Kernel-Scheduled Entities for FreeBSD

Jason Evans
jasone@freebsd.org

November 7, 2000

Abstract

FreeBSD has historically had less than ideal support for multi-threaded application programming. At present, there are two threading libraries available. libc_r is entirely invisible to the kernel, and multiplexes threads within a single process. The linuxthreads port, which creates a separate process for each thread via *rfork*(), plus one thread to handle thread synchronization, relies on the kernel scheduler to multiplex "threads" onto the available processors.

Both approaches have scaling problems. libc_r does not take advantage of multiple processors and cannot avoid blocking when doing I/O on "fast" devices. The linuxthreads port requires one process per thread, and thus puts strain on the kernel scheduler, as well as requiring significant kernel resources. In addition, thread switching can only be as fast as the kernel's process switching.

This paper summarizes various methods of implementing threads for application programming, then goes on to describe a new threading architecture for FreeBSD, based on what are called kernel-scheduled entities, or scheduler activations.

1 Background

FreeBSD has been slow to implement application threading facilities, perhaps because threading is difficult to implement correctly, and the FreeBSD's general reluctance to build substandard solutions to problems. Instead, two technologies originally developed by others have been integrated.

libc_r is based on Chris Provenzano's userland pthreads implementation, and was significantly reworked by John Birrell and integrated into the FreeBSD source tree. Since then, numerous other people have improved and expanded libc_r's capabilities. At this time, libc_r is a high-quality userland implementation of POSIX threads.

The linuxthreads port is based on Xavier Leroy's LinuxThreads, which is now an integral part of GNU libc. Most of the original porting work was done by Richard Seaman. The linuxthreads port is completely compatible with LinuxThreads as it runs on Linux.

Both of these libraries have scalability and performance problems for certain types of threaded programs.

2 Threading Architectures

Following is a short description of several threading architectures, along with a representative implementation for each of the more common ones. In all cases, threading is preemptive.

2.1 Userland (ala FreeBSD's libc_r)

Userland threads are implemented entirely in an application program, with no explicit support from the kernel. In most cases, a threading library is linked in to the application, though it is also possible to hand code user threading within an application.

In order for userland threading to work, two main issues have to be resolved:

Preemptive scheduling: Threads must be periodically preempted so that all runnable threads of sufficient priority get to run. This is done by a combination of a timer signal (SIGALARM for libc_r), which allows the userland threads scheduler (UTS) to run, and *setjmp*()/*longjmp*() calls to switch between threads.

Process blocking: Normally, when a process makes a system call that cannot be completed immediately, the process blocks and another process is scheduled in order to make full use of the processor. However, a threaded program may have multiple runnable threads, so blocking in a system call should be avoided. This is accomplished by converting potentially blocking system calls to non-blocking. This works well in all cases except for operations on so-called fast devices such as local filesystems, where it is not possible to convert to a non-blocking system call. libc_r handles non-blocking and incomplete system calls by converting file descriptors to non-blocking, issuing I/O requests, then adding file descriptors to a central *poll*()-based event loop.

Userland threads have the advantage of being very fast in the simple case, though the complexities of call conversion eats into this performance advantage for applications that make many system calls.

As libc_r currently exists, there are several problems:

1. The central *poll*() loop does not scale well to large numbers of threads. This could probably be solved by converting to the new *kqueue*() interface.
2. The entire process blocks on I/O to fast devices. This problem could probably be solved by integrating asynchronous I/O into libc_r, which would first require stabilizing the AIO implementation, then require switching from a *poll*()-based event loop to a *kqueue*()-based event loop in order to be able to integrate I/O completion notifications into the same loop.
3. Since all threads are multiplexed onto a single process, libc_r cannot take advantage of multiple CPUs in an SMP system. There is no reasonable retrofit to libc_r which solves this scalability problem.

2.2 Process-based (ala Linux's LinuxThreads)

Process-based threading is confusingly referred to as kernel threading much of the time. True kernel threads are threads that run within the kernel in order to run various portions of the kernel in parallel. Process-based threads are threads that are based on some number of processes that share their address space, and are scheduled as normal processes by the kernel. LinuxThreads implements process-based threading.

New processes are created such that they share the address space of the existing process(es), as shown in Figure 2. For Linux, this is done via the *clone*() system call, whereas for FreeBSD it is done via a special form of the *rfork*() system call. See

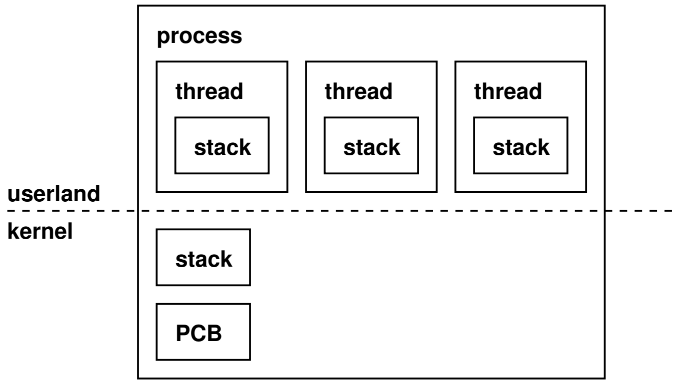

Figure 1: Userland threading model

As soon as a second thread is created via *pthread_create*(), LinuxThreads creates an additional process that is used exclusively for synchronization among processes. This means that for a program with n threads, there are $n + 1$ processes, where at some point in time there were multiple threads.

LinuxThreads is elegant in that it is simple. However, it has at least the following POSIX compliance issues that cannot be easily addressed:

- Each thread runs in a separate process. Each process has a unique process ID (pid), but POSIX requires all threads to appear to have the same pid. Fixing this would require significant modifications to the kernel's data structures.
- The thread priority semantics specified by POSIX cannot be implemented, because each thread is actually a process, which makes thread contention at the application level impossible. All thread contention is by definition at the system level. This has the additional disadvantage that multi-threaded applications compete unfairly with single-threaded applications, which makes running a mixture of applications difficult on a single machine.

Process-based threading also has some inherent performance and scalability issues that cannot be overcome:

- Switching between threads is a very expensive operation. It requires switching to kernel mode, switching the old thread (process) out, and then running the new thread (process). There is no solution to this problem except to optimize process switching, which defies optimization beyond a certain point. This problem is aggravated by cache locality considerations.
- Each thread (process) requires all the kernel resources typically associated with a process. This includes a kernel stack, which means that applications with many threads require large amounts of kernel resources.

2.3 Multi-level (ala Solaris's LWPs)

Multi-level threading is a hybrid of user-level and process-based threading. Threads are multiplexed onto a pool of processes. The size of the process pool is normally determined automatically by heuris-

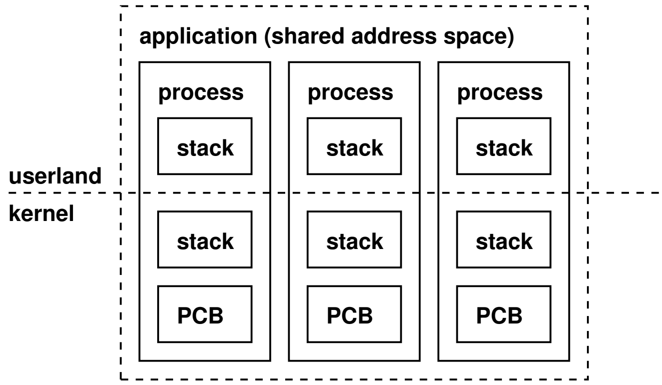

Figure 2: Process-based threading model

tics internal to the threading library that take into account issues such as the number of processors, number of threads, and processor binding of threads.

The idea of multi-level threading is to achieve the performance of userland threading and the SMP scalability of process-based threading. Ideally, most thread scheduling is done by a UTS to avoid the context switch overhead of kernel calls, but multiple threads can run concurrently by running on more than one process at the same time.

In practice, multi-level threading's main shortcoming is its complexity. Ideally, the advantages of userland and process-based threading are combined, without any of the disadvantages, but in practice, some of the disadvantages tend to slip in. The overhead of the multi-level scheduling compounds this.

Multi-level threading does not require light-weight processes (LWPs) in order to work, but Solaris uses LWPs to address the POSIX compliance issues mentioned above for purely process-based threading. Also, in theory, LWPs are light-weight, though Solaris's LWPs no longer generally meet this criterion. That is, by the time Sun got the kinks worked out, LWPs were no longer light-weight.

2.4 Scheduler Activations

This is a very brief overview of scheduler activations (SAs) as presented in [Anderson], and is meant only as a basis for the more complete treatment of kernel-scheduled entities (KSEs) for FreeBSD later in this paper. There are many details left out in this section, which are treated in detail in the original paper.

The SAs approach, like multi-level threading, strives to merge the advantages of userland and process-based threading while avoiding the disadvantages of both approaches. SAs differ from multi-level scheduling in that additional kernel facilities are added in order to provide the UTS with exactly the information and support it needs in order to control scheduling. Simply put, SAs allow the kernel and the UTS to do their jobs without any guess work as to what the other is doing.

A process that takes advantage of SAs has a significantly different flow of control than a normal process, from the perspective of the kernel. A normal process has a number of data structures associated with it in the kernel, including a stack and process control block (PCB). When a process is switched out, its machine state is saved in the PCB. When the process is run again, the machine state is restored from the PCB and the process continues running, whether in kernel or user mode.

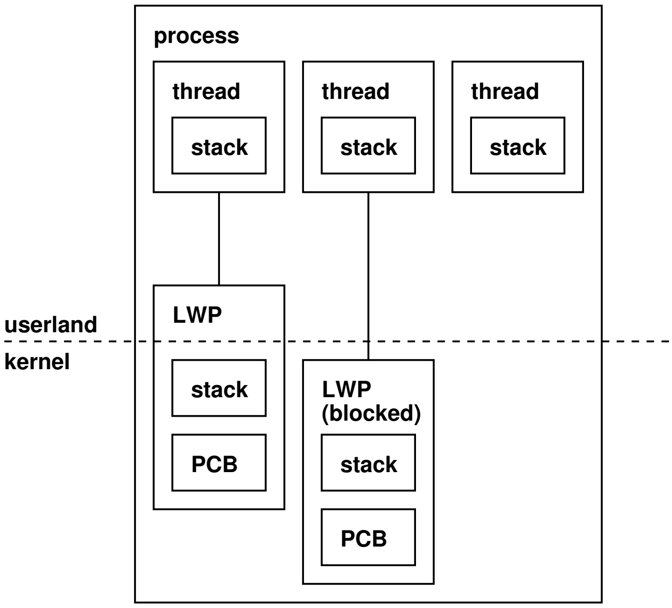

Figure 3: Light-weight process-based threading model

A process that is using SAs does not have a kernel stack or PCB. Instead, every time a process is run, a SA is created that contains a kernel stack and thread control block (TCB), and the process runs in the context of the SA. When the SA is preempted or blocked, machine state is stored in the SA's TCB, and the kernel stack is optionally used for completion of a pending system call. See Figure 4.

It is possible to run more than one SA for the same process concurrently on multiple processors. However, the UTS needs to know at all times exactly what processors it is running on, so that it can make informed scheduling decisions. As part of the solution to this problem, every time a SA is started, it initially starts executing in the UTS. Additionally, the kernel makes upcalls to the process to notify it of important events that may affect thread scheduling decisions. The following upcalls are necessary:

***sa_new*(cpu_id):** Execute a thread on the processor with ID *cpu_id*.

***sa_preempt*(sa_id):** A thread was preempted, with SA ID *sa_id*.

***sa_block*(sa_id, pcb):** A thread blocked in the kernel, with SA ID *sa_id* and machine state *pcb*.

***sa_unblock*(sa_id, pcb):** A thread that was blocked in the kernel has completed, with SA ID *sa_id* and machine state *pcb*.

Also, the following system calls are necessary:

***sa_alloc_cpus*(ncpus):** Allocate *ncpus* additional CPUs, if possible.

***sa_dealloc_cpu*():** Remove a CPU (reduce concurrency by one).

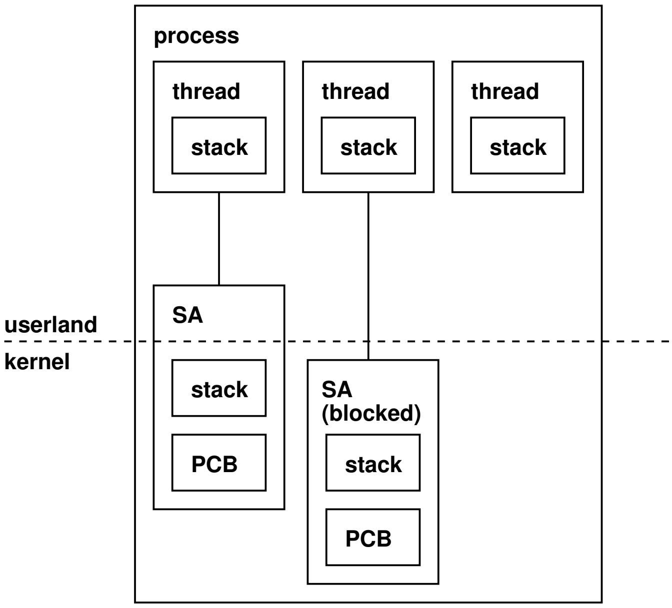

Figure 4: Scheduler activations

3 Kernel-scheduled entities

Kernel-scheduled entities (KSEs) are similar in concept to scheduler activations, but support for threads with system-level scheduling contention is added. Figure 5 shows how KSEs and userland interact. Scheduling contention is controlled via KSE groups (KSEGs). A process has one or more KSEGs associated with it. Each KSEG has a concurrency level associated with it, which controls the maximum number of concurrent KSEs that can be run for that KSEG. Each KSEG is a separate entity from a timesharing perspective.

A process starts out with one KSEG that has a concurrency level of one. As new threads are created, the concurrency level of that KSEG can be adjusted, up to the maximum number of CPUs available for execution of the process. If a thread with system-level scheduling contention is desired, a new KSEG with a concurrency level of one can be created for that thread.

Since each KSEG is a separate entity from a timesharing perspective, allowing a process to create an arbitrary number of KSEGs would give the process an unfair scheduling advantage. Therefore, instead of enforcing process limits, KSEG limits are enforced. In the common case of single-threaded applications, there is a one to one correspondence between the two methods of enforcing resource limits, but in the case of multi-threaded applications, this prevents a single user from being able to gain a larger portion of the CPU time than would be possible with multiple single-threaded applications.

The KSE architecture makes a distinction between a KSE and a KSE context (KSEC). KSEs are used in the kernel in the scheduler queues, and as a general handle, much the same way as a process is used in current BSD kernels. Given a pointer to a KSE, it is possible to access its associated KSEG, proc, and KSEC instances. A KSEC contains the state of a suspended thread of execution. When a running KSE is blocked, the execution state is saved in the KSEC so that when it is possible to continue execution, the KSEC can be re-attached to a KSE and continued. When a KSE is preempted, the execution state

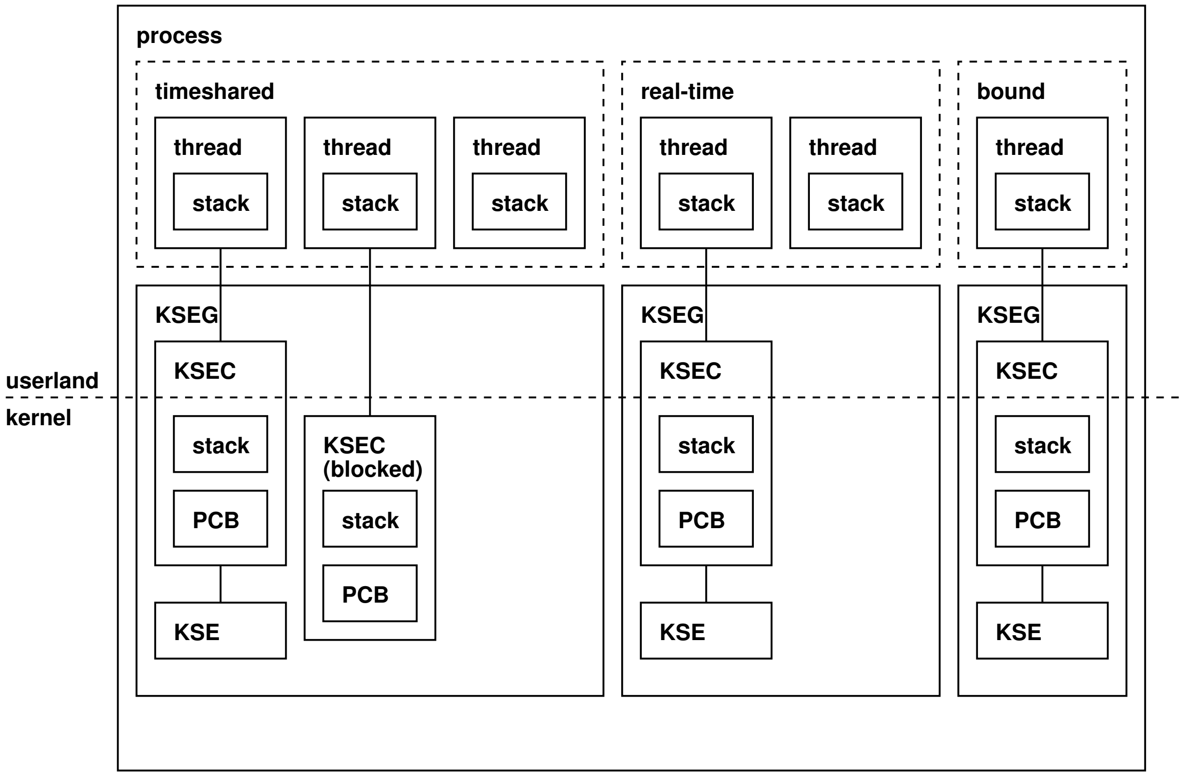

Figure 5: Kernel-scheduled entities

is saved in a KSEC so that an userland state can be handed to the UTS.

KSEs are only evident within the kernel. The interface with userland only deals with processes, KSEGs, and KSECs. KSEs themselves are irrelevant to userland because they serve essentially as an anonymous handle that binds the various kernel structures together.

3.1 Operation without upcalls

Unless upcalls are explicitly activated by a program, execution is almost identical to the traditional BSD process model. The proc structure is still broken into four components, and the KSE is used as the handle to the process most places in the kernel, but otherwise, little is changed. Figure 6 shows the linkage between the four data structures that comprise the single-threaded component. The dashed lines denote linkage that only exists if upcalls are not activated. In other words, the data structure linkage for a single-threaded application with upcalls activated on would not include the dashed lines.

3.2 Operation with upcalls

At the time upcalls are activated via a system call, program flow changes radically. Non-blocking system calls will behave normally, but blocking system calls, preemption, and running will cause the program to receive upcalls from the kernel. Figure 7 shows the data structure linkage for a process

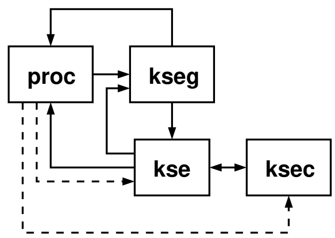

Figure 6: KSE data structure linkage for a single-threaded process

running on a 4 CPU machine that has real-time, system scope, and timeshared threads. The KSEs that have an associated KSEC are currently running.

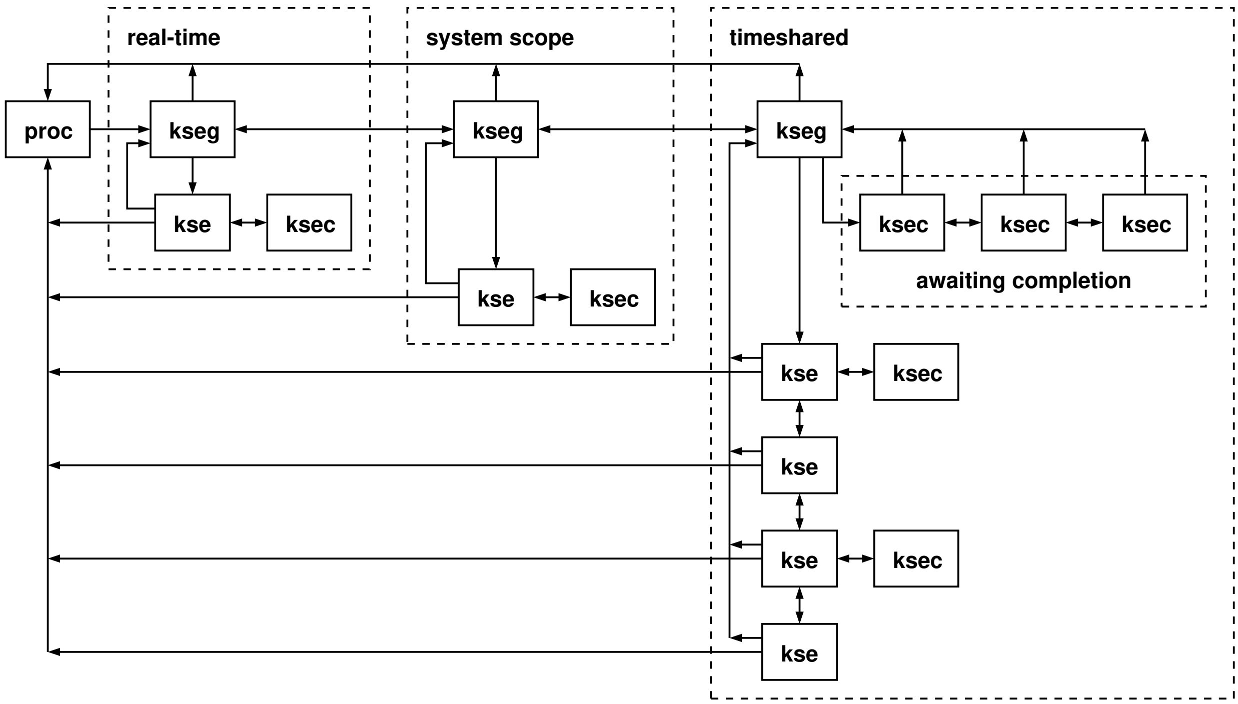

Figure 7: KSE data structure linkage for a multi-threaded process on a 4 CPU machine

3.3 APIs

KSEs require the ability to make upcalls to userland. This is a very different flow of control than normal process execution uses. Normal processes don't know when they are preempted or resumed; execution appears seamless. With KSEs, the process is notified of every preemption and resumption, which requires upcalls.

The following upcalls are necessary:

***ksec_new*(ksec_id, cpu_id, kseg_id):** Run a thread on this KSEC (whose ID is *ksec_id*), on the CPU with ID *cpu_id*, as part of the KSEG with ID *kseg_id*.

***ksec_preempt*(ksec_id, kse_state):** The KSEC with ID *ksec_id* was preempted, with userland execution state *ksec_state*.

***ksec_block*(ksec_id):** The KSEC with ID *ksec_id* has blocked in the kernel.

ksec_unblock(ksec_id, kse_state): The KSEC with ID *ksec_id* has completed in the kernel, with with userland execution state *ksec_state*.

The following system calls are necessary:

void *kse_init*(struct kseu *context): Start using KSEs. *context* contains the necessary data for the kernel to make upcalls. This function appears to return every time an upcall is made. Initially, there is only one KSEG (ID 0), which has a concurrency level of 1.

int *kseg_create*(void): Create a KSEG and return its KSEG ID (unique within this process), or -1 if there is an error (resource limit exceeded).

int *kseg_concurrency*(int kseg_id, int adjust): Adjust the concurrency of the KSEG with ID *kseg_id*. Decrementing the concurrency to 0 destroys the KSEG, as soon as there are no more active KSECs in the KSEG. If *adjust* is 0, the KSEG is not modified, but the concurrency is still returned. This system call returns the KSEG's instantaneous concurrency level after adjusting it.

int *kseg_bind*(int kseg_id, int cpu_id): Bind the KSEG with ID *kseg_id* to the CPU with ID *cpu_id*. This system call returns the CPU ID that the KSEG is bound to, or -1 if there is an error (invalid CPU ID, or the KSEG's concurrency is greater than 1).

The semantics of the *kseg_concurrency*() system call differ from those of the *sa_alloc_cpus*() and *sa_dealloc_cpus*() system calls as presented in [Anderson]. *kseg_concurrency*() does not immediately relinquish the processor and discard the KSEC, whereas *sa_dealloc_cpus*() immediately discards the SA with which the system call is made. In order to get the same effect, a *kseg_concurrency*(kseg_id, -1) call must be followed by a *sched_yield*() call.

There is no *kseg_unbind*() system call, mainly because the common usage case (libpthread) doesn't include binding, then unbinding a KSEG. If the need arises to unbind a bound KSEG, unbinding can be simulated by destroying the KSEG and creating a new, unbound KSEG. Figure 7 shows the basic linkage between processes (proc), KSEGs (kseg), KSEs (kse), and KSECs (ksec). The diagram corresponds to a four processor machine. Two of the processors are running KSEs on behalf of bound KSEGs, and the other two processors are running KSEs on behalf of a softly-affined KSEG.

3.4 Kernel Scheduler

A number of changes to the kernel scheduler are mandatory in order to support KSEs. Chief among these changes are:

- KSEs are are placed in the scheduler queues, rather than processes. This enables concurrent execution of KSEs that are associated with the same process.

- Timesharing calculations are done with KSEGs, rather than processes. This allows multiple KSEGs in one process to compete for CPU time at a system-wide level.
- The state for blocked (incomplete) system calls is stored in KSECs. This means that this queue consists of KSECs rather than KSEs. In other words, the scheduler deals with KSEs in some places, and KSECs in other places.

Additionally, soft processor affinity for KSEs is important to performance. KSEs are not generally bound to CPUs, so KSEs that belong to the same KSEG can potentially compete with each other for the same processor; soft processor affinity tends to reduce such competition, in addition to well-known benefits of processor affinity.

3.5 Userland threads scheduler

The KSE-based UTS is actually simpler than is possible for a userland-only threads implementation, mainly because there is no need to perform call conversion. The following is a simplified representation of the core UTS logic.

1. Find the highest priority thread that is mapped to the kseg that this KSEC is part of. Optionally heuristically try to improve cache locality by running a thread that may still be partially warm in the processor cache.
2. Set a timer that will indicate the end of the scheduling quantum.
3. Run the thread.

Of course, there are a number of ways to enter this code, such as getting a new KSEC, running a thread to the end of a quantum, or rescheduling due to another thread blocking. However, the fact remains that the UTS logic is quite simple.

3.5.1 Temporary priority inversion

The UTS always has the information it needs to make fully informed scheduling decisions. However, in the case of priority-based thread scheduling, there are some circumstances that can cause temporary scheduling inversions, where a thread may continue to run to the end of its quantum despite there being a higher priority runnable thread. This can happen when:

1. A new thread is created that has a lower priority than its creator, but a higher priority than a thread that is concurrently running on another processor.
2. A KSE (running thread A) is preempted by the kernel, and the upcall notification causes preemption of thread B, which is higher priority than thread A, though thread C is also running on another processor and has a lower priority than both A and B. In this case, A will be scheduled and C will continue to run, even though thread B is higher priority than C. Note that there are much more convoluted examples of this same basic idea.

Such temporary inversions are technically violations of the policy that lower priority threads never run in the presence of higher priority threads, but there are two reasons not to do anything about it:

1. Solutions to this problem require additional system calls in which the UTS explicitly asks the kernel to preempt KSEs. This is expensive.

2. Temporary inversions are logically equivalent to the race condition where the UTS determines that a thread should be preempted in favor of scheduling another thread, while the thread races to complete its quantum. It is not important whether the UTS or the thread wins the race; allowing the lower priority thread to complete its quantum without competition from the UTS simply allows the thread to always win.

3.6 Initialization and upcalls

The code that is necessary to start up KSEs in a userland program looks something like:

```
void
foo(void)
{
        struct kseu       kseu_context;

        kse_init(&kseu_context);
        /* Handle upcall and longjmp() to UTS. */
}
```

Each upcall appears to be a return from *kse_init*(). The stack on which *kse_init*() is called must never be unwound, since the kernel fakes up a context at the point where *kse_init*() is called for each upcall. The argument to *kse_init*() points to a data structure that contains enough information for the kernel to determine where to write events. Due to the asynchronous completion of blocked KSECs, it is possible for a single upcall to report a large number of *ksec_unblock*() events, the number of which is only (potentially) bounded by the resource limit settings for the process. Since *ksec_unblock*() events include the userland KSEC execution state, the amount of space needed to report all events during an upcall can vary wildly; static event buffers are inadequate for such a dynamic problem. Therefore, *kseu_context* contains a pointer to a chain of objects with embedded TCBs, and the kernel stores the TCBs from *ksec_preempt*() and *ksec_unblock*() events in the chain elements. Figure 8 shows the basics of how *struct kseu* is laid out.

3.6.1 Bounds on events per upcall

In order for TCB chaining to work, the UTS must assure that the TCB chain is always long enough to handle the maximum possible number of events. Following is an analysis of bounds that can be placed on the number of events reported by the kernel during an upcall.

***ksec_new*():** Each upcall implies a single *ksec_new*() event, the three arguments of which can be stored in *struct kseu*.

***ksec_preempt*():** There can be no more *ksec_preempt*(e)vents than there are processors, since in the worst case, all processors are running KSEs for this process, which all are preempted, and all the corresponding events are reported in the next upcall.

***ksec_block*():** There can be no more *ksec_block*() events than there are processors, since in the worst case, all processors are running KSEs for this process, which all block in the kernel, and all the

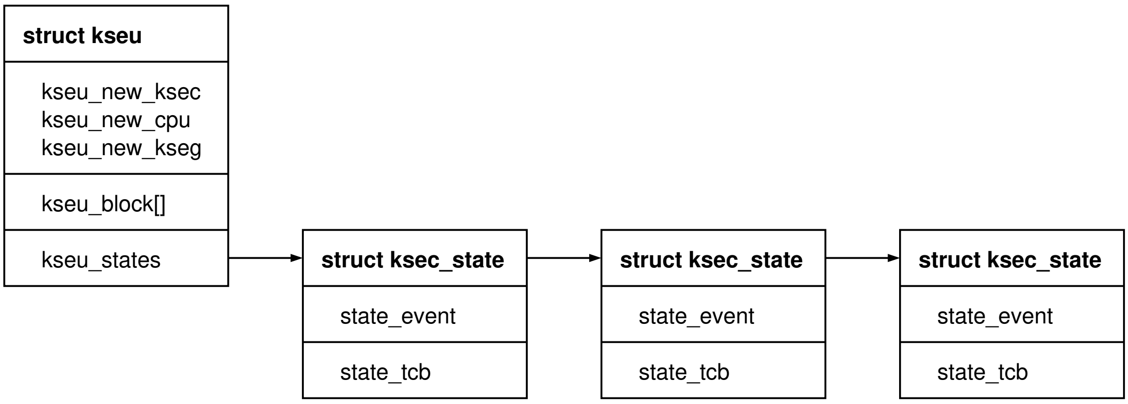

Figure 8: TCB chaining for upcall event storage

corresponding events are reported in the next upcall. The obvious way to report these events is to embed an array of *ksec_id*s in *struct kseu*, and use as many of the elements in the array as necessary. Alternatively, since the UTS has enough information to associate CPU IDs with KSEC IDs, a bitmap could be used, where a 1 bit would indicate that the KSEC that most recently ran on that processor blocked.

***ksec_unblock*():** There can be no more *ksec_unblock*() events than there are userland threads, since each thread can only be blocked once at any given time. The UTS has intimate knowledge of how many threads are running at all times, so it is able to calculate the maximum number of threads that can potentially block (and by extension later unblock) in the kernel at any given time, and it can keep a running total of the number of threads currently blocked. The sum of these values is the maximum number of *ksec_unblock*() events possible in a single upcall.

In practice, the bounds discussed above are rather academic, because it turns out that the UTS needs exactly as many TCBs as there are threads; the UTS can make good internal use of TCBs that the kernel won't possibly need for storing thread state. The only tricky part is avoiding race conditions, such as a TCB not being attached to the chain of free TCBs by the time a KSE blocks in the kernel.

The above bounds ignore one important detail that must be accounted for: the UTS itself may be preempted, or it may block due to one or more page faults. Therefore, there must be enough additional TCBs to allow the UTS to block on all processors.

In total, the number of TCBs needed at any given time is $P \times 2 + T$, where P is the number of processors, and T is the number of threads.

3.6.2 Per-upcall event ordering

The techniques described in the previous section for storing events separate the event types that pass TCBs from those that don't. This means that event ordering is not implicit, and would require extra work to preserve. However, it turns out that in most cases, order doesn't matter, and in the cases that it does matter, the UTS can always determine order.

As an example, suppose *ksec_block*() and *ksec_unblock*() events occur for the same KSEC ID in the same upcall. In one interpretation, where the *ksec_block*() event occurred first, the UTS last knew the corresponding thread to be running, so the two events simply mean that the thread can be continued. In the other interpretation, where the *ksec_unblock*() event occurred first, the KSEC ID has been recycled. However, this isn't possible, since there must have been an intervening *ksec_new*() event, which couldn't have happened until after the KSEC unblocked.

3.7 Signals

Signal delivery in threaded programs is a hairy problem even in the best of circumstances. Userland threads libraries do a lot of work to wrap signal-related library functions in such a way that the threading kernel is able to intercept all signals and deliver them to an appropriate thread. With process-based threads libraries, it is extremely difficult to implement proper signal delivery semantics. Similarly, multi-level threads libraries cannot implement proper signal delivery without explicit kernel support. Solaris has tried to solve the problem of signal delivery in a number of ways, but there were correctness issues until Solaris 2.5. In that version and after, the kernel keeps track of a special LWP for each multi-threaded process, and all signals are delivered to that LWP. From there, the UTS simulates signal delivery to the correct thread, if any.

The KSE framework provides a novel (simple) method of signal delivery. Signals can simply be delivered as part of the normal upcall vectoring. From there, the UTS can simulate final delivery to the correct thread, if any.

3.8 Kernel scheduler

The kernel scheduler needs a major overhaul in order to support KSEs. Support for the following features is necessary:

- Soft affinity. The scheduler should try to keep processes running on the same processors if there is the possibility of reaping benefits from increased cache locality.
- Hard affinity (binding). Some programs may wish to bind a KSE (pedantically speaking, a KSEG is bound) to a particular processor in order to improve cache locality, or to tighten the bounds on a real-time task. Bound KSEs are run only on one CPU, unless priority inheritance forces it to be run on another CPU in order to avoid deadlock due to priority inversion.
- Increased parallelism. The current scheduler can only be executed by one processor at a time, which makes it a severe bottleneck, depending on the type of system load.

Part of the solution is to implement per-CPU scheduling queues. Each CPU has an associated queue for softly affined KSEs, as well as a queue for bound KSEs. In addition, there is a global queue for fixed-priority KSEs, such as interrupt threads and real-time KSEs. Fixed-priority KSEs prefer to run on the same CPU as the last time they were run, but scheduling issues keep more complex affinity algorithms from being beneficial for this class of KSEs. See Figure 9 for a simplistic representation of the various scheduling queues. The diagram ignores some details such as split queues for various priorities.

Each CPU schedules from its own queues, and resorts to stealing runnable softly affined KSEs from other CPUs if there are no runnable KSEs. Every so often (exact number to be determined by benchmarking, but a likely number is 5 seconds), CPU loads are calculated, and if the loads are too disparate, the under-loaded CPU(s) steal additional KSEs from the overloaded CPU(s) in an attempt to balance the load. Instantaneous per-CPU load is defined as the number of runnable timesharing KSEs in a CPU's queues. Instantaneous system load is the sum of the instantaneous per-CPU loads. Note that normal (non-balancing) KSE stealing is adequate to keep the system busy if there is enough work to do, but that without periodic load balancing, time sharing becomes unfair if there aren't approximately equal CPU loads.

This is essentially the approach that was taken in DEC OSF/1 3.0 [Denham], and it seems to have worked well there. One notable difference between our kernel and OSF/1 is that our interrupts are backed by threads, whereas OSF/1 keeps *spl*()s (IPLs in their terminology). However, this doesn't affect the design, since OSF/1 already has to handle real-time scheduling.

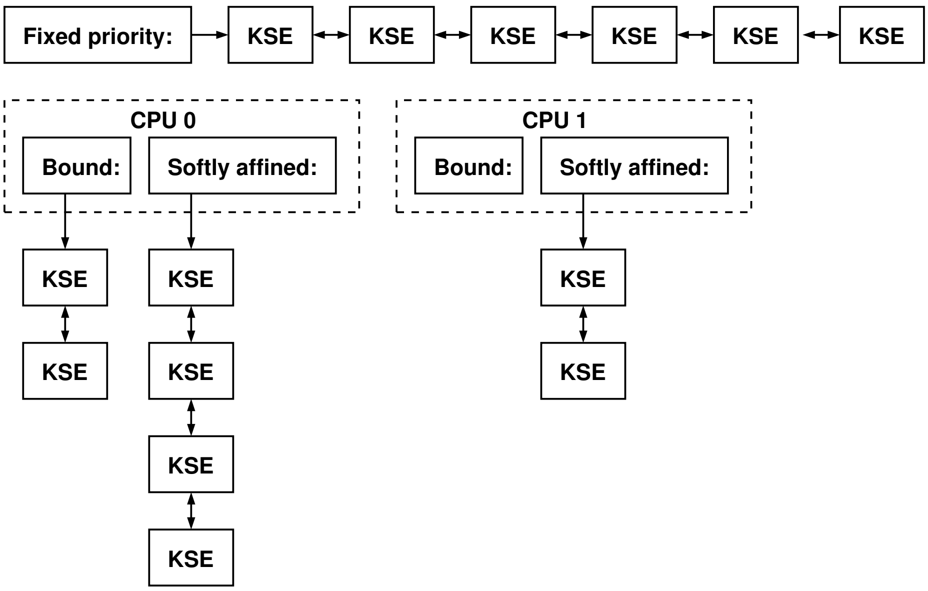

Figure 9: Kernel scheduler queues

4 Summary

Discussion about improved threads support in FreeBSD has been ongoing for several years. The KSE project aims to implement a kernel mechanism that allows the kernel and userland to support threaded processes and communicate with each other effectively so that the necessary information is available for both to do their jobs efficiently and correctly.

Glossary

KSE: Kernel-scheduled entity.

KSEC: Kernel-scheduled entity context.

KSEG: Kernel-scheduled entity group.

PCB: Process control block.

SA: Scheduler activation.

TCB: Thread control block.

UTS: Userland threads scheduler. Userland, multi-level, and scheduler activation-based threads libraries all have a UTS.

References

[Anderson] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*, ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, Pages 53-79.

[Boykin] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso, *Programming under Mach*, Addison-Wesley Publishing Company, Inc. (1993).

[Butenhof] David R. Butenhof, *Programming with POSIX threads*, Addison Wesley Longman, Inc. (1997).

[Denham] Jeffrey M. Denham, Paula Long, and James A. Woodward, *DEC OSF/1 Symmetric Multiprocessing*, Digital Technical Journal, Vol. 6, No. 3.

[Kleiman] Steve Kleiman, Devang Shah, and Bart Smaalders, *Programming with Threads*, SunSoft Press (1996).

[Mauro] Jim Mauro and Richard McDougall, *Solaris Internals*, Sun Microsystems Press (2001).

[McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing Company, Inc. (1996).

[Vahalia] Uresh Vahalia, *UNIX^{TM} Internals: The New Frontiers*, Prentice-Hall, Inc. (1996).