# Implementing Lightweight Threads

*D. Stein, D. Shah* – SunSoft, Inc.

## ABSTRACT

We describe an implementation of a threads library that provides extremely lightweight threads within a single UNIX process while allowing fully concurrent access to system resources. The threads are lightweight enough so that they can be created quickly, there can be thousands present, and synchronization can be accomplished rapidly. These goals are achieved by providing user threads which multiplex on a pool of kernel-supported threads of control. This pool is managed by the library and will automatically grow or shrink as required to ensure that the process will make progress while not using an excessive amount of kernel resources. The programmer can also tune the relationship between threads and kernel supported threads of control. This paper focuses on scheduling and synchronizing user threads, and their interaction with UNIX signals in a multiplexing threads library.

### Introduction

This paper describes a threads library implementation that provides extremely lightweight threads within a single UNIX process while allowing fully concurrent access to system resources. The threads are lightweight enough so that they can be created quickly, there can be thousands present, and synchronization can be accomplished rapidly. These goals are achieved by providing user threads which multiplex on a pool of kernel-supported threads of control. The implementation consists of a threads library and a kernel that has been modified to support multiple threads of control within a single UNIX process [Eykholt 1992].

The paper contains 7 sections: The first section is an overview of the threads model exported by the library. The second section is an overview of the threads library architecture. Readers familiar with the SunOS Multi-thread Architecture [Powell 1991] can skip the first two sections. The third section details the interfaces supplied by the kernel to support multiple threads of control within a single process. The fourth section details how the threads library schedules threads on kernel-supported threads of control. The fifth section details how the library implements thread synchronization. The sixth section details how the threads library implements signals even though threads are not known to the kernel. The final section briefly describes a way to implement a debugger that understands threads.

### Threads Model

A traditional UNIX process has a single thread of control. In the SunOS Multi-thread Architecture [Powell 1991], there can be more than one thread of control, or simply more threads, that execute independently within a process. In general, the number or identities of threads that an application process applies 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. For example, if one thread opens a file, another thread can read it. There is no system-enforced protection between threads.

There are a variety of synchronization facilities to allow threads to cooperate in accessing shared data. The synchronization facilities include mutual exclusion (mutex) locks, condition variables, semaphores and multiple readers, single writer (readers/writer) locks. The synchronization variables are allocated by the application in ordinary memory. Threads in different processes can also synchronize with each other via synchronization variables placed in shared memory or mapped files, even though the threads in different processes are generally invisible to each other. Such synchronization variables must be marked as being process-shared when they are initialized. Synchronization variables may also have different variants that can, for example, have different blocking behavior even though the same synchronization semantic (e.g., mutual exclusion) is maintained.

Each thread has a program counter and a stack to maintain of local variables and return addresses. Each thread can 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.

Each thread has its own signal mask. This permits a thread to block asynchronously generated signals while it accesses state that is also modified by a signal handler. Signals that are synchronously generated (e.g., SIGSEGV) are sent to the thread that caused them. Signals that are generated externally are sent to one of the threads within a process that has it unmasked. If all threads mask a signal, it is set pending on the process until a thread unmasks that signal. Signals can also be sent to particular threads within the same process. As in single-threaded processes, the number of signals received by the process is less than or equal to the number sent.

All threads within a process share the set of signal handlers[1]. If the handler is set to `SIG_IGN` any received signals are discarded. If the handler is set to `SIG_DFL` the default action (e.g., stop, continue, exit) applies to the process as a whole.

A process can fork in one of two ways: The first way clones the entire process and all of its threads. The second way only reproduces the calling thread in the new process. This last is useful when the process is simply going to call `exec()`.

### Threads Library Architecture

Threads are the programmer's interface for multi-threading. Threads are implemented by a dynamically linked library that utilizes underlying kernel-supported threads of control, called lightweight processes (LWPs)[2], as shown in Figure 1.

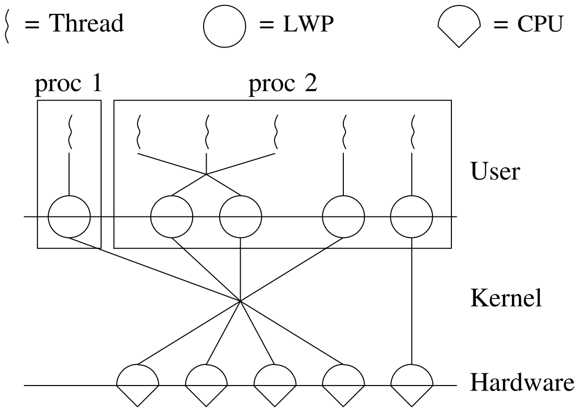


**Figure 1**: Multi-thread Architecture Examples

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 CPUs according to their scheduling class and priority.

The threads library schedules threads on a pool of LWPs in the process, in much the same way as the kernel schedules LWPs on a pool of processors[3]. Threads themselves are simply data structures and stacks that are maintained by the library and are unknown to the kernel. The thread library can cause an LWP to stop executing its current thread and start executing a new thread without involving the operating system. Similarly, threads may also be created, destroyed, blocked, activated, etc., without operating system involvement. LWPs are relatively much more expensive than threads since each one uses dedicated kernel resources. If the threads library dedicated an LWP to each thread as in [Cooper 1990] then many applications such as data bases or window systems could not use threads freely (e.g. one or more per client) or they would be inefficient. 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 a particular thread must be visible to the system. For example, when a thread must support real-time response (e.g., mouse tracking thread) and thus be scheduled with respect to all the other execution entities in the system (i.e., global scheduling scope). The library accommodates this by allowing a thread to be permanently bound to an LWP. Even when a thread is bound to an LWP, it is still a thread and can interact or synchronize with other threads in the process, bound or unbound.

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. A threads programmer can think of LWPs used by the application as the degree of real concurrency that the application requires.

### LWP interfaces

LWPs are like threads. They share most of the process resources. Each LWP has a private set of registers and a signal mask. LWPs also have

[1] Allowing each thread to have its own vector of signal handlers was rejected because it would add a non-trivial amount of storage to each thread and it allowed the possibility of conflicting requests by different threads (e.g., `SIG_DFL` and a handler) that would make both implementation and use difficult. In addition, per-thread can be programmed on top of a shared vector of handlers, if required.

[2] 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.

[3] Actually, the SunOS 5.0 kernel schedules kernel threads on top of processors. Kernel threads are used for the execution context underlying LWPs. See [Eykholt 1992].

attributes that are unavailable to threads. For example they have a kernel-supported scheduling class, virtual time timers, an alternate signal stack and a profiling buffer.

The system interface to LWPs[4] is shown in Figure 2.

```
int      _lwp_create(context, flag, lwpidp)
void     _lwp_makecontext(ucp, func, arg, private,
                          stack, size);
void    *_lwp_getprivate();
void     _lwp_setprivate(ptr);
void     _lwp_exit();
int      _lwp_wait(waitfor, departedp);
lwpid_t  _lwp_self();
int      _lwp_suspend(lwpid);
int      _lwp_continue(lwpid);
void     _lwp_mutex_lock(lwp_mutexp);
void     _lwp_mutex_unlock(lwp_mutex);
int      _lwp_cond_wait(lwp_condp, lwp_mutexp);
int      _lwp_cond_timedwait(lwp_condp, lwp_mutexp,
                          timeout);
int      _lwp_cond_signal(lwp_condp);
int      _lwp_cond_broadcast(lwp_condp);
int      _lwp_sema_wait(lwp_semap);
int      _lwp_sema_post(lwp_semap);
int      _lwp_kill(lwpid, sig);
```

**Figure 2:** LWP Interfaces

The `_lwp_create()` interface creates another LWP within the process. The `_lwp_makecontext()` interface creates a machine context that emulates the standard calling sequence to a function. The machine context can then be passed to `_lwp_create()`. This allows some degree of machine independence when using the LWP interfaces.

The LWP synchronization interfaces implement mutual exclusion locks, condition variables and counting semaphores. Variants are allowed that can support priority inversion prevention protocols such as priority inheritance. Counting semaphores are also provided to ensure that a synchronization mechanism that is safe with respect to asynchronous signals is available. In general, these routines only actually enter the kernel when they have to. For example, in some cases acquiring an LWP mutex does not require kernel entry if there is no contention. The kernel has no knowledge of LWP synchronization variables except during actual use.

The LWP synchronization variables are placed in memory by the application. If they are placed in shared memory or in mapped files that are accessible to other processes, they will synchronize LWPs between all the mapping processes even though the LWPs in different processes are generally invisible to each other. An advantage of using memory mapped files is that the synchronization variables along with other shared data can be preserved in a file. The application can be restarted and continue using its shared synchronization variables without any initialization. When a LWP synchronization primitive causes the calling LWP to block, the LWP is then suspended on a kernel-supported sleep queue associated with the offset in the mapped file or the shared memory segment[5] [6]. This allows LWPs in different processes to synchronize even though they have the synchronization variable mapped at different virtual addresses.

Both the `_lwp_getprivate()` and `_lwp_setprivate()` interfaces provide one pointer's worth of storage that is private to the LWP. Typically, a threads package can use this to point to its thread data structure. On SPARC this interface sets and gets the contents of register %g7[7].

An alternative to this approach is to have a private memory page per LWP. This requires more kernel effort and perhaps more memory requirements for each LWP. However, it is probably the most reasonable choice on register constrained machines, or on machines where user registers have not been reserved to the system.

The kernel also provides two additional signals. The first, SIGLWP is simply a new signal reserved for threads packages. It is used as an inter-LWP signal mechanism when directed to particular LWPs within the process via the `_lwp_kill()` interface. The second signal is SIGWAITING. This is currently generated by the kernel when it detects that all the LWPs in the process have blocked in indefinite waits. It is used by threads packages to ensure that processes don't deadlock indefinitely due to lack of execution resources.

## Threads Library Implementation

Each thread is represented by a thread structure that contains the thread ID, an area to save the thread execution context, the thread signal mask, the thread priority, and a pointer to the thread stack. The storage for the stack is either automatically allocated by the library or it is passed in by the application on thread creation. Library allocated stacks are obtained by mapping in pages of anonymous memory. The library ensures that the page following the stack is invalid. This represents a ''red zone'' so that the process will be signalled if a thread should run off the stack. If the application passed in its own stack storage, it can provide a red zone or pack the stacks in its own way.

[4] The LWP interfaces described here are compatible with the current UNIX International LWP interfaces.

[5] Multics parlance used to call this the virtual address and what most people today call the virtual address (the effective addresses computed by the instructions) was called the logical address,

[6] Internally, the kernel uses a hash queue to synchronize LWPs. The hash lookup is based on a two word value that represents the internal file handle (vnode) and offset.

[7] On SPARC, register %g7, is reserved to the system in the Application Binary Interface [USO 1990]. ABI compliant applications cannot use it.

When a thread is created, a thread ID is assigned. In an earlier threads library implementation, the thread ID was a pointer to the thread structure. This implementation was discarded in favor of one where the thread ID was used as an index in a table of pointers to thread structures. This allows the library to give meaningful errors when a thread ID of an exited and deallocated thread is passed to the library.

### Thread-local Storage

Threads have some private storage (in addition to the stack) called thread-local storage (TLS). 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. It is used for thread-private data that must be accessed quickly. The thread structure itself and a version of `errno` that is private to the thread is allocated in TLS.

Thread-local storage is obtained via a new `#pragma, unshared`, supported by the compiler and linker. The contents of TLS are zeroed, initially; static initialization is not allowed. The size of TLS is computed by the run-time linker at program start time by summing the sizes of the thread-local storage segments of the linked libraries and computing offsets of the items in TLS[8]. The linker defines a symbol called `_etls` that represent the size of TLS. The threads library uses this at thread creation time to allocate space for TLS from the base of the new thread's stack. After program startup, the size of TLS is fixed and can no longer grow. This restricts programmatic dynamic linking (i.e., `dlopen()`) to libraries that do not contain TLS. Because of these restrictions, thread-local storage is not an exported interface. Instead a programmatic interface called thread-specific data [POSIX 1992] is available.

On SPARC, global register %g7 is assumed by the compiler to point to the base address of TLS. This is the same register used by the LWP private storage interfaces. The compiler generates code for TLS references relative to %g7. The threads library ensures that %g7 is set to the base address of TLS for the currently executing thread. The impact of this on thread scheduling is minimal.

## Thread Scheduling

The threads library implements a thread scheduler that multiplexes thread execution across a pool of LWPs. The LWPs in the pool are set up to be nearly identical. This allows any thread to execute on any of the LWPs in this pool. When a thread executes, it is attached to an LWP and has all the attributes of being a kernel-supported thread.

All runnable, unbound, threads are on a user level, prioritized dispatch queue. Thread priorities range from 0 to ''infinity''[9] A thread's priority is fixed in the sense that the threads library does not change it dynamically as in a time shared scheduling discipline. It can be changed only by the thread itself or by another thread in the same process. The unbound thread's priority is used only by the user level thread scheduler and is not known to the kernel.

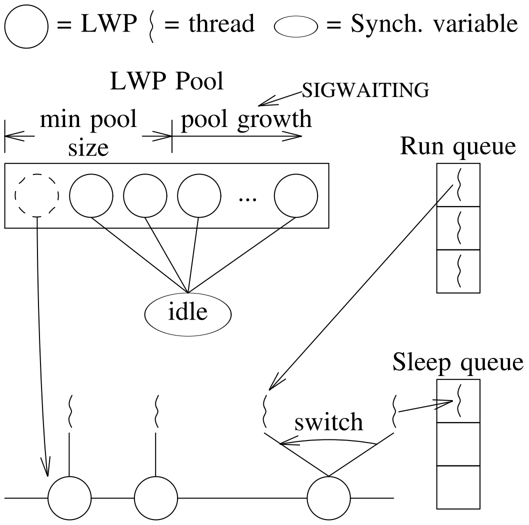


**Figure 3**: Thread Scheduling

Figure 3 gives an overview of the multiplexed scheduling mechanism. An LWP in the pool is either idling or running a thread. When an LWP is idle it waits on a synchronization variable (actually it can use one of two, see below) for work to do. When a thread, T1 is made runnable, it is added to the dispatch queue, and an idle LWP L1 (if one exists) in the pool is awakened by signalling the idle synchronization variable. LWP L1 wakes up and switches to the highest priority thread on the dispatch queue. If T1 blocks on a local synchronization object (i.e., one that is not shared between processes), L1 puts T1 on a sleep queue and then switches to the highest priority thread on the dispatch queue. If the dispatch queue is empty, the LWP goes back to idling. If all LWPs in the pool are busy when T1 becomes runnable, T1 simply stays on the dispatch queue, waiting for an LWP to become available. An LWP becomes available either when a new one is added to the pool or when one of the running threads blocks on a process-local synchronization variable, exits or stops, freeing its LWP.

[8] The size of TLS and offsets in TLS are computed at startup time rather than link time so that the amount of thread-local storage required by any particular library is not compiled into the application binary and the library may change it without breaking the binary interface.

[9] Currently, "infinity" is the maximum number representable by 32 bits.

*Thread States and the Two Level Model*

An unbound thread can be in one of five different states: RUNNABLE, ACTIVE, SLEEPING, STOPPED, and ZOMBIE. The transitions between these states and the events that cause these transitions are shown in Figure 4.

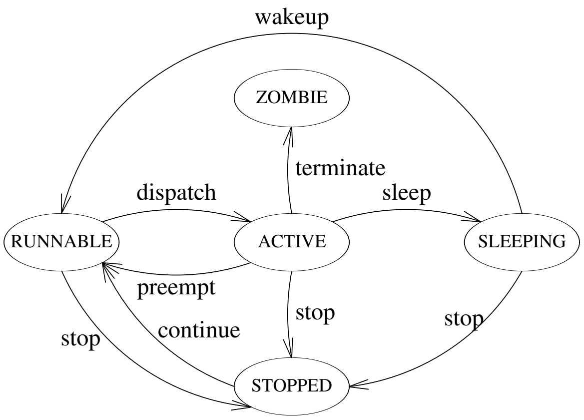


**Figure 4:** Thread states

While a thread is in the ACTIVE state, its underlying LWP can be running on a processor, sleeping in the kernel, stopped, or waiting for a processor on the kernel's dispatch queue. The four LWP states and the transitions between them can be looked upon as a detailed description of the thread ACTIVE state. This relationship between LWP states and thread states is shown in Figure 5.

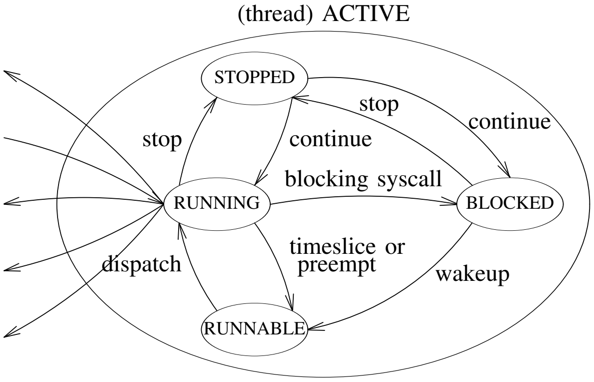


**Figure 5**: LWP states

*Idling and parking*

When an unbound thread exits and there are no more RUNNABLE threads, the LWP that was running the thread switches to a small idle stack associated with each LWP and *idles* by waiting on a global LWP condition variable. When another thread becomes RUNNABLE the global condition variable is signaled, and an idling LWP wakes up and attempts to run any RUNNABLE threads.

When a bound thread blocks on a process-local synchronization variable, its LWP must also stop running. It does so by waiting on a LWP semaphore associated with the thread. The LWP is now *parked*. When the bound thread unblocks, the parking semaphore is signalled so that the LWP can continue executing the thread.

When an unbound thread becomes blocked and there are no more RUNNABLE threads, the LWP that was running the thread also *parks* itself on the thread's semaphore, rather than *idling* on the idle stack and global condition variable. This optimizes the case where the blocked thread becomes runnable quickly, since it avoids the context switch to the idle stack and back to the thread.

*Preemption*

Threads compete for LWPs based on their priorities just as kernel threads compete for CPUs. A queue of ACTIVE threads is maintained. If there is a possibility that a RUNNABLE thread has a higher priority than that of some ACTIVE thread, the ACTIVE queue is searched to find such a thread. If such a thread is found, then this thread is removed from the queue and preempted from its LWP. This LWP then schedules another thread which is typically the higher priority RUNNABLE thread which caused the preemption.

There are basically two cases when the need to preempt arises. One is when a newly RUNNABLE thread has a higher priority than that of the lowest priority ACTIVE thread and the other is when the priority of an ACTIVE thread is lowered below that of the highest priority RUNNABLE thread.

ACTIVE threads are preempted by setting a flag in the thread's thread structure and then sending its LWP a SIGLWP. The threads library always installs its own handler for SIGLWP. When an LWP receives the signal, the handler checks if the current thread has a preemption posted. If it is, it clears the flag and then switches to another thread.

One side-effect of preemption is that if the target thread is blocked in the kernel executing a system call, it will be interrupted by SIGLWP, and the system call will be re-started when the thread resumes execution after the preemption. This is only a problem if the system call should not be re-started.

*The Size of the LWP Pool*

By default, the threads library automatically adjusts the number of LWPs in the pool of LWPs that are used to run unbound threads. There are two main requirements in setting the size of this pool: First, it must not allow the program to deadlock due to lack of LWPs. For example if there are more runnable unbound threads than LWPs and all the active threads block in the kernel in indefinite waits (e.g., read a tty), then the process can make no further progress until one of the waiting threads returns. The second requirement is that the library should make efficient use of LWPs. If the library were to create one LWP for every thread, in many cases these LWPs would simply be idle and the operating system would be overloaded by the resource requirements of many more LWPs than are actually being used. The advantages of a two-level model would be lost.

Let *C* be the total number of ACTIVE and RUNNABLE threads in an application, at any instant. In many cases, the value of *C* will fluctuate, sometimes quite dramatically as events come in and are processed, or threads synchronize. Ideally, if the time for creating or destroying an LWP were zero, the optimal design would be for the threads library to keep the size of the LWP pool equal to *C* at all times. This design would meet both requirements for the pool perfectly. In reality, since adjusting the size of the LWP pool has a cost, the threads library does not attempt to match it perfectly with *C* since this would be inefficient.

The current threads library implementation starts by guaranteeing that the application does not deadlock. It does so by using the new SIGWAITING signal that is sent by the kernel when all the LWPs in the process block in indefinite waits. The threads library ignores SIGWAITING by default. When the number of threads exceeds the number of LWPs in the pool, the threads library installs a handler for SIGWAITING. If the threads library receives a SIGWAITING and there are runnable threads, it creates a new LWP and adds it to the pool. If there are no runnable threads at the time the signal is received and the number of threads is less than the number of LWPs in the pool, it disables SIGWAITING.

A more aggressive implementation might attempt to compute a weighted time average of the application's concurrency requirements, and adjust the pool of LWPs more aggressively. Currently, it is not known whether the advantages of doing this would outweigh the extra overhead of creating more LWPs.

The application sometimes knows its concurrency requirements better than the threads library. The threads library provides an interface, `thr_setconcurrency()` that gives the library a hint as to what the expected concurrency level is. For example, a multi-threaded file copy program, with one input and one output thread might set its concurrency level to two; a multi-threaded window system server might set its concurrency level to the number of expected simultaneously active clients times the number of threads per client. If *n* is the level of concurrency given in `thr_setconcurrency()`, the threads library will ensure there are at least *n* LWPs in the pool when the number of threads is greater than or equal to *n*. If the number of threads is less than *n*, the library ensures that the number of LWPs is at least equal to the number of threads. Any growth in the number of LWPs past *n* is due to receiving a SIGWAITING.

The number of LWPs in the pool can grow to be greater than the number of threads currently in the process due to previous receptions of SIGWAITING or uses of `thr_setconcurrency()`. This can result in an excessive number of LWPs in the pool. The library therefore ''ages'' LWPs; they are terminated if they are unsued for a ''long'' time, currently 5 minutes. This is implemented by setting a per-LWP timer whenever the LWP starts idling. If the timer goes off, the LWP is terminated.

### Mixed Scope Scheduling

A mixture of bound and unbound threads can coexist in the same process. Applications such as window systems can benefit from such mixed scope scheduling. A bound thread in the real-time scheduling class can be devoted to fielding mouse events which then take precedence over all other unbound threads which are multiplexing over time-sharing LWPs. When a bound thread calls the system's priority control interface, `priocntl()`, it affects the LWP it is bound to, and thus the thread itself. Thus, bound, real-time threads can coexist with unbound threads multiplexing across time-shared LWPs. Unbound threads continue to be scheduled in a multiplexed fashion in the presence of bound threads.

*Reaping bound/unbound threads*

When a detached thread (bound or unbound) exits, it is put on a single queue, called `deathrow` and their state is set to ZOMBIE. The action of freeing a thread's stack is not done at thread exit time to minimize the cost of thread exit by deferring unnecessary and expensive work. The threads library has a special thread called the reaper whose job is to do this work periodically. The reaper runs when there are idle LWPs, or when a reap limit is reached (the `deathrow` gets full). The reaper traverses the `deathrow` list, freeing the stack for each thread that had been using a library allocated stack.

When an undetached thread (bound or unbound) exits, it is added to the zombie list. Threads on the zombie list are reaped by the thread that executes `thr_join()` on them.

Since the reaper runs at high priority, it should not be run too frequently. Yet, it should not be run too rarely, since the rate at which threads are reaped has an impact on the speed of thread creation. This is because the reaper puts the freed stacks on a cache of available thread stacks, which speeds up stack allocation for new threads, an otherwise expensive operation.

## Thread Synchronization

The threads library implements two basic types of synchronization variables, process-local (the default) or process-shared. In both cases the library ensures that the synchronization primitives themselves are safe with respect to asynchronous signals and therefore they can be called from signal handlers.

**Process-local Synchronization Variables**

The default blocking behavior is to put the thread to sleep. Each synchronization variable has a sleep queue associated with it. The synchronization primitives put the blocking threads on the synchronization variable's sleep queue and surrenders the executing thread to the scheduler. If the thread is unbound, the scheduler dispatches another thread to the thread's underlying LWP. A Bound thread, however, must stay permanently bound to its LWP so the LWP is *parked* on its thread.

Blocked threads are awakened when the synchronization variables become available. The synchronization primitives that release blocked threads first check if threads are waiting for the synchronization variables. A blocked thread is removed from the synchronization variable's sleep queue and is dispatched by the scheduler. If the thread is bound, the scheduler unparks the bound thread so that its LWP is dispatched by the kernel. For unbound threads, the scheduler simply places the thread on a run queue corresponding to its priority. The thread is then dispatched to an LWP in priority order.

**Process-shared Synchronization variables**

Process-shared synchronization objects can also be placed in memory that is accessible to more than one process and can be used to synchronize threads in different processes. Process-shared synchronization variables must be initialized when they are created because their blocking behavior is different from the default. Each synchronization primitive supports an initialization function that must be called to mark the process-shared synchronization variable as process-shared. The primitives can then recognize the synchronization variables that are shared and provide the correct blocking behavior. The primitives rely on LWP synchronization primitives to put the blocking threads to sleep in the kernel still attached to their LWPs, and to correctly synchronize between processes.

## Signals

The challenge in providing the SunOS MT signal semantics for user threads in a two-level model of multi-threading is that signals are sent by the kernel but user level threads and their masks are invisible to the kernel. In particular, since signal delivery to a thread is dependent on the thread signal mask, the challenge is to elicit the correct program behavior even though the kernel cannot make the correct signalling decisions because it cannot see all the masks.

The implementation has the additional goal of providing cheap async safe synchronization primitives. A function is said to be async safe if it is reentrant with respect to signals, i.e., it is callable from a signal handler invoked asynchronously. Low overhead async safe synchronization primitives are crucial for multi-threaded libraries containing internal critical sections, such as the threads library and its clients, such as libc, etc. For example, consider a call to the threads library, say `mutex_lock()`, which is interrupted asynchronously while holding an internal threads library lock, *L*. The asynchronous handler could potentially call into `mutex_lock()` also and try to acquire *L* again, resulting in a deadlock. One way of making `mutex_lock()` async safe is to mask signals while in *L*'s critical section. Thus, efficient signal masking was an important goal since it could provide efficient async safe critical sections.

**Signal Model Implementation**

One implementation strategy would be for each LWP that is running a thread to reflect the thread's signal mask. This allows the kernel to directly choose a thread to signal from among the ACTIVE threads within a process. The signals that a process can receive changes as the threads in the application cycle through the ACTIVE state.

This strategy has a problem with threads that are rarely ACTIVE and are the only threads in the application that have certain signals enabled. These threads are essentially asleep waiting to be interrupted by a signal which they will never receive. In addition, a system call must be done whenever an LWP switches between threads having different masks or when an active thread adjusts its mask.

This problem can be solved if the LWP signal masks and the ACTIVE thread signal masks are treated more independently. The set of signals that a process can receive is equal to the intersection of all the thread signal masks. The library ensures that the LWP signal mask is either equal to the thread mask or it is less restrictive. This means occasionally signals are sent by the kernel to ACTIVE threads that have the signal disabled.

When this occurs the threads library prevents the signal from actually reaching the interrupted thread by interposing its own signal handlers below the application signal handlers. When a signal is delivered, the global handler checks the current thread's signal mask to determine if the thread can receive this signal. If the signal is masked, the global handler sets the current LWP's signal mask to the current thread's signal mask. Then the signal is resent to the process if it is an undirected signal or to its LWP if it is a directed signal. The kernel signal delivery mechanism provides information that allows the signal handler to distinguish between directed and undirected signals. If the signal is not masked, the global handler calls the signal's application handler. If the signal is not appropriate for any of the currently `ACTIVE` threads, the global handler can cause one of the inactive threads to run if it has the signal unmasked.

Synchronously generated signals are simply delivered by the kernel to the ACTIVE thread that caused them.

*Sending a directed signal*

A thread can send a signal to another thread in the same process using `thr_kill()`. The basic means of sending a signal to a thread is to send it to the LWP it runs on. If the target thread is not ACTIVE, `thr_kill()` just posts the signal on the thread in a pending signals mask. When the target thread resumes execution, it receives the pending signals[10]. If the target thread is ACTIVE, `thr_kill()` sends the signal to the target thread's LWP via `lwp_kill()`. If the target LWP is blocking the signal (which implies that the thread is blocking it too), the signal stays pending on the LWP, until the thread unblocks it. While the signal is pending on the LWP, the thread is temporarily bound to the LWP until the signals are delivered to the thread.

**Signal Safe Critical Sections**

As stated previously, to prevent deadlock in the presence of signals, critical sections that are reentered in a signal handler in both multi-threaded applications and the threads library should be safe with respect to signals. All asynchronous signals should be masked during such critical sections.

The threads library signals implementation allows multi-threaded applications to make critical sections safe as efficiently as possible via a low overhead implementation of the `thr_sigsetmask()` interface. If signals do not occur, `thr_sigsetmask()` does not result in a system call. In this case, it is as fast as just modifying the user-level thread signal mask.

The threads library has an even faster means of achieving signal safety for its internal critical sections. The threads library sets/clears a special flag in the threads structure whenever it enter/exits an internal critical section. Effectively, this flag serves as a signal mask to mask out all signals. If the flag is set when a signal is delivered, the threads library global signal handler will defer the signal as described in the above section.

## Debugging Threads

A debugger that can control library supported threads requires access to information about the threads inside the debugged process. The normal kernel-supported debugging interfaces (/proc) are insufficient. One could build complete knowledge of the threads implementation into the debugger, but that would force a re-release of the debugger whenever some internal threads library data structure changes. Instead the threads library provides a separate dynamically linked thread debugging library that the debugger links with via `dlopen()`. The thread debugging library contains interfaces that allow the debugger access to general thread information, without the debugger itself containing knowledge of the threads library implementation. All the threads library specific knowledge is contained in the thread debugging library. The interfaces in the thread debugging library allow the debugger to for example, enumerate the list of threads in the debugged process, or get/set the state and/or registers of each thread (ACTIVE or not).

We have a version of DBX with thread extensions that dynamically links with this library. The new features allow the programmer to debug a multi-thread process. It will list all threads in the process and their states, whether they are sleeping, running, active, stopped or zombied. DBX can change its focus to any thread so that its internal state can be analyzed with the standard DBX commands.

[10]The threads library context switch code ensures this by sending the pending signals to the LWP that resumes the thread.

## References

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

[Eykholt 1992] 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 System V Release 4 Kernel'', Proc. 1992 USENIX Summer Conference.

[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.

[Khanna 1992] Sandeep Khanna, Michael Sebrée, John Zolnowsky, ''Realtime Scheduling in SunOS 5.0'', Proc. 1992 USENIX Winter Conference.

[POSIX 1992] POSIX P1003.4a, ''Threads Extension for Portable Operating Systems'', IEEE.

[Powell 1991] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, ''SunOS Multi-thread Architecture'', Proc. 1991 USENIX Winter Conference.

[Sha 1990] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, ''Priority Inheritance Protocols: An Approach to Realtime Synchronization'', Vol 39, No 9, September 1990, IEEE Transactions on Computers.

[USO 1990] UNIX Software Operation, ''System V Application Binary Interface, SPARC Processor Supplement'', UNIX Press.

### Author Information

Dan Stein is currently a member of technical staff at SunSoft, Inc. where he is one of the developers of the SunOS mult-thread Architecture. He graduated from the University of Wisconsin in 1981 with a B.S. in Computer Science.

Devang Shah is currently a Member of Technical Staff at SunSoft, Inc. He received an M.A. in Computer Science from the University of Texas at Austin in 1989 and a B. Tech. in Electronics Engg. from the Institute of Technology, B.H.U., India in 1985. At UT-Austin he extended SunOS 3.2 to provide lightweight processes. Prior to UT-Austin, he worked at Tata Consultancy Services, Bombay.