation)
   thread_t thread;
   void (*continuation)(thread_t);
```

Makes a non-blocking attempt to allocate a new kernel stack. `stack_alloc_try` uses a cache of unused kernel stacks; if the cache is empty, it returns `FALSE`. If `stack_alloc_try` succeeds in allocating a stack, then it initializes the stack with `stack_attach` and returns `TRUE`.

B.2.3 stack_free

```
void stack_free(thread)
   thread_t thread;
```

Calls `stack_detach(thread)` to get the thread's stack, and then frees the stack by returning it to the cache of unused stacks.

B.2.4 stack_collect

```
void stack_collect()
```

When the system is running out of free physical memory, the pageout daemon calls `stack_collect` to recover unused resources in the stack management system. The stack management system may cache some stacks, to improve the performance of `stack_alloc` and `stack_free` and so that `stack_alloc_try` has a ready source of stacks. `stack_collect` should return to the virtual memory system any unused stacks that may be cached.

B.2.5 stack_privilege

```
void stack_privilege(thread)
   thread_t thread;
```

Makes the thread *stack-privileged*. The `stack_alloc_try` operation must always succeed for stack-privileged threads. The `stack_privilege` operation allocates a reserved stack for the thread, so that if the stack cache is empty the thread can instead use its reserved stack. The `stack_free` operation must be careful not to return a reserved stack to the stack cache. Clients of `stack_handoff` should avoid using handoff when the current thread is running on its reserved stack.

B.3 Context Switch

B.3.1 current_thread

```
thread_t current_thread();
```

Returns a pointer to the current thread.

B.3.2 stack_handoff

```
void stack_handoff(old_thread, new_thread)
   thread_t old_thread;
   thread_t new_thread;
```

Performs a stack handoff, moving the current kernel stack from the current thread to the new thread. `stack_handoff` changes address spaces if necessary. When `stack_handoff` returns, the value of `current_thread()` has changed to be the new thread.

B.3.3 call_continuation

```
void call_continuation(continuation)
   void (*continuation)(void);
```

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.

B.3.4 switch_context

```
thread_t switch_context(old_thread, continuation, new_thread)
   thread_t old_thread;
   void (*continuation)(void);
   thread_t new_thread;
```

Resumes the new thread on its kernel stack. This stack may be left behind from a previous call to `switch_context` without a continuation, or the stack may come from `stack_attach`. This call changes address spaces if necessary.

If a continuation for the current thread is supplied, then `switch_context` does not save register context and does not return. Otherwise, `switch_context` saves the current thread's register context and kernel stack and returns when the calling thread is rescheduled, returning the previously running thread.

B.3.5 load_context

```
void load_context(new_thread)
   thread_t new_thread;
```

Loads the first thread on a processor. Like `switch_context`, but without a currently running thread. Used once per processor during system initialization.

B.4 Kernel Exit

B.4.1 thread_syscall_return

```
void thread_syscall_return(kr)
   kern_return_t kr;
```

Causes the current thread to return to user space from a system call, with the specified return value as the status code for the system call.

B.4.2 thread_exception_return

```
void thread_exception_return()
```

Causes the current thread to return to user space from an exception, page-fault, or device interrupt.

B.4.3 thread_bootstrap_return

```
void thread_bootstrap_return()
```

Causes a newly-created thread to enter user space for the first time.

B.4.4 thread_set_syscall_return

```
void thread_set_syscall_return(thread, retval)
   thread_t thread;
   kern_return_t retval;
```

Changes the thread's saved system call user register context to exception-type user register context. Before this call, the thread could have used `thread_syscall_return`. After `thread_set_syscall_return`, the thread should use `thread_exception_return`. When it does, the user's system call returns with `retval` as the status code.

Bibliography

[Accetta *et al.* 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93-113, June 1986.

[Anderson *et al.* 92] 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.

[Appel & Jim 89] Andrew W. Appel and Trevor Jim. Continuation-Passing, Closure-Passing Style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages,* pages 293-302, January 1989.

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

[Bershad *et al.* 90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems,* 8(1):37-55, February 1990.

[Bershad 90] Brian N. Bershad. *High Performance Cross-Address Space Communication.* PhD dissertation, University of Washington, Seattle, WA, June 1990.

[Birrell 89] Andrew D. Birrell. An Introduction to Programming with Threads. Research Report 35, DEC Systems Research Center, January 1989.

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

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

[Black *et al.* 91] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel Operating System Architecture and Mach. *Journal of Information Processing*, 14(4):442-453, December 1991.

[Carter *et al.* 91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152-164, October 1991.

[Carter 93] John Carter. Personal communication, June 14, 1993.

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

[Cheriton 91] David Cheriton. Personal communication, May 9, 1991.

[Clinger & Rees 92] William Clinger and Jonathan Rees (eds). Revised^4 Report on the Algorithmic Language Scheme. TR 92-1261, Cornell University Dept. of Computer Science, Ithaca, NY, January 1992.

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

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

[Custer 93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

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

[Draves 91] Richard P. Draves. Page Replacement and Reference Bit Emulation in Mach. In *Proceedings of the USENIX Mach Symposium*, pages 201-212, November 1991.

[Dybvig & Hieb 89] R. Kent Dybvig and Robert Hieb. Engines from Continuations. *Computer Languages*, 14(2):109-123, 1989.

[Eykholt *et al.* 92] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams. Beyond Multiprocessing: Multithreading the SunOS Kernel. In Proceedings of the Summer 1992 USENIX Conference, pages 11-18, 1992.

[Felleisen *et al.* 88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps. In *Conference Record of the 1988 ACM Symposium on LISP and Functional Programming*, pages 52-62, July 1988

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

[Goldberg & Robson 83] Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA, 1983.

[Golub *et al.* 90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87-95, June 1990.

[Golub & Draves 91] David B. Golub and Richard P. Draves. Moving the Default Memory Manager out of the Mach Kernel. In *Proceedings of the USENIX Mach Symposium*, pages 177-188, November 1991.

[Habermann & Nassi 80] A. N. Habermann and Isaac R. Nassi. Efficient Implementation of Ada Tasks. CMU-CS-80-103, Computer Science Department, Carnegie-Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, January 1980.

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

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

[Haynes *et al.* 84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and Coroutines. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 293-298, August 1984.

[Haynes *et al.* 86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Computer Languages*, 11(3/4):143-153, 1986.

[Haynes & Friedman 87] Christopher T. Haynes and Daniel P. Friedman. Abstracting Timed Preemption with Engines. *Computer Languages*, 12(2):109-121, 1987.

[Hildebrand 92] Dan Hildebrand. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113-126, April 1992.

[Hutchinson *et al.* 89] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. 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.

[Kane 88] Gerry Kane. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Khanna *et al.* 92] Sandeep Khanna, Michael Sebrée, and John Zolnowsky. Realtime scheduling in SunOS 5.0. In Proceedings of the Winter 1992 USENIX Conference, pages 375-390, 1992.

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

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

[Letwin 88] Gordon Letwin. *Inside OS/2*. Microsoft Press, Redmond, WA, 1988.

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

[Malan *et al.* 91] Gerald Malan, Richard Rashid, David Golub, and Robert Baron. DOS as a Mach 3.0 Application. In *Proceedings of the USENIX Mach Symposium*, pages 27-40, November 1991.

[McJones & Swart 89] Paul. R. McJones and Garret. F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. In *Proceedings of the Winter 1989 USENIX Conference*, pages 393-404, February 1989. Also DEC SRC Report 21, DEC Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, 1987.

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

[Mogul *et al.* 87] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39-51, November 1987.

[Motorola 90a] Motorola. *MC88100 RISC Microprocessor User's Manual*, 2nd. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Motorola 90b] Motorola. *MC88200 Cache/Memory Management Unit User's Manual*, 2nd. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Nelson 91] Greg (Charles G.) Nelson, ed. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Ousterhout *et al.* 88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, B. B. Welch. The Sprite Network Operating System. *Computer*, 21(2):23-36, February 1988.

[Phelan *et al.* 93] James M. Phelan, James Arendt, and Gary R. Ormsby. An OS/2 Personality on Mach. In *Proceedings of the USENIX Mach III Symposium*, pages 191-202, April 1993.

[Rashid *et al.* 87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 31-39, October 1987.

[Rashid *et al.* 89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Doug Orr, Richard Sanzi. Mach: A Foundation for Open Systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109-113, September 1989.

[Ritchie & Thompson 78] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Bell System Technical Journal*, 57(6): 1905-1929, July-August 1978.

[Roy 93] Paul J. Roy. Unix File Access and Caching in a Multicomputer Environment. In *Proceedings of the USENIX Mach III Symposium*, pages 21-38, April 1993.

[Saltzer 66] J. H. Saltzer. Traffic Control in a Multiplexed Computer System. MAC-TR-30, Massachusetts Institute of Technology, Cambridge, MA, July 1966.

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

[Scheifler & Gettys 90] R. W. Scheifler and J. Gettys. The X Window System. *Software—Practice and Experience*, 20(2):5-34, October 1990.

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

[Schulman 93] Andrew Schulman. *Undocumented DOS: A Programmer's Guide to Reserved MS-DOS Functions and Data Structures, 2nd ed*. Addison-Wesley, Reading, MA, 1993.

[Sitaram & Felleisen 90] Dorai Sitaram and Matthias Felleisen. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation*, 3:67-99, 1990.

[Sites 92] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[Steele 78] Guy L. Steele Jr. RABBIT: A Compiler for SCHEME, MIT AI Lab, May 1978.

[Stoy & Strachey 72] J. E. Stoy and C. Strachey. OS6—An Experimental Operating System for a Small Computer. *The Computer Journal*, 15(2):117-124,15(3):195-203, 1972.

[Strachey & Wadsworth 74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, January 1974.

[Tanenbaum 91] Andy Tanenbaum. Personal communication, May 14, 1991.

[Tanenbaum 87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Tanenbaum *et al.* 90] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46-63, December 1990.

[Tevanian 87] Avadis Tevanian, Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environment: The Mach Approach*. PhD dissertation, Carnegie Mellon University, Pittsburgh, PA, December 1987.

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

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

[Young 89] Michael Wayne Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. PhD dissertation, Carnegie Mellon University, Pittsburgh, PA, November 1989.

[Yuhara *et al.* 94] M. Yuhara, B. N. Bershad, C. Maeda, J. E. B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, pages 153-166, January 1994.