SunOS Multi-thread Architecture

M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks
Sun Microsystems Inc.

ABSTRACT

We describe a model for multiple threads of control within a single UNIX process. The main goals are to provide extremely lightweight threads and to rationalize and extend the UNIX Application Programming Interface for a multi-threaded environment. The threads are intended to be sufficiently lightweight so that there can be thousands present and that synchronization and context switching can be accomplished rapidly without entering the kernel. These goals are achieved by providing lightweight user-level threads that are multiplexed on top of kernel-supported threads of control. This architecture allows the programmer to separate logical (program) concurrency from the required real concurrency, which is relatively costly, and to control both within a single programming model.

Introduction

The reasons for supporting multiple threads of control in SunOS fall into two categories, those motivated by multiprocessor hardware and those motivated by application concurrency. It is possible to exploit multiprocessors to varying degrees depending on how much the uniprocessor software base is modified. In the simplest case, only separate user processes can run on the additional processors; the applications are unchanged. To allow a single application to use multiple processors (e.g. array processing workload), the application must be restructured.

The second category of reasons for multiple threads of control is application concurrency. Many applications are best structured as several independent computations. A database system may have many user interactions in progress while at the same time performing several file and network operations. A window system can treat each widget as a separate entity. A network server may indirectly need its own service (and therefore another thread of control) to handle requests. In each case, although it is possible to write the software as one thread of control moving from request to request, the code may be simplified by writing each request as a separate sequence, and letting the language, library, and operating system handle the interleaving of the different operations.

These examples are not intended to be exhaustive, but they indicate the opportunities to exploit powerful hardware and build complex applications and services with this technology. The examples show that the user model for multiple threads of control must support a variety of applications and environments. The architecture should, where possible, use current programming paradigms and preserve software compatibility.

As is true of many system services today, the programmer's view of the multiple threads of control service is not always identical to what the kernel implements. The software view is created by a combination of the kernel, run-time libraries, and the compilation system. This approach increases the portability of applications and systems, by hiding some details of the implementation, while providing better performance, by allowing library code to do some work without involving the kernel.

The remainder of this paper is divided into five sections. The first section gives an overview of the architecture and introduces our terminology. The second section discusses our design goals and principles. The third section gives additional details of operation and interfaces and how the UNIX process model is reinterpreted in the new environment. The fourth section gives some performance data and operational experience. The last section compares this architecture with others.

The terminology of multiprocessor and multithreaded computation is unfortunately not universally agreed upon. We have chosen terms that are most common and have tried to be consistent, but the reader is warned that some people use these words with other meanings. Examples of other models can be found in the last section of this paper.

Multi-Threading Architecture Overview

The multi-threaded programming model has two levels. The most important level is the thread interface, which defines most aspects of the programming model. That is, programmers write programs using threads. The second level is the lightweight process (LWP) which is defined by the services the operating system must provide. After describing each level, we explain why both levels are essential.

Threads

A traditional UNIX process has a single thread of control. A thread of control, or more simply a thread, is a sequence of instructions being executed in a program. A thread has a program counter (PC) and a stack to keep track of local variables and return addresses. A multi-threaded UNIX process is no longer a thread of control in itself, instead it is associated with one or more threads. Threads execute independently. There is in general no way to predict how the instructions of different threads are interleaved, though they have execution priorities that can influence the relative speed of execution. In general, the number or identities of threads that an application process chooses to apply to a problem are invisible from outside the process. Threads can be viewed as execution resources that may be applied to solving the problem at hand.

Threads share the process instructions and most of its data. A change in shared data by one thread can be seen by the other threads in the process. Threads also share most of the operating system state of a process. Each sees the same open files. For example, if one thread opens a file, another thread can read it. Because threads share so much of the process state, threads can affect each other in sometimes surprising ways. Programming with threads requires more care and discipline than ordinary programming because there is no system-enforced protection between threads.

Each thread may make arbitrary system calls and interact with other processes in the usual ways. Some operations affect all the threads in a process. For example, if one thread calls `exit()`, all threads are destroyed. Other UNIX system services have new interpretations; e.g. a floating-point overflow trap applies to a particular thread, not the whole program.

The architecture provides a variety of synchronization facilities to allow threads to cooperate in accessing shared data. The synchronization facilities include mutual exclusion (mutex) locks, condition variables and semaphores. For example, a thread that wants to update a variable might block waiting for a mutual exclusion lock held by another thread that is already updating it. To support different frequencies of interaction and different degrees of concurrency, several synchronization mechanisms with different semantics are provided.

As shown in Figure 1, threads in different processes can synchronize with each other via synchronization variables placed in shared memory, even though the threads in different processes are generally invisible to each other. Synchronization variables can also be placed in files and have lifetimes beyond that of the creating process. For example, a file can be created that contains data base records. Each record can contain a mutual exclusion lock variable that controls access to the associated record. A process can map the file and a thread within it can obtain the lock associated with a particular record that is to be modified. When the modification is complete the thread can release the lock and unmap the file. Once the lock has been acquired, if any thread within any process mapping the file attempts to acquire the lock that thread will block until the lock is released.

Lightweight processes

Threads are an appropriate paradigm for most programs that wish to exploit parallel hardware or express concurrent program structure. For those situations that require more control over how the program is mapped onto parallel hardware, and to optimize the costs of concurrent execution and synchronization, a second interface is defined.

In the SunOS multi-thread architecture, a UNIX process consists mainly of an address space and a set of lightweight processes (LWPs[1]) that share that address space. Each LWP can be thought of as a virtual CPU which is available for executing code or system calls. Each LWP is separately dispatched by the kernel, may perform independent system calls, incur independent page faults, and may run in parallel on a multiprocessor. All the LWPs in the system are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority.

Threads are implemented using LWPs. Threads are actually represented by data structures in the address space of a program. LWPs within a process

[1] The LWPs in this document are fundamentally different than the LWP library in SunOS 4.0. Lack of imagination and a desire to conform to generally accepted terminology lead us to use the same name.

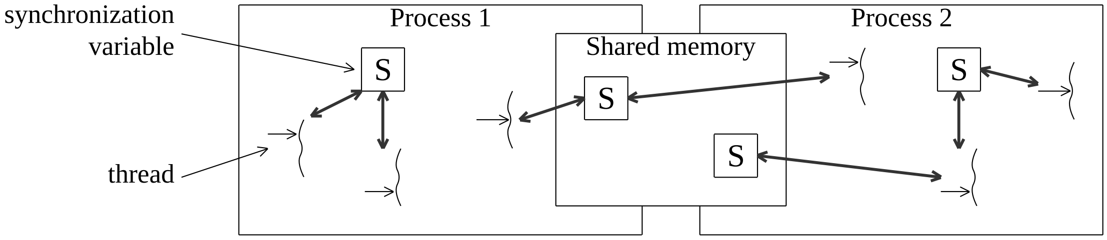

Figure 1: Synchronization variables

execute threads as shown in Figure 2. An LWP chooses a thread to run by locating the thread state in process memory (a). After loading the registers and assuming the identity of the thread, the LWP executes the thread's instructions (b). If the thread cannot continue, or if other threads should be run, the LWP saves the state of the thread back in memory (c). The LWP can now select another thread to run (d).

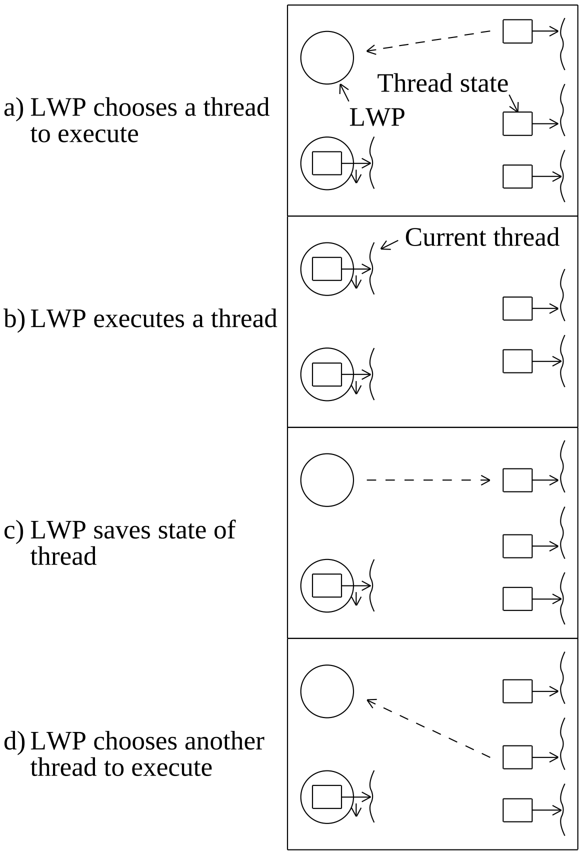

Figure 2: LWPs running threads

When a thread needs to access a system service by performing a kernel call, taking a page fault, or to interact with threads in other processes, it does so using the LWP that is executing it. The thread needing the system service remains bound to the LWP executing it until the system call is completed. If a thread needs to interact with other threads in the same process, it can do so without involving the operating system. As Figure 2 shows, switching from one thread to another occurs without the kernel knowing it. Much as the UNIX `stdio` library routines (such as `fopen()` and `fread()`) are implemented using the UNIX system calls (`open()` and `read()`), the thread interface is implemented using the LWP interface, and for many of the same reasons.

An LWP may also have some capabilities that are not exported directly to threads, such as a special scheduling class. A programmer can take advantage of these capabilities while still retaining use of all the thread interfaces and capabilities (e.g. synchronization) by specifying that the thread is to remain permanently bound to an LWP.

Threads are the primary interface for application parallelism. Few multi-threaded programs will use the LWP interface directly, but it is sometimes important to know that it is there. Some languages define concurrency mechanisms that are different from threads. An example is a Fortran compiler that provides loop level parallelism. In such cases, the language library may implement its own notion of concurrency using LWPs. Most programmers can program using the threads interface and let the library take care of mapping threads onto the kernel primitives. The decision of how many LWPs should be created to run the threads can be left to the library, or may be specified by the programmer.

Why have both threads and LWPs?

One might wonder why it is necessary to have two interfaces that are so similar. The multi-threaded architecture must meet a variety of different expectations. Some programs have large amounts of logical parallelism, such as a window system that provides each widget with one input handler and one output handler. Other programs need to map their parallel computation onto the actual number of processors available. In both cases, programs want to easily have complete access to the system services.

Threads are implemented by the library and are not known to the kernel. Thus, threads may be created, destroyed, blocked, activated, etc., without involving the kernel. LWPs are implemented by the kernel. If a thread wants to read from a file, the kernel needs to be able to switch to other processing when the LWP blocks in the file system code waiting for the I/O to finish. The kernel has to preserve the state of the read operation and continue it when the I/O interrupt arrives. However, if each thread were always known to the kernel, it would have to allocate kernel data structures for each one and get involved in context switching threads even though most thread interactions involve threads in the same process. In other words, kernel-supported parallelism (LWPs) is relatively expensive compared to threads. Having all threads supported directly by the kernel would cause applications such as the window system to be much less efficient. Although the window system may be best expressed as a large number of threads, only a few of the threads ever need to be active (i.e. require kernel resources, other than virtual memory) at the same instant.

Sometimes having more threads than LWPs is a disadvantage. A parallel array computation divides the rows of its arrays among different threads. If there is one LWP per processor, but multiple threads per LWP, each processor would spend overhead switching between threads. It would be better to know that there is one thread per LWP, divide the rows among a smaller number of threads, and reduce the number of thread switches. By specifying that each thread is permanently bound to its own LWP, a programmer can write thread code that is really LWP code, much like locking down pages turns virtual memory into real memory.

A mixture of threads that are permanently bound to LWPs and unbound threads is also appropriate for some applications. An example of this would be some real-time applications that want some threads to have system-wide priority and real-time scheduling, while other threads can attend to background computations.

By defining both levels of interface in the architecture, we make clear the distinction between what the programmer sees and what the kernel provides. Most programmers program using threads and do not think about LWPs. When it is appropriate to optimize the behavior of the program, the programmer has the ability to tune the relationship between threads and LWPs. This allows programmers to structure their application assuming extremely lightweight threads while bringing the appropriate degree of kernel-supported concurrency to bear on the computation. To some degree, a threads programmer can think of LWPs used by the application as the degree of real concurrency that the application requires.

Summary

Figure 3 shows all of the pieces in one diagram. The assignment of threads to LWPs is either controlled by the threads package or is specified by the programmer. The kernel sees LWPs and may schedule these on the available processors.

Process 1 is the traditional UNIX process with a single thread attached to a single LWP. Process 2 has threads multiplexed on a single LWP as in typical coroutine packages, such as SunOS 4.0 `liblwp`. Process 3 through 5 depict new capabilities of the SunOS multi-thread architecture. Process 3 has several threads multiplexed on a lesser number of LWPs. Process 4 has its threads permanently bound to LWPs. Process 5 shows all the possibilities; a group of threads multiplexed on a group of LWPs, while having threads bound to LWPs. In addition, the process has asked the system to bind one of its LWPs to a CPU. Note that the bound and unbound threads can still synchronize with each other both within the same process and between processes in the usual way.

Design Goals

Having described the overall thread model and language used to describe the model, we can now describe the goals of the architecture. The following goals are approximately in order of importance.

- The architecture should describe structures and mechanisms that work among threads in the same program, between different programs (processes), and between processors (whether the processors are executing in the same or different processes).
- The architecture should support threads that are as cheap as possible. Threads within a program should not be forced to cross protection boundaries to synchronize or context switch, nor should threads require excessive kernel resources.
- The architecture must support both multiprocessor and uniprocessor implementations.
- All current UNIX semantics should be provided in user programs and libraries wherever possible. The degenerate case of a process being constructed of an address space and one lightweight process must provide complete UNIX semantics.
- Different lightweight processes should be able to do independent, simultaneous system calls.
- The mechanisms defined in the system should be simple and fundamental. For example, there should be a method of using threads that does not force the threads library to use `malloc()`. This prevents interference with other application or language run-time system memory allocators.

The following are not exactly goals, but are principles that were used to help design the architecture.

- Per-thread state must be kept to a minimum. Each additional piece of state above the minimum necessary must be justified so as not to add undue

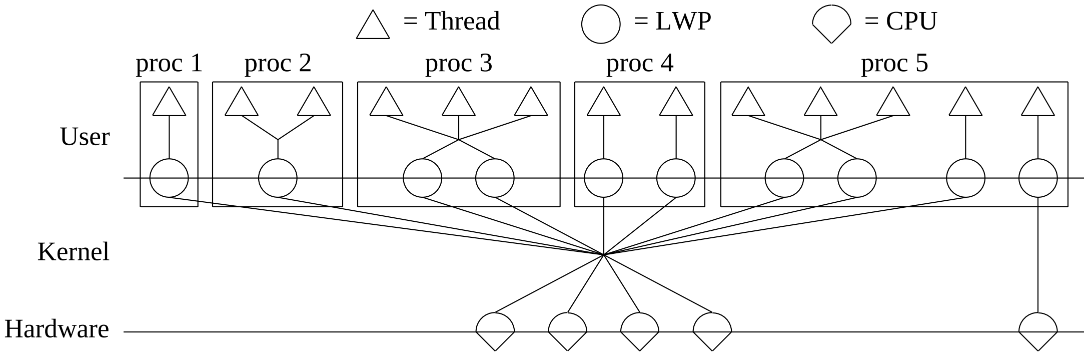

Figure 3: Multi-thread architecture examples

''weight'' to a thread.

- An address space with one thread (and therefore one lightweight process) should behave like a standard UNIX process; the addition of a new thread (and possibly a lightweight process) that does not interact with the first thread should not change the behavior of the first thread.
- The opportunity should be provided for different implementations. For example, by allowing but not requiring threads to share the whole address space, by allowing but not requiring threads to be multiplexed on lightweight processes, and by allowing but not requiring synchronization primitives to be executed in user mode.
- Wherever possible, equivalent semantics to UNIX should be provided, even if that doesn't seem like the best way to implement the function. Alternative operations should be added to do things the ''right'' way.
- The process is the unit of work. Threads are resources of the process and are applied to the work of the process in much the same way as file descriptors. For example, threads in other processes are invisible.

Multi-threaded Operations

System calls

The base programmer interface for functions other than those relating to threads or multi-threading is the System V Interface Definition, Third Edition (SVID3). In general, most current UNIX system calls remain unchanged. The main difference is that system calls that block do so to the lightweight process and therefore to the thread that executes them. However programmers must understand that threads and LWPs share almost all the programmer visible process resources such as address space and file descriptor table. This can lead to several potential trouble spots:

- Because file descriptors are shared, if one thread closes a file, it is closed for all threads. Care must be taken with seeks before reads or writes, because another thread could change the seek position before the read or write (this is similar to what happens now when a parent and child process share a file descriptor).
- There is only one working directory for each process. If one thread changes the working directory, it is changed for all of them.
- There is only one set of user and group IDs for each process, so if one thread changes one of these, it is changed for all of them. Because these can change concurrently, the kernel must ensure that the values are sampled, atomically, only once per system call.
- Multiple threads may manipulate the shared address space at the same time via `mmap()`, `brk()`, or `sbrk()`.
- Programs must not make assumptions about ''the'' stack, because there may be several.

Threads and lightweight processes

One lightweight process is created by the kernel when a program is started, and it starts executing the thread compiled as the main program. Additional threads are created by calls to the library specifying a procedure for the new thread to execute and a stack area for it to use.

Depending on the implementation, on the library, or on programmer supplied parameters, a thread may be associated with the same or different lightweight processes during its lifetime. There may be a one-to-one relationship between threads and lightweight processes, or one or more lightweight processes may be multiplexed by the thread library among a set of threads. Ordinarily, a thread cannot tell what the relationship between lightweight processes and threads is, although for performance reasons, or to avoid some deadlocks, a program may require there to be more or fewer lightweight processes.

When a thread executes a kernel call, it remains bound to the same lightweight process for the duration of the kernel call. If the kernel call blocks, that thread and its lightweight process remain blocked. Other lightweight processes may execute other threads in that program, including performing other kernel calls. The same principle applies to page faults.

There is no system-wide name space for threads or lightweight processes. Thus, for example, it is not possible to direct a signal to a particular lightweight process from outside a process or to know which lightweight process sent a particular message.

Thread state

The following state is unique to each thread:

- Thread ID
- Register state (including PC and stack pointer)
- Stack
- Signal mask
- Priority
- Thread-local storage

All other process state is shared by the threads within the process.

Thread-local storage

Threads have some private storage (in addition to the stack) called thread-local storage. Most variables in the program are shared among all the threads executing it, but each thread has its own copy of thread-local variables. Conceptually, thread-local storage is unshared, statically allocated data. The C library variable `errno` is a good example of a variable that should be placed in thread-local storage. This allows each thread to reference `errno` directly and it allows threads to interleave execution without fear of corrupting `errno` in other threads. Thread-local storage is potentially expensive to access, so it should be limited to the essentials, such as supporting older, non-reentrant interfaces.

It is implementation-dependent whether or not a thread is absolutely prevented from accessing another thread's stack or thread-local variables, but a correct thread must never attempt to do so.

Thread-local storage is obtained via a new `#pragma`, supported by the compiler and linker. The contents of thread-local storage are zeroed, initially; static initialization is not allowed. In C, thread-local storage for `errno` would be declared as follows:

```
#pragma unshared errno
extern int errno;
```

The size of thread-local storage is computed by the run-time linker at program start time by summing the thread-local storage requirements of the linked libraries. This prevents the exact size of thread-local storage from being part of the library interface. Once the size is computed it is not changed (e.g. by future dynamic linking in the process). This restriction prevents the size of thread-local storage from changing once a thread is started. Thus thread-local storage requirements are known at thread startup time and can be allocated as part of stack storage. More dynamic mechanisms (such as POSIX thread-specific data [POSIX 1990]) can be built using thread-local storage.

Thread synchronization

Threads synchronize with each other using facilities supplied by the implementation that present a standard set of semantics. The following synchronization types are supported:

- Mutual exclusion (mutex) locks
- Counting semaphores
- Condition variables
- Multiple readers, single writer locks

The architecture allows a range of implementations of each synchronization type to be supported. For example, mutual exclusion locks may be implemented as spin locks, sleep locks, or adaptive locks, etc.

These facilities use synchronization variables in memory. The variables may be statically allocated and/or at fixed addresses (within the alignment constraints of the variable). The programmer may choose the particular implementation variant of the synchronization semantic at the time the variable is initialized. If the variable is initialized to zero, a default implementation is used.

Synchronization variables may also be placed in memory that is shared between processes. The programmer can select an implementation variant of each synchronization type that allows the variable to synchronize threads in the processes sharing the variable. Synchronization primitives apply to the shared variable as part of the underlying mapped object. In other words, synchronization variables may be shared between processes even though they are mapped at different virtual addresses.

Synchronization variables that are not in shared memory are completely unknown to the kernel. Synchronization variables that are in shared memory or in files are also unknown to the kernel unless a thread is blocked on them. In the latter case the thread is temporarily bound to the LWP that is blocked by the kernel, as in a system call.

Signal handling

Each thread has its own signal mask. This permits a thread to block some signals while it uses state that is also modified by a signal handler. All threads in the same address space share the set of signal handlers, which are set up by `signal()` and its variants, as usual. If desired, it would be possible for a particular application to implement per-thread signal handlers using the per-process signal handlers. For example, the signal handler can use the ID of the thread handling the signal as an index into a table of per-thread handlers. If the threads library were to implement per-thread signal handlers it must decide on the correct semantics when several threads have different combinations of signal handlers, `SIG_IGN`, and `SIG_DFL`. In addition, all threads would be burdened with the handler state. For this reason, we felt that library support of per-thread signal handlers was overly complex and possibly confusing to the application programmer.

If a signal handler is marked `SIG_DFL` or `SIG_IGN` the action on receipt of the signal (exit, core dump, stop, continue, or ignore) affects all the threads in the receiving process.

Signals are divided into two categories: traps and interrupts. Traps (e.g. `SIGILL`, `SIGFPE`, `SIGSEGV`) are signals that are caused synchronously by the operation of a thread, and are handled only by the thread that caused them. Several threads in the same address space could conceivably generate and handle the same kind of trap simultaneously. Interrupts (e.g. `SIGINT`, `SIGIO`) are signals that are caused asynchronously by something outside the process. An interrupt may be handled by any thread that has it enabled in its signal mask. If more than one thread is enabled to receive the interrupt, only one is chosen. Thus, several threads can be in the process of handling the same kind of signal simultaneously. If all threads mask a signal, it will pend on the process until a thread unmasks that signal. As in single-threaded processes, the number of signals received by the process is less than or equal to the number sent.

For example, an application can enable several threads to handle a particular I/O interrupt. As each new interrupt comes in, another thread is chosen to handle the signal until all the enabled threads are active. New signals will then pend waiting for threads to complete processing and re-enable signal handling.

Threads may send signals to other threads within the process via a new interface; `thread_kill()`. In this case the signal behaves like a trap and can be handled only by the specified thread. The programmer may also send a signal to all the threads via `sigsend()`. A thread cannot send a signal to a specific thread in another process because threads in other processes are invisible.

Threads that are not bound to LWPs may not use alternate signal stacks. Adding alternate signal stacks to the unbound thread state was deemed too expensive to implement because this would require a system call to establish the alternate stack for each context switch of a thread requiring it. Threads bound to LWPs may use alternate stacks as this state is associated with each LWP.

Non-local goto

`setjmp()` and `longjmp()` work only within a particular thread. In particular, it is an error for a thread to `longjmp()` into another thread. Therefore, it is possible to `longjmp()` from a signal handler only when the `setjmp()` was executed by the thread that is handling the signal.

Thread interfaces

Most of the interfaces available to threads are those that are available to UNIX processes in single-threaded UNIX. As mentioned above, some of those interfaces have different implications in a multi-threaded environment, but the intent is to provide ‘‘UNIX semantics’’ as the ordinary programming model. This section describes some of the additional interfaces needed to create and manage threads.

The syntax of the interfaces is shown in Figure 4.

Thread creation

`thread_create()` creates a new thread. If `stack_addr` is not `NULL`, `stack_size` bytes of memory starting at `stack_addr` are used for the thread stack. In this case any thread-local storage is also placed on the stack so as not to interfere with stack growth. This allows a language run-time library to control thread storage without interference with its memory allocator. It is machine dependent whether the initial stack pointer is at higher or lower addresses in the specified stack. If `stack_addr` is `NULL` the stack is allocated from the heap. If `stack_size` is not zero the stack will be of the specified size. Otherwise a default stack size is used. Zeroed thread-local storage is also allocated to the thread. `thread_create()` returns the ID of the new thread. The thread IDs have meaning only within a process. The initial thread priority and signal mask is set to the same values as its creator. When the new thread is started, it begins execution by a procedure call to `func(arg)`. If `func` returns, the thread exits (calls `thread_exit()`). The `flags` argument provides the following (or’able) options:

`THREAD_STOP`
: The thread is to be immediately suspended after it is created. The thread will not run until another thread executes `thread_continue()` to start it. If `THREAD_STOP` is not specified, the thread is immediately runnable.

`THREAD_NEW_LWP`
: A new LWP is created along with the thread. The new LWP is added to the pool of LWPs used to execute threads.

`THREAD_BIND_LWP`
: A new LWP is created and the new thread is permanently bound to it.

`THREAD_WAIT`
: Specifies that another thread will eventually wait for this thread to exit. This also means that the thread ID of a thread created with `THREAD_WAIT` will not be reused until the waiting thread returns. If the thread is not created with `THREAD_WAIT`, the thread ID may be reused at any time after the thread exits.

Thread concurrency control

`thread_setconcurrency()` sets the degree of real concurrency (i.e. the number of LWPs) that unbound threads in the application require to `n`. The number of LWPs permanently bound to threads is not included in `n`. If `n` is zero (the default), the library automatically creates as many LWPs for use in scheduling unbound threads as required to avoid deadlock. This number can be incremented by creating a thread with the `THREAD_NEW_LWP` flag. If `n` is less than the current maximum, LWPs are removed from the pool. `thread_setconcurrency()` guarantees only that this degree of concurrency is available to application threads. The actual number of LWPs employed by the library at any one time may vary.

The number of LWPs automatically created by the library (`n` = 0) is sufficient to avoid deadlock, but it may not be enough to avoid poor performance; the library may create too few or too many LWPs. The programmer may tune the number of LWPs by creating threads with the `THREAD_NEW_LWP` flag or using `thread_setconcurrency()` as required by the application.

Thread termination

`thread_exit()` terminates the current thread and deallocates thread resources allocated by the threads package.

Waiting for threads

`thread_wait()` blocks until the specified thread exits. It is an error to wait for a thread that was created without the `THREAD_WAIT` attribute, to wait for the current thread, or to have multiple `thread_wait()`s on the same thread. If `thread_id` is `NULL`, then any thread marked `THREAD_WAIT` that exits causes `thread_wait()` to return. If a stack was supplied by the programmer when the thread was created, it may be reclaimed when `thread_wait()` returns successfully. `thread_wait()` returns the ID of the thread that exited if the wait is successful. After

```
thread_id_t
thread_create(char *stack_addr,
        unsigned int stack_size,
        void (*func)(),
        void *arg,
        int flags);

int
thread_setconcurrency(int n);

void
thread_exit();

thread_id_t
thread_wait(thread_id_t thread_id);

thread_id_t
thread_get_id();

int
thread_sigsetmask(int how,
        sigset_t *set,
        sigset_t *oset);

int
thread_kill(thread_id_t thread_id,
        int sig);

int
thread_stop(thread_id_t thread_id);

int
thread_continue(thread_id_t thread_id);

int
thread_priority(thread_id_t thread_id,
        int priority);

void
mutex_init(mutex_t *mp,
        int type,
        void *arg);

void
mutex_enter(mutex_t *mp);

void
mutex_exit(mutex_t *mp);

int
mutex_tryenter(mutex_t *mp);

void
cv_init(condvar_t *cvp,
        int type,
        void *arg);

void
cv_wait(condvar_t *cvp,
        mutex_t *mutexp);

void
cv_signal(condvar_t *cvp);

void
cv_broadcast(condvar_t *cvp);

void
sema_init(sema_t *sp,
        unsigned int count,
        int type,
        void *arg);

void
sema_p(sema_t *sp);

void
sema_v(sema_t *sp);

int
sema_tryp(sema_t *sp);

void
rw_init(rwlock_t *rwlp,
        int type,
        void *arg);

void
rw_enter(rwlock_t *rwlp,
        rw_type_t type);

void
rw_exit(rwlock_t *rwlp);

int
rw_tryenter(rwlock_t *rwlp,
        rw_type_t type);

void
rw_downgrade(rwlock_t *rwlp);

int
rw_tryupgrade(rwlock_t *rwlp);
```

Figure 4: Thread interface functions

thread_wait() returns successfully, the returned thread_id is unusable in any subsequent thread operation.

An alternate interface for this function is waitid() with id_type equal to one of the following:

P_THREAD
: waitid() waits for the thread specified by id.

P_THREAD_ALL
: waitid() waits for any thread marked THREAD_WAIT.

The exit status of a thread is always zero.

Thread identification

thread_get_id() returns the thread ID of the caller.

Thread signal mask

thread_sigsetmask() or sigprocmask() sets the thread's signal mask.

Thread signaling

thread_kill() causes the specified signal to be sent to the specified thread. An alternate interface for this function is sigsend() with id_type equal to one of the following:

P_THREAD
: sig is sent to the thread within the process specified by id.

P_THREAD_ALL
: sig is sent to all the threads within the process.

Thread execution control

thread_stop() prevents the specified thread from running. If thread_id is NULL then the current thread is immediately stopped. thread_continue() initially starts a thread or restarts a thread after thread_stop(). The effect of thread_continue() may be delayed, but thread_stop() does not return until the specified thread is stopped.

Thread priority control

thread_priority() sets the priority of the specified thread. If thread_id is NULL the current thread is used. The priority must be greater than or equal to zero. Increasing the specified priority gives increasing scheduling priority. The old priority is returned. If the specified thread is not running then it may or may not execute immediately even though its new priority is greater than a currently executing thread.

Thread synchronization

The thread synchronization facilities are designed to synchronize threads both within a process and between processes. When a synchronization variable is initialized, the programmer must specify whether the synchronization variable is to be shared between processes. The programmer can usually also specify other variants such as extra debugging, spin waiting, etc. The programmer may bitwise-or THREAD_SYNC_SHARED into the variant type to specify that the variable is to be shared between processes.

Any synchronization variable that is statically or dynamically allocated as zero may be used immediately without further initialization, and provides the default implementation variant in the default initial state. A dynamic initialization with an implementation variant type of zero also specifies the default implementation variant.

Mutex locks

Mutex locks provide simple mutual exclusion. They are low overhead in both space and time and are therefore suitable for high frequency usage. Mutex locks are strictly bracketing in that it is an error for a thread to release a lock not held by the thread. Mutex locks are used to prevent data inconsistencies in critical sections of code. They may also be used to preserve code that is single threaded.

mutex_enter() acquires the lock, potentially blocking if it is already held. mutex_exit() releases the lock, potentially unblocking a waiter. mutex_tryenter() acquires the lock if it is not already held. mutex_tryenter() can be used to avoid deadlock in operations that would normally violate the lock hierarchy.

Condition variables

Condition variables are used to wait until a particular condition is true. Condition variables must be used in conjunction with a mutex lock. This implements a typical monitor.

cv_wait() blocks until the condition is signaled. It releases the associated mutex before blocking, and reacquires it before returning. Since the reacquiring of the mutex may be blocked by other threads waiting for the mutex, the condition that caused the wait must be re-tested. Thus, typical usage is:

```
mutex_enter(&m);
...
while (some_condition) {
        cv_wait(&cv, &m);
}
...
mutex_exit(&m);
```

This allows the condition to be a complicated expression, as it is protected by the mutex. There is no guaranteed order of acquisition if more than one thread blocks on the condition variable.

`cv_signal()` wakes up one of the threads blocked in `cv_wait()`. `cv_broadcast()` wakes up all of the threads blocked in `cv_wait()`. Since `cv_broadcast()` causes all threads blocking on the condition to re-contend for the mutex, it should be used with care. For example, it is appropriate to use `cv_broadcast()` to allow threads to contend for variable amounts of resources when resources are released.

Semaphores

The semaphore synchronization facilities provide classic counting semaphores. They are not as efficient as mutex locks, but they need not be bracketed so that they may be used for asynchronous event notification (e.g. in signal handlers). They also contain state so they may be used asynchronously without acquiring a mutex as required by condition variables.

`sema_p()` decrements the semaphore, potentially blocking the thread. `sema_v()` increments the semaphore, potentially unblocking a waiting thread. `sema_tryp()` decrements the semaphore if blocking is not required.

Multiple readers, single writer locks

Multiple readers, single writer locks allow many threads simultaneous read-only access to an object protected by this lock simultaneously. It allows only one thread to access an object for writing at any one time, and excludes any readers. A good candidate for a multiple readers, single writer lock is an object that is searched more frequently than it is changed. For brevity this type of lock is also known as a readers/writer lock.

`rw_enter()` attempts to acquire a reader or writer lock. `type` may be one of the following:

`RW_READER` Acquire a readers lock.

`RW_WRITER` Acquire a writer lock.

`rw_exit()` releases a readers or writer lock. `rw_tryenter()` acquires a readers or writer lock if doing so would not require blocking. `rw_downgrade()` atomically converts a writer lock into a reader lock. Any waiting writers remain waiting. If there are no waiting writers it wakes up any pending readers. `rw_tryupgrade()` attempts to atomically convert a reader lock into a writer lock. If there is another `rw_tryupgrade()` in progress or there are any writers waiting, it returns a failure indication.

Lightweight process state

A lightweight process consists of a data structure in the kernel used for processor scheduling, page fault handling, and kernel call execution. It also contains state that is private to the LWP and an association with a process (address space). The following programmer-visible state is maintained by the kernel and is unique to each LWP within a process:

- LWP ID
- Register state (including PC and stack pointer)
- Signal mask
- Alternate signal stack and masks for alternate stack disable and onstack
- User and user+system virtual time alarms
- User time and system CPU usage
- Profiling state
- Scheduling class and priority

All other process state is shared by the LWPs within the process.

Note that even though the CPU usage, virtual time alarms, and alternate signal stack are available to each LWP, this state is not kept for each thread that is multiplexed on LWPs. Threads that require this state must be bound to an LWP. Whether the LWP state includes a separate stack area known to the kernel or not is implementation dependent. Of course, the lightweight process runs with a stack.

Signals

A new signal, `SIGWAITING`, is sent to the process when all its LWPs are waiting for some indefinite, external event (e.g. in `poll()`). The default handling for `SIGWAITING` is to ignore it. The threads package can use the receipt of `SIGWAITING` to cause extra LWPs to be created as required to avoid deadlock. This is similar in functionality to the architecture described in [Anderson 1990].

While `SIGWAITING` is sent for ‘‘indefinite’’ waits, supposedly short term blocking for things like page faults or file system I/O may take a long time relative to the speed of the CPUs. It may be desirable to define an alternate signal that is sent in these cases.

Time, interval timers, and profiling

There is only one real-time interval timer per process, so it delivers one signal to an address space when it reaches the specified time interval. Library routines may implement multiple per-thread timers using the per-address space timer when that functionality is required. Each LWP has two private interval timers; one decrements in LWP user time and the other decrements in both LWP user time and when the system is running on behalf of the LWP. When these interval timers expire either `SIGVTALRM` or `SIGPROF`, as appropriate, is sent to the LWP that owns the interval timer.

Profiling is enabled for each LWP individually. Each LWP can set up a separate profiling buffer, but it may also share one if accumulated information is desired. Profiling information is updated at each clock tick in LWP user time. The state of profiling is inherited from the creating LWP.

Resource usage

The resource limits set limits on the resource usage of the entire process (i.e. the sum of the resource usage of all the LWPs in the process). When a soft resource limit has been exceeded, the LWP that exceeded the limit is sent the appropriate signal. The sum of the resource usage (including CPU usage) for all LWPs in the process is available via `getrusage()`.

Process creation and destruction

The `fork()` system call attempts to duplicate the existing UNIX semantics. It duplicates the address space and creates the same LWPs in the same states as in the original. This duplicates the threads in the original process. Calling `fork()` may cause interruptible system calls to return `EINTR` when the calls are made by any LWP (thread) other than the one calling `fork()`.

A new system call, `fork1()`, causes the current thread/LWP to fork, but the other threads and LWPs that existed in the original process are not duplicated in the new process. `fork1()` is defined as follows:

```
int fork1();
```

The return values are similar to `fork()`.

Both the `exit()` and `exec()` system calls work as usual, except that they destroy all the LWPs in the address space. Both calls block until all the LWPs (and therefore all active threads) are destroyed. When `exec()` rebuilds the process, it creates a single LWP. The process startup code then builds the initial thread.

Why have both `fork()` *and* `fork1()`*?*

UNIX `fork()` seems to have two generic uses; to duplicate the entire process (the BSD `dump` program uses this technique), or to create a new process in order to set up for `exec()`. For the latter purpose, `fork1()` is much more efficient because there is no need to duplicate all the LWPs. There are, however, dangers to using `fork1()`. First, since threads are maintained by the threads library as data structures, the threads library must take care that after `fork1()` only the issuing thread remains in the new address space, which is a duplicate of the old one. Secondly, the programmer must be careful to call only functions that do not require locks held by threads that no longer exist in the new process. This can be difficult to determine as libraries can create hidden threads. Lastly, locks that are allocated in memory that is sharable (i.e. `mmap()`'ed with the `MAP_SHARED` flag) can be held by a thread in both processes, unless care is taken to avoid this. The latter problem can also arise with `fork()`.

Having `fork()` completely duplicate the process is the semantic that is most similar to the single-threaded `fork()`. It allows both generic uses and there are fewer pitfalls for the programmer. Having `fork1()`, which forks only one thread, permits optimized `fork()` and immediate `exec()` (e.g. `system()`).

Scheduling

LWPs (and bound threads) can change their scheduling class and class priority via the `priocntl()` system call. A new scheduling class for ‘‘gang’’ scheduling is available for implementations of fine grain parallelism. The LWP may also ask to be bound to a CPU, depending on the scheduling class.

Debugging

The **/proc** file system has been extended to reflect the changes to the process model required by the addition of multi-threading at the process level. Of necessity, a kernel process model interface can provide access only to kernel-supported threads of control, namely LWPs. Debugger control of library threads is accomplished by cooperation between the debugger and the threads library, with the aid of the **/proc** file system to control the kernel-supported LWPs.

The details of the `proc` file system and some of the enhancements for multi-threading support can be found in [Faulkner 1991].

Performance

All the performance numbers in this section were obtained on a SPARCstation 1+ (Sun 4/65), which is a 25Mhz SPARC platform. The measurements were made using the built-in microsecond resolution real-time timer. The numbers reflect an untuned prototype system.

Thread creation time

The first measurement is for thread creation time. It measures the time consumed to create a thread using a default stack that is cached by the threads package. The measured time only includes the actual creation time, it does not include the time for the initial context switch to the thread. The results are shown in Figure 5. The ratio column gives the ratio of the creation time in that row to the creation time in the previous row.

	Time (usec)	ratio
Unbound thread create	56	-
Bound thread create	2327	42

Figure 5: Thread creation time

Measurements were taken for creating both bound and unbound threads. Bound thread creation involves calling the kernel to also create an LWP to run it. Unbound thread creation is done without kernel involvement.

Thread synchronization time

The second measurement is for thread synchronization time. It measures the time it takes for two threads to synchronize with each other using two synchronization variables, as shown below:

```
sema_t s1, s2;

thread1()
{
        ...
        start_timer();
        sema_v(&s1);
        sema_p(&s2);
        t = end_timer();
        ...
}

thread2()
{
        ...
        sema_p(&s2);
        sema_v(&s1);
        ...
}
```

The numbers presented in Figure 6 are the results of the above measurement divided by two, since there are actually two synchronizations involved. The ratio column gives the ratio of the synchronization time in that row to the synchronization time in the previous row.

	Time (usec)	ratio
Setjmp/longjmp	59	-
Unbound thread sync	158	2.7
Bound thread sync	348	2.2
Cross process thread sync	301	.86

Figure 6: Thread synchronization time

The first measurement is a simple routine that does a `setjmp()` and `longjmp()` to itself. It is presented as a baseline for thread switching time. The next two measurements are for unbound and bound threads synchronizing within a process. The last measurement is for threads in two different processes synchronizing through a file in shared memory.

Comparison with other Thread Models

This section addresses the similarities and differences between the SunOS multi-thread (MT) architecture and other commercially available multi-thread interfaces. Instead of comparing procedural interfaces, the discussions concentrate on comparing and contrasting architectural issues. The comparisons underscore what we believe are the key differences rather than being comprehensive.

Mach Release 2.5 C Threads

Mach Release 2.5 C Threads [Cooper 1990], [Tevanian 1987] exemplifies a thread interface that provides the programmer with the means to express concurrency, independent of the underlying system support. While this is a desirable trait, Mach 2.5 C Threads does not acknowledge the existence of a second layer of abstraction (i.e. LWPs) and therefore does not allow the programmer to control the degree of kernel resources it uses. In many useful applications the programmer must know and manipulate the degree of actual kernel resources required. For example, a window system programmer must know that extremely lightweight threads are available, since a window system may use thousands. A micro-tasking Fortran run-time library relies on kernel-supported threads that are scheduled on processors as a group. Database programmers may require a mixture of the two situations. In addition, there may be aspects to kernel-supported threads that are too ‘‘heavyweight’’ to export to lightweight threads (e.g. virtual time) and are required by some applications.

In Mach 2.5, C Threads libraries have been constructed that map threads directly to kernel-supported threads or multiplex threads on kernel-supported threads, but one application cannot have both types at the same time. In addition, there can be no direct access to ‘‘heavyweight’’ features of kernel-supported threads since that would allow only a one-to-one mapping between threads and kernel-supported threads.

Newer versions of Mach [Golub 1990] have corrected some of these deficiencies by extending the C Threads interface to provide a two-level model similar to ours. In the new library, C Threads are multiplexed on Mach kernel threads. In addition, new C Threads interfaces allow C Threads to bind to Mach kernel threads.

The main difference between the C Threads synchronization primitives and the SunOS MT architecture primitives is the scope of operation. C Threads does not explicitly support the use synchronization variables allocated in `mmap`'ed memory even though Mach virtual memory supports the sharing of memory between tasks. The SunOS MT architecture supports this and also allows the placement of synchronization variables in files to control access to the file data, and having the lifetime of such synchronization variables be greater than that of the creating process.

C Threads supports per-process signal state. There is no per-thread signal mask. There is no way for a thread to control when it can handle a signal except by preventing all the threads in a process from handling it. When a particular thread is in a critical section of code with respect to the signal handler, it must block the interrupt for all threads. This can cause severe performance problems in heavily asynchronous applications. The alternate solution for C Threads is Mach IPC. Mach IPC, however, does not allow asynchronous interruption of a computation. For example, an application that creates a thread to perform some long computation may wish to terminate the computation regardless of results. There is no way to interrupt the computation unless it is coded to occasionally poll for IPC. This forces the programmer to change the computation code so that polling is done frequently enough to respond to a termination request but not so frequently as to slow down the computation.

Chorus

Chorus [Armand 1990] intentionally avoided user-level threads because of a perceived impact on real-time requirements. For example, the two levels of scheduling interfere with the requirement that the highest priority runnable thread is always allowed to run. SunOS meets this requirement by allowing a thread to bind to an LWP and thus achieve a system-wide scheduling priority. In addition, the bound thread can ask that the underlying LWP be made a member of a real-time scheduling class, which provides more exact scheduling control.

Chrous threads each have a signal mask and a vector of signal handlers. The effect of receipt of an asynchronously generated signal and combinations of catching, `SIG_DFL`, and `SIG_IGN` are computed. If one or more threads are catching the signal, it is delivered to all catching threads (broadcast delivery). Otherwise, if any thread has set the handler to `SIG_IGN`, the signal is discarded. Otherwise the default action is taken on the process. The main deficiency in this model is that broadcast delivery can cause ‘‘synchronization storms’’ when the handling threads try to synchronize. It also causes much extra work for the kernel. Lastly, broadcast makes the number of signals delivered to a process uncountable in a non-queuing signal implementation. For example, if several threads are waiting for a keyboard interrupt, and two are sent, some threads will receive two signals while others will receive one.

The per-thread signal handlers add some code modularity, at the cost of complexity in the handling of `SIG_DFL` and `SIG_IGN` as noted above. The modularity added is relatively minor because asynchronous signals are mostly controlled by the application, not the library. In addition, serial handling of the same signal within a thread is still a problem, just as it is in single-threaded UNIX.

University of Washington.

The variant of the Topaz [McJones 1989] operating system by the University of Washington [Anderson 1990] implements a portable threads interface with lightweight user-level threads that use kernel resources only as required. In most cases threads can synchronize without kernel involvement, while at the same time, I/O, page faults, and other blocking operations do not stop the entire process. This approach has the same advantages as our threads multiplexed on LWPs. However, programmer control over the use of kernel resources is not supported.

The main underlying difference between the University of Washington work and the SunOS MT architecture is that the University of Washington work uses lightweight ‘‘scheduler activations’’ that do upcalls into user space to give schedulable execution contexts to the threads package. An upcall by a new scheduler activation informs the threads package whenever a scheduler activation currently in use by the process blocks in the kernel. This gives the threads package the opportunity to schedule another runnable thread. This is similar to the function of the new `SIGWAITING` signal in our architecture. This signal also gives the threads library the opportunity to schedule a runnable thread by first creating a new LWP. The main difference is that the current definition of `SIGWAITING` is much more coarse than the way scheduler activations are used. The former is sent only when the LWP blocks in an indefinite wait. The latter is sent whenever the thread blocks in the kernel for any event. In the future, we plan to experiment with sending signals on ‘‘faster’’ events.

The University of Washington approach gives much finer-grained control over scheduling threads on processors, though it is not clear that this is an absolute requirement. In general, the SunOS MT architecture satisfies most of the requirements that motivated the University of Washington group. The critical observation made by both efforts was that the kernel need not be invoked for every thread operation.

POSIX P1003.4a

Comparison with POSIX P1003.4a Pthreads [POSIX 1990] is somewhat difficult at this time, as it is a moving target. Currently (pre Draft 10) it seems that the signal model is a direct superset of the SunOS model. In addition, there seems to be support for the two-level threads model in the scheduling interfaces. However, the interaction between synchronization variables and mapped files (P1003.4) is missing.

Sun LWP library.

The Sun LWP library [Kepecs 1985] supplied in SunOS 4.0 is a classic user-level-only threads package. It contained no explicit kernel support. Threads (called LWPs) synchronized with each other without kernel involvement. If an LWP called a blocking system call or took a page fault, the entire application blocked. This could be mitigated somewhat by using a non-blocking I/O library instead of the standard UNIX

I/O interfaces. The non-blocking I/O library uses kernel-supported asynchronous I/O facilities to mimic standard I/O interfaces and allows the package to switch LWPs when one blocked on an indefinite I/O. The application still blocked when a page fault was taken.

The SunOS multi-thread architecture completely supersedes this interface in functionality.

Summary

The SunOS multi-threading architecture provides the following advantages:

- The two level (threads and LWP) model allows the programmer to decouple logical program parallelism from the relatively expensive kernel-supported parallelism. Programmers can rely on the availability of extremely lightweight threads.
- The architecture allows the programmer to control the degree of real concurrency the application requires or allows the threads package to automatically decide this.
- The architecture has a uniform synchronization model between threads both inside and outside a process.
- The programmer can control the mapping of threads onto LWPs to achieve particular performance or functionality without leaving the threads model.
- The programmer can control the allocation of stacks and thread-local storage. This allows coexistence with different memory allocation models (e.g. garbage collection).
- A minimalist translation of the UNIX environment to threads allows higher-level interfaces such as POSIX Pthreads to be implemented on top of SunOS threads.

References

[Anderson 1990]
T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy, ‘‘Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism’’, Department of Computer Science and Engineering, University of Washington, Technical Report 90-04-02, April 1990.

[Armand 1990]
F. Armand, F. Herrmann, J. Lipkis, M. Rozier, ‘‘Multi-threaded Processes in Chorus/MIX’’, Proc. EUUG Spring 1990 Conference, Munich, Germany, April 1990.

[Cooper 1990]
E.C. Cooper, R.P. Draves, ‘‘C Threads’’, Department of Computer Science, Carnegie Mellon University, September 1990.

[Faulkner 1991]
R. Faulkner, R. Gomes, ‘‘The Process File System and Process Model in UNIX System V’’, Proc. 1991 USENIX Winter Conference.

[Golub 1990]
D. Golub, R. Dean, A. Florin, R. Rashid, ‘‘UNIX as an Application Program’’, Proc. 1990 USENIX Summer Conference, pp 87-95.

[Kepecs 1985]
J. Kepecs, ‘‘Lightweight Processes for UNIX Implementation and Applications’’, Proc. 1985 USENIX Summer Conference, pp 299-308.

[McJones 1989]
P.R. McJones and G.F. Swart, ‘‘Evolving the UNIX System Interface to Support Multithreaded Programs’’, Proc. 1989 USENIX Winter Conference, pp 393-404.

[POSIX 1990]
POSIX P1003.4a, ‘‘Threads Extension for Portable Operating Systems’’, IEEE.

[Tevanian 1987]
A. Tevanian, R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young, ‘‘Mach Threads and the UNIX Kernel: The Battle for Control’’, Proc. 1987 USENIX Summer Conference, pp University of California at Berkeley in 1984.