Mach Threads and the Unix Kernel:
The Battle for Control

Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub,
David L. Black, Eric Cooper and Michael W. Young.

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper examines a kernel implemented lightweight process mechanism built for the Mach operating system. The pros and cons of such a mechanism are discussed along with the problems encountered during its implementation.

1. Introduction

The early Unix notion of process was based on the hardware abstraction of its day: a single CPU executing within a memory address space. Even today, although, multiprocessors are becoming increasingly common, neither Unix System V nor 4.3 BSD provide a way to manage more than one thread of control within an address space.

The addition of lightweight processes to Unix would provide many advantages. In fact, the lack of kernel support has caused Unix programmers to implement a variety of coroutine packages to support multi-stack applications. Lightweight threads of control can allow a programmer to encapsulate computations with their stack state and thus achieve greater modularity. Research systems, such as THOTH [2] and its successor, Stanford's V Kernel [3], have shown that multiple threads of control within a single process can be an especially important tool for writing server applications. A thread package could provide an attractive way to take advantage of the parallelism afforded by tightly-coupled shared memory multiprocessors.

This paper examines a kernel-implemented thread facility built for the Mach operating system [1]. The pros and cons of such a mechanism are discussed along with the problems encountered during its implementation.

2. Kernel Implemented Threads vs. Coroutines

Some of the advantages of lightweight processes can be achieved by out-of-kernel solutions, but often at the expense of either preemption or parallelism. Two approaches are common: multi-process shared memory implementations and single process coroutines.

Parallel execution can be achieved with multiple processes in conjunction with some kind of shared memory facility. For example, in Dynix [4] users can allocate a number of processes equal to the number of processors and effectively manage shared computations through an *mmap*ed region of shared memory. Similar tricks allow programmers to build multiprocessor applications using Encore's UMAX [5] operating system. Typically such systems amount to a second layer of scheduling similar to that used within the operating system itself.

A significant advantage of coroutine packages is that they can significantly reduce the costs of multi-thread management or at least isolate them within a user process. Out-of-kernel coroutine packages do, however, have many problems:

- Scheduling is very difficult to do. Most coroutine packages use non-preemptive scheduling.
- It is impossible to have truly parallel execution in a pure coroutine package.
- If a coroutine takes a page fault or other type of trap that causes it to wait (e.g. for disk I/O), then all other coroutines must wait.
- Only a single coroutine may be executing a system call. Therefore, if coroutine executes a system call that blocks causes the entire set of coroutines to block.

The primary disadvantage of an in-kernel thread implementation is potential cost. In addition to the cost of crossing the user process/kernel protection boundary with a trap or system call, there is also the cost of thread data structures, which must be managed in kernel virtual address space, and the cost of general purpose preemptive scheduling, which will typically be much higher than those of a specialized coroutine package.

After considering the alternatives and their problems, it was decided that in Mach it would make sense to provide primitive multi-threaded support within the kernel which would provide for both parallelism and preemption. This support would then serve as the base upon which lightweight process packages could be implemented.

3. Mach Task and Thread Primitives

Mach splits the Unix abstraction of process into two components: the *task* and the *thread*. A Mach *task* consists of a collection of system resources, including an address space. It can be thought of as that part of a Unix process consisting of its address space, file descriptors, resource usage information, etc. In essence, a task is a process without a flow of control or register set (hardware state).

A Mach *thread* is the basic unit of execution. A thread executes within the context of exactly one task. However, any number of threads may execute within a single task. Threads execute in pseudo-parallel on a uniprocessor. When running on a tightly coupled multiprocessor, multiple threads may execute in parallel. A traditional BSD process is implemented in Mach as a task with a single thread of control.

Appendices I and II list the operations supplied by Mach for creating, managing and destroying tasks and threads. Note that all such operations are performed using object handles which are in fact capabilities to communication channels (i.e., Mach *ports*).

4. User Level Thread Synchronization

At any given point in time, a thread can be in one of three states:

1. A thread that is in *running state* is either executing on some processor, or is eligible for execution on a processor as far as the user is concerned. A thread may be in running state yet blocked for some reason inside the kernel (perhaps waiting for a page fault to be handled).

2. If a thread is in *will-suspend* state, then it can still execute on some processor until a call to *thread_wait* is invoked.

3. A thread that is in *suspended* state is not executing on processor. The thread will not execute on any processor until it returns to running state.

Each of these states can also apply to a task. That is, a task may be in running, will-suspend or suspended state. The state of a task will affect all threads executing within that task. For example, a thread can be eligible for execution only if both it and its task are in the running or the will-suspend state.

The Mach kernel does not enforce a synchronization model. Instead, it provides basic primitives upon which different models of synchronization may be built. One form such synchronization could take would be the Mach IPC facility [1]. Should an application desire its own thread-level synchronization, it can use the suspend, resume and wait primitives. For example, to implement P and V style semaphores with shared memory, one could use:

```
P(semaphore)
{
        lock(semaphore);
        while (semaphore->inuse) {
                thread_suspend(thread_self());
                enqueue(semaphore->queue, thread_self());
                unlock(semaphore);
                thread_wait(thread_self(), TRUE);
                lock(semaphore);
        }
        semaphore->inuse = TRUE;
        unlock(semaphore);
}

V(semaphore)
{
        lock(semaphore);
        semaphore->inuse = FALSE;
        next = dequeue(semaphore->queue);
        if (next != THREAD_NULL)
                thread_resume(next);
        unlock(semaphore);
}
```

In this example, lock and unlock could be implemented as spin locks on shared memory. To perform the P operation, the caller checks if the semaphore is in use. If so, it puts itself on a queue of threads waiting for the semaphore and goes into a suspended state. When it is placed back in running state it once again checks the semaphore, suspending itself again if necessary. To perform the V operation, a thread checks for other waiting threads and places the first thread in the semaphore queue into the running state.

Note that placing a thread into the suspended state is separated into a suspend operation followed by a wait operation. If there were not such a separation, it would be impossible for an application to correctly synchronize unless the kernel provided semaphores directly. If suspend/wait were a single operation a thread would be forced to call it either before or after unlocking the semaphore. If the thread made the call before unlocking the semaphore then the application would deadlock because the semaphore was never unlocked. If the thread made the call after unlocking the semaphore then it would be possible for the thread holding the semaphore to perform its resume before the waiting thread is able to suspend itself. In this case, the thread would suspend itself and never be resumed.

Of course, the kernel could implement semaphores directly (as does, for example, System V). It was felt, however, that a semaphore package would only add yet another synchronization mechanism to the kernel on top of that provided by the Mach IPC facility. The kernel would inevitably implement only a small set of semaphore types and applications that wanted to use different semaphore semantics would still be forced to use an extra layer of synchronization and manage additional data structures.

5. The C-Threads Package

The exported thread primitives are intentionally low level to allow flexibility in dealing with a variety of programming languages and architectures. By providing a minimal kernel interface, it is possible to implement many different application or language interfaces to threads without burdening some applications in favor of others. For example, a feature that provided dynamically growing stacks might be useful for a naive C programmer, but it might be extra baggage for a Lisp programmer.

Higher level interfaces to threads can be provided in the form of:

- run time libraries,
- new language constructs and/or
- home grown packages developed for specific applications.

One such high-level package for programming in C, called *C-threads*, has already been implemented. It provides a high level C interface to the low level thread primitives along with a collection of other mechanisms useful in various parallel programming paradigms (similar to those available in languages such as Mesa [6]).

The C-Threads package provides

- multiple threads of control for parallelism,
- shared variables,
- mutual exclusion for critical sections and
- condition variables for synchronization of threads.

To provide multiple threads of control, the C-Threads interface defines *cthread_fork* for creating new threads, *cthread_exit* for exiting threads and *cthread_join* to wait for a particular thread to finish.

Threads that wish to access shared data may use the mutual exclusion facilities provided by C-Threads. In particular, *mutex_alloc* and *mutex_free* allocate and deallocate *mutex* objects. The *mutex* objects support the functions *mutex_lock* and *mutex_unlock* which correspond to typical P and V operations.

Synchronization in C-Threads may also be accomplished with condition variables. *Condition_alloc* and *condition_free* allocate and free condition variables. When a thread wishes to indicate that a condition is true, it uses *condition_signal* to awaken at least one of the threads waiting for the condition. The *condition_broadcast* primitive causes all threads waiting for a condition to wake up. A thread may of course wait for a condition using *condition_wait*.

There are, currently, three separate C-Threads implementations. The first implements threads as coroutines in a single task. The second uses a separate task for each cthread, using inherited shared memory to partially simulate the environment in which multiple threads run. The third implementation uses the thread primitives provided by the kernel.

The coroutine version is generally easier to use for debugging since the order of context switching is repeatable and the user need not worry about concurrent calls to C library routines. However, the coroutine version can not exhibit parallelism as the other two versions do. The multiple task version can be an effective way to achieve parallelism on architectures which do not allow full, uniform access-delay sharing of memory.

6. The Effect of Threads on Unix Features

From a Unix programmer's perspective, the separation of the Unix protection domain from its control abstraction has been accomplished at no apparent cost. Mach provides complete emulation of 4.3BSD Unix, even for binaries on VAX machines. Overall system performance has not been eroded.

However, should one desire to use a multithreaded task along with Unix features, there are many potential pitfalls. Unix was not designed to work in a multithreaded environment. Some of the obvious problems are:

- The semantics of common functions (e.g. fork) are not well defined in the presence of multiple threads.
- Many standard library routines return results in static areas.
- Most C compilers return structures in a static area.
- The definitions of static returned values such as *errno* are inappropriate for a multithreaded environment.
- Many library routines, never expected to be run in a multithreaded application, are coded in non-reentrant ways. Many traditional Unix libraries would not even work if a signal routine were to be called at the wrong time!

Where the semantics of Unix operations are not well defined in the presence of multiple threads, it was necessary to determine some reasonable definition. Two examples of this are *fork* and *signals*.

The Unix fork primitive raises the question: "When a thread in a task containing multiple threads executes the fork system call, which threads does the child task contain?". There are two possible answers:

1. The child task contains exactly one thread corresponding to the calling thread.
2. The child task contains the same number of threads as the parent. Each thread in the child corresponds to a thread in the parent.

Mach implements the first choice, which is really the most logical when the properties of tasks and threads are considered. Fork is largely an address space manipulation and corresponds very closely to the *task_create* operation. Unix process semantics dictate that the child must contain at least one thread. The logical choice for this thread is a replica of the calling thread. This choice also corresponds to the common case when a thread within a server task decides to fork to perform an operation in a separate address space.

Signals present an interesting problem in the domain of multiple threads. Are signals sent to tasks or threads? Considering that the logical equivalent of a Mach task is a Unix process, and that signals are sent to processes, it is appropriate to define signals as being sent to a task. Unfortunately, a task is not an executable unit and can therefore not handle a signal. To overcome this problem, the Mach kernel chooses some thread within the task to handle the signal. The actual thread that will handle the signal is not well-defined. In fact, the current implementation causes the first thread to notice the signal to be the handler. This is clearly not an optimal solution because it can seriously confuse a Unix programmer that wishes to use signals to cause a stack unwinding operation such as *longjmp*. The better long term solution is to convert signals into Mach IPC messages. Each task could then designate one or more threads that would receive signals on a special *signal port*.

7. Implementation: Details, Problems and Issues

Within the Mach kernel, the task (sic) of incorporating threads exacted a significant toll on the implementors. This was due to the fact that Mach currently provides for Unix compatibility directly in the kernel. The 4.3 BSD kernel code was designed (presumably unintentionally) to make a multiple thread per address space implementation very difficult. For example, both the u-area and kernel stack reside in the user's address space and are even assumed to exist at the same address for all processes. Unix process management is not restricted to a handful of scheduling modules. Instead, it is spread throughout unrelated kernel code. A form of process management can even been found in device drivers. The 4.3 BSD signal mechanism is neither well defined nor even appropriate for such an environment.

Perhaps the most annoying problem was that of u-area management. There are literally thousands of lines of kernel code that use *u.* to reference the u-area directly. This assumes that the u-area is at the same address for all processes. Since threads must share an entire address space and must each have their own u-like data structure, the traditional u-area cannot exist at a single, unique address. In fact, the problem is even worse: some fields of the old u-area refer to data which should be thread specific properties while other fields refer to task specific properties. Therefore, within the BSD compatibility code, each u-area reference can no longer be a simple memory reference to a fixed address. Instead, each u-area reference must be a pointer dereference with the pointer depending on whether the desired field is a task or thread feature. Rather than inspect and modify each of the thousands of lines of C code, a few tricks were played with the C preprocessor. Two new structures, *uthread* and *utask* were defined to hold the thread and task specific u-area information. For example:

```
struct uthread {
        int     uu_thread1;
        int     uu_thread2;
        .
        .
        .
};

struct utask {
        int     uu_task1;
        int     uu_task2;
        .
        .
        .
};
```

uu_thread1, uu_thread2, ... corresponded to fields in the typical Unix u-area. Next, *u* itself was defined as follows:

```
#define u       (current_thread()->u_address)
```

with the *u_address* field of the thread structure defined as:

```
struct thread {
        .
        .
        .
        struct u_address {
                struct uthread  *uthread;
                struct utask    *utask;
        } u_address;
        .
        .
        .
};
```

Finally, each potential u-area field was defined as:

```
#define u_thread1       uthread->uu_thread1
#define u_thread2       uthread->uu_thread2
        .
        .
        .
#define u_task1         utask->uu_task1
#define u_task2         utask->uu_task2
```

When a task is created it is allocated a utask structure. When a thread is created it is allocated a uthread structure. The pointer to this structure, along with the pointer to the task's utask structure, are then saved in the u_address sub-structure of the thread structure.

The good news is that these definitions handle almost all uses of the u-area. The bad news is that most u-area references change from one instruction to several. This increases both execution time and code space. After some intense hacking, each u-area reference was reduced to only 3 VAX instructions. The first instruction fetches the current thread, the second instruction loads the appropriate pointer (uthread or utask), and the third instruction performs the actual u-area reference.

Even though each u-area reference is now significantly more expensive than in a standard Unix system, the Mach kernel still performs better than a 4.3 kernel in measurements of overall performance -- largely due to improvements in Mach's handling of virtual memory. For example, to compile all programs in /bin on a vanilla 4.3 system (using a CMU enhanced cc and cpp) takes 1017 seconds on a VAX 780 (with a Fujitsu Eagle disk drive). A Mach kernel without multiple thread support and normal u-area references takes only 964 seconds. A Mach kernel with multiple thread support and the expensive u-area references requires 986 seconds to complete the test. Given that approximately 700 seconds is spent in user time, and since a Mach kernel can not improve on the user time of existing binaries, it makes sense to factor that 700 seconds out of the measurements. Therefore, we see that 4.3 is responsible for 1017 - 700 = 317 seconds. A non-thread Mach kernel is responsible for 964 - 700 = 264 seconds. A thread Mach kernel is responsible for 986 - 700 = 286 seconds. So, a non-thread Mach kernel is approximately (317-264)/317 = 16.7% faster than 4.3. A thread Mach kernel is still (317-286)/317 = 9.7% faster than 4.3.

It is expected that some improvement in the kernel supporting threads will be gained by identifying u-area hotspots: those places in the kernel that make many references to the u-area. Once identified, these sections of kernel code can be reworked to avoid using the u-area, or to use it in a more efficient way.

8. Performance Issues

While an order of magnitude less expensive than the Unix *fork*/*exit* operations, *thread_create* and *thread_terminate* are still moderately expensive operations as indicated in table 8-1. In addition to the O(1 millisecond) each operation takes on a MicroVAX II, there are also the memory costs incurred by the thread data structures themselves and the necessity for a (pagable) kernel stack for each thread which must be physically resident when the thread is runnable. For this reason, an application that needs huge numbers of thread-like entities (perhaps millions) would probably be best implemented as a hybrid of kernel-supplied threads and coroutines. That is, a huge number of coroutines would map to a much smaller number of threads executing in a single process. The number of threads used could correspond either to the number of available processors or the number of concurrently executing system calls or traps.

Some of the costs of threads can be "optimized away". For example, the C-threads package caches threads which have exited so they can be reused when a new *cthread_fork* is called. In a multiprocessor with a sufficient number of processors, context switching is eliminated entirely. On a single processor machine, however, many of the costs of scheduling Unix processes remain in the scheduling of threads.

Fork/Exit vs. Thread Create/Terminate

Kernel operation	VAX Instructions Executed (typical case)
fork	3069
wait3	3440
exit	916
fork/wait3/exit (total)	7425
thread_create	375
thread_exit	409
thread total	784

Table 8-1:

Number of VAX CPU instructions executed.
(Mach, MicroVAX II, 4K page size)

9. Conclusion and Status

The Mach thread implementation is running (April 1987) on multiprocessor and uniprocessor VAX, Encore and Sequent machines within CMU. A version of Eric Cooper's C-Threads package which uses threads is also working. Mach is being released externally to interested researchers. The first release (Release 0) of Mach began in December of 1986.

10. Acknowledgements

The multiple thread kernel support was implemented by Avie Tevanian, David Golub and David Black. Eric Cooper implemented the C-Thread package and along with others had much input on the final interface. No one in their right mind would claim credit for thinking up the u-area hacks.

I. Thread Operations

Following is a list of all kernel supported thread operations:

```
    thread_create(task, child, child_data)
            task_t          task;           /* parent task */
            thread_t        *child;         /* new thread */
            port_t          *child_data;    /* child data port */
```

Thread_create create a new thread in the specified task. Initially, the thread is in *suspended* state and its registers contain undefined values.

```
    thread_terminate(thread)
            thread_t        thread;         /* thread to terminate */
```

Thread_terminate destroys the specified thread.

```
    thread_suspend(thread)
            thread_t        thread;         /* thread to suspend */
```

The specified thread is placed in *will-suspend* state.

```
    thread_resume(thread)
            thread_t        thread;         /* thread to resume */
```

The specified thread is placed in *running* state.

```
    thread_wait(thread, wait)
            thread_t        thread;         /* thread to cause to wait */
            boolean_t       wait;           /* wait for it to stop? */
```

If the specified thread is in *will-suspend* state then *thread_wait* will place it in *suspended* state. If the *wait* parameter is TRUE, the calling thread will wait for the thread to come to a complete stop.

```
    thread_status(thread, status)
            thread_t        thread;         /* thread to query */
            thread_status_t *status;        /* thread status information */
```

Thread_status returns the register state of the specified thread. The *status* parameter returns the address of a machine-dependent status structure describing the register state for the machine type the thread is executing on.

```
    thread_mutate(thread, status)
            thread_t        thread;         /* thread to mutate */
            thread_status_t *status;        /* status information to set */
```

Thread_mutate sets the register state of the specified thread. As in *thread_status*, the *status* structure is machine-dependent.

II. Task Operations

Following is a list of all kernel support task operations:

```
    task_create(parent, inherit, child, child_port)
            task_t          parent;         /* the parent task */
            boolean_t       inherit;        /* pass VM to child? */
            task_t          *child;         /* new task */
            port_t          *child_port;    /* new task's data port */
```

Task_create creates a new task. The child's address space is created using the parents inheritance values if the *inherit* flag is TRUE. If the *inherit* flag is FALSE, the child is created with an empty address space. Access to the child's task and data ports are returned in *child* and *child_port* respectively.

```
    task_terminate(task)
            task_t  task;                   /* task to terminate */
```

The specified task is destroyed.

```
    task_suspend(task)
            task_t  task;                   /* task to suspend */
```

The specified task is placed in *will-suspend* state.

```
    task_resume(task)
            task_t  task;                   /* task to resume */
```

The specified task is placed in *running* state.

```
    task_wait(task, wait)
            task_t          task;           /* task to cause to wait */
            boolean_t       wait;           /* wait for it to stop? */
```

If the specified task is in *will-suspend* state then *task_wait* places it in *suspended* state. If the *wait* flag is TRUE the calling thread will wait for all threads in the task to come to a complete stop.

```
    task_threads(task, list)
            task_t          task;           /* task to generate list for */
            thread_t        *list[];        /* list of threads */
```

Task_threads returns the list of all threads in a task.

```
    task_ports(task, list)
            task_t          task;           /* task to generate list for */
            port_t          *list[];        /* list of ports */
```

Task_ports returns the list of all ports the specified task has access to.

References

[1] 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 Summer Usenix*. July, 1986.

[2] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM :105-115, February, 1979.

[3] D. R. Cheriton and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proceedings of the 9th Symposium on Operating System Principles*, pages 128-139. ACM, October, 1983.

[4] Sequent Computer Systems, Inc.
Dynix Programmer's Manual
Sequent Computer Systems, Inc., 1986.

[5] Encore Computer Corporation.
UMAX 4.2 Programmer's Reference Manual
Encore Computer Corporation, 1986.

[6] Lampson, B.W. and D.D. Redell.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-113, February, 1980.

Table of Contents

List of Tables