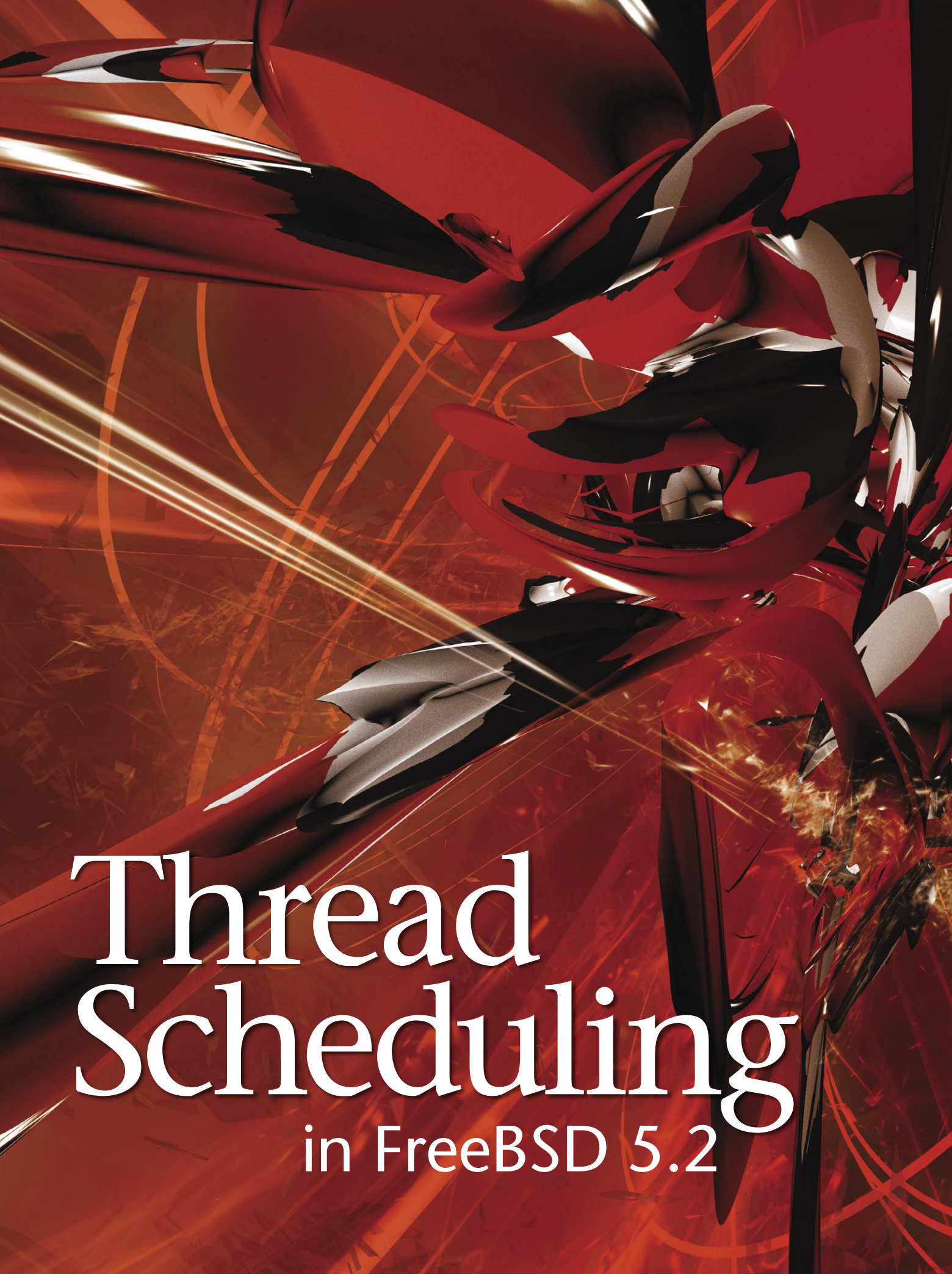

Thread
Scheduling
in FreeBSD 5.2

MARSHALL KIRK McKUSICK, CONSULTANT
GEORGE V. NEVILLE-NEIL, CONSULTANT

A busy system makes thousands of scheduling decisions per second, so the speed with which scheduling decisions are made is critical to the performance of the system as a whole. This article—excerpted from the forthcoming book, *The Design and Implementation of the FreeBSD Operating System*—uses the example of the open source FreeBSD system to help us understand thread scheduling. The original FreeBSD scheduler was designed in the 1980s for large uniprocessor systems. Although it continues to work well in that environment today, the new ULE scheduler was designed specifically to optimize multiprocessor and multithread environments.[1] This article first studies the original FreeBSD scheduler, then describes the new ULE scheduler. The article does not describe the realtime scheduler that is also available in FreeBSD.

To help get a better handle on thread scheduling, we take a look at how FreeBSD 5.2 handles it.

Thread Scheduling in FreeBSD 5.2

Other Unix systems have added a dynamic scheduler switch that must be traversed for every scheduling decision. To avoid this overhead, FreeBSD requires that the scheduler be selected at the time the kernel is built. Thus, all calls into the scheduling code are resolved at compile time rather than going through the overhead of an indirect function call for every scheduling decision. By default, kernels up through FreeBSD 5.1 use the scheduler. Beginning with FreeBSD 5.2, the ULE scheduler is used by default.

THE ORIGINAL FREEBSD SCHEDULER

All threads that are runnable are assigned a scheduling priority that determines in which run queue they are placed. In selecting a new thread to run, the system scans the run queues from highest to lowest priority and chooses the first thread on the first nonempty queue. If multiple threads reside on a queue, the system runs them *round robin*—that is, it runs them in the order that they are found on the queue, with equal amounts of time allowed. If a thread blocks, it is not put back onto any run queue. If a thread uses up the *time quantum* (or *time slice*) it is allowed, it is placed at the end of the queue from which it came, and the thread at the front of the queue is selected to run.

The shorter the time quantum, the better the interactive response. However, longer time quanta provide higher system throughput because the system will have less overhead from doing context switches, and processor caches will be flushed less often. The time quantum FreeBSD uses is 0.1 second. This value was empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors. Perhaps surprisingly, the time quantum has remained unchanged over the past 20 years. Although the time quantum was originally selected on centralized timesharing systems with many users, it is still correct for decentralized workstations today. While workstation users expect a response time faster than that anticipated by the timesharing users of 20 years ago, the shorter run queues on the typical workstation make a shorter quantum unnecessary.

TIME-SHARE THREAD SCHEDULING

The FreeBSD time-share-scheduling algorithm is based on *multilevel feedback queues*. The system adjusts the priority of a thread dynamically to reflect resource requirements (e.g., being blocked awaiting an event) and the amount of resources consumed by the thread (e.g., CPU time). Threads are moved between run queues based on changes in their scheduling priority (hence the word *feedback* in the name *multilevel feedback queue*). When a thread other than the currently running thread attains a higher priority (by having that priority either assigned or given when it is awakened), the system switches to that thread immediately if the current thread is in user mode. Otherwise, the system switches to the higher-priority thread as soon as the current thread exits the kernel. The system tailors this *short-term scheduling algorithm* to favor interactive jobs by raising the scheduling priority of threads that are blocked waiting for I/O for one or more seconds and by lowering the priority of threads that accumulate significant amounts of CPU time.

Short-term thread scheduling is broken into two parts. The next section describes when and how a thread's scheduling priority is altered; the section after that describes the management of the run queues and the interaction between thread scheduling and context switching.

CALCULATIONS OF THREAD PRIORITY

A thread's scheduling priority is determined directly by two values associated with the thread structure: kg_estcpu and kg_nice. The value of kg_estcpu provides an estimate of the recent CPU utilization of the thread. The value of kg_nice is a user-settable weighting factor that ranges numerically between -20 and 20. The normal value for kg_nice is zero. Negative values increase a thread's priority, whereas positive values decrease its priority.

A thread's user-mode scheduling priority is calculated after every four clock ticks (typically 40 milliseconds) that it has been found running by this equation:

$$kg\_user\_pri = PRI\_MIN\_TIMESHARE + \left\lceil \frac{kg\_estcpu}{4} \right\rceil + 2 \times kg\_nice$$

Values less than PRI_MIN_TIMESHARE (160) are set to PRI_MIN_TIMESHARE (see table 1); values greater than PRI_MAX_TIMESHARE (223) are set to PRI_MAX_TIMESHARE. This calculation causes the priority to decrease linearly based on recent CPU utilization. The user-controllable kg_nice parameter acts as a limited weighting factor. Negative values retard the effect of heavy CPU utilization by offsetting the additive term containing kg_estcpu. Otherwise, if we ignore the second term, kg_nice simply shifts the priority by a constant factor.

The CPU utilization, kg_estcpu, is incremented each time that the system clock ticks and the thread is found to be executing. In addition, kg_estcpu is adjusted once per second via a digital decay filter. The decay causes about 90 percent of the CPU usage accumulated in a one-second interval to be forgotten over a period of time that is dependent on the system *load average*. To be exact, kg_estcpu is adjusted according to

$$kg\_estcpu = \frac{(2 \times load)}{(2 \times load + 1)}\ kg\_estcpu + kg\_nice,$$

where the *load* is a sampled average of the sum of the lengths of the run queue and of the short-term sleep queue over the previous one-minute interval of system operation.

To understand the effect of the decay filter, consider the case where a single compute-bound thread monopolizes the CPU. The thread's CPU utilization will accumulate clock ticks at a rate dependent on the clock frequency. The load average will be effectively 1, resulting in a decay of

$$kg\_estcpu = 0.66 \times kg\_estcpu + kg\_nice.$$

If we assume that the thread accumulates T_i clock ticks over time interval i and that kg_nice is zero, then the CPU utilization for each time interval will count into the current value of kg_estcpu according to

$kg\_estcpu = 0.66 \times T_0$
$kg\_estcpu = 0.66 \times (T_1 + 0.66 \times T_0) = 0.66 \times T_1 + 0.44 \times T_0$
$kg\_estcpu = 0.66 \times T_2 + 0.44 \times T_1 + + 0.30 \times T_0$
$kg\_estcpu = 0.66 \times T_3 + . . . + 0.20 \times T_0$
$kg\_estcpu = 0.66 \times T_4 + . . . + 0.13 \times T_0.$

Thus, after five decay calculations, only 13 percent of T_0 remains present in the current CPU utilization value for the thread. Since the decay filter is applied once per second, about 90 percent of the CPU utilization is forgotten after five seconds.

TABLE 1 Thread-scheduling Classes

Range	Class	Thread type
0-63	ITHD	bottom-half kernel (interrupt)
64-127	kern	top-half kernel
128-159	realtime	realtime user
160-223	timeshare	time-sharing user
224-255	idle	idle user

Threads that are runnable have their priority adjusted periodically as just described. However, the system ignores threads that are blocked awaiting an event: these threads cannot accumulate CPU usage, so an estimate of their filtered CPU usage can be calculated in one step. This optimization can significantly reduce a system's scheduling overhead when many blocked threads are present. The system recomputes a thread's priority when that thread is awakened and has been sleeping for longer than one second. The system maintains a value, kg_slptime, that is an estimate of the time a thread has spent blocked waiting for an event. The value of kg_slptime is set to zero when a thread calls sleep() and is incremented once per second while the thread remains in a SLEEPING or STOPPED state. When the thread is awakened, the system computes the value of kg_estcpu according to

$$kg\_estcpu = \left[\frac{(2 \times load)}{(2 \times load + 1)}\right]^{kg\_slptime} \times kg\_estcpu,$$

and then recalculates the scheduling priority using this equation. This analysis ignores the influence of kg_nice; also, the *load* used is the current load average rather than the load average at the time that the thread blocked.

THE ULE SCHEDULER

The ULE scheduler was developed as part of the overhaul of FreeBSD to support SMP (symmetric multiprocessing). A new scheduler was undertaken for several reasons:

- To address the need for processor affinity in SMP systems

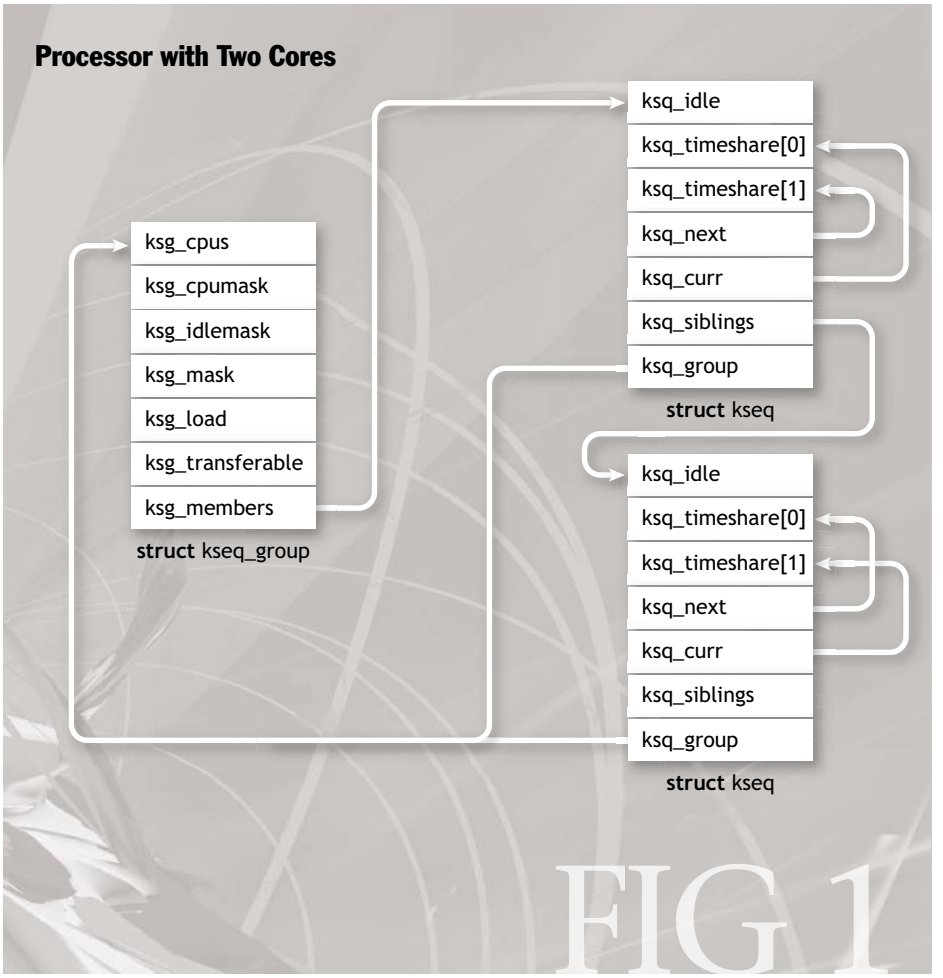
Thread Scheduling in FreeBSD 5.2

- To provide better support for SMT (*symmetric multithreading*)—processors with multiple, on-chip CPU cores
- To improve the performance of the scheduling algorithm so that it is no longer dependent on the number of threads in the system

The goal of a multiprocessor system is to apply the power of multiple CPUs to a problem, or set of problems, to achieve a result in less time than it would run on a single-processor system. If a system has the same number of runnable threads as it does CPUs, then achieving this goal is easy. Each runnable thread gets a CPU to itself and runs to completion. Typically, many runnable threads are competing for a few processors. One job of the scheduler is to ensure that the CPUs are always busy and not wasting their cycles. When a thread completes its work, or is blocked waiting for resources, it is removed from the processor on which it was running. While a thread is running on a processor, it brings its working set—the instructions it is executing and the data on which it is operating—into the memory cache of the CPU.

Migrating a thread has a cost. When a thread is moved from one processor to another, its in-cache working set is lost and must be removed from the processor on which it was running and then loaded into the new CPU to which it has been migrated. The performance of an SMP system with a naive scheduler that does not take this cost into account can fall beneath that of a single-processor system. The term *processor affinity* describes a scheduler that migrates threads only when necessary to give an idle processor something to do.

Many microprocessors now provide support for symmetric multithreading where the processor is built out of multiple CPU cores, each of which can execute a thread. The CPU cores in an SMT processor share all the processor's resources, such as memory caches and access to main memory, so they are more tightly synchronized than the processors in an SMP system. From a thread's perspective,



it does not know that other threads are running on the same processor because the processor is handling them independently. The one piece of code in the system that needs to be aware of the multiple cores is the scheduling algorithm. The SMT case is a slightly different version of the processor affinity problem presented by an SMP system. Each CPU core can be seen as a processor with its own set of threads. In an SMP system composed of CPUs that support SMT, the scheduler treats each core on a processor as a less powerful resource but one to which it is cheaper to migrate threads.

The original FreeBSD scheduler maintains a global list of threads that it traverses once per second to recalculate their priorities. The use of a single list for all threads means that the performance of the scheduler is dependent on the number of tasks in the system, and as the number of tasks grows, more CPU time must be spent in the scheduler maintaining the list. A design goal of the ULE scheduler was to avoid the need to consider all the runnable threads in the system to make a scheduling decision.

The ULE scheduler creates a set of three queues for each CPU in the system. Having per-processor queues makes it possible to implement processor affinity in an SMP system.

One queue is the *idle queue*, where all idle threads are stored. The other two queues are designated *current* and *next*. Threads are picked to run, in priority order, from the *current* queue until it is empty, at which point the *current* and *next* queues are swapped and scheduling is started again. Threads in the *idle* queue are run only when the other two queues are empty. Realtime and interrupt threads are always inserted into the *current* queue so that they will have the least possible scheduling latency. Interactive threads are also inserted into the *current* queue to keep the interactive response of the system acceptable. A thread is considered to be interactive if the ratio of its voluntary sleep time versus its runtime is below a certain threshold. The interactivity threshold is defined in the ULE code and is not configurable. ULE uses two equations to compute the interactivity score of a thread. For threads whose sleep time exceeds their runtime, the following equation is used:

$$\text{interactivity score} = \frac{\text{scaling factor}}{\frac{\text{sleep}}{\text{run}}}$$

When a thread's runtime exceeds its sleep time, the following equation is used instead:

$$\text{interactivity score} = \frac{\text{scaling factor}}{\frac{\text{sleep}}{\text{run}}} + \text{scaling factor}$$

The scaling factor is the maximum interactivity score divided by two. Threads that score below the interactivity threshold are considered to be interactive; all others are noninteractive. The sched_interact_update() routine is called at several points in a thread's existence—for example, when the thread is awakened by a wakeup() call—to update the thread's runtime and sleep time. The sleep-time and runtime values are allowed to grow only to a certain limit. When the sum of the runtime and sleep time passes the limit, the values are reduced to bring them back into range. An interactive thread whose sleep history was not remembered at all would not remain interactive, resulting in a poor user experience. Remembering an interactive thread's sleep time for too long would allow the thread to have more than its fair share of the CPU. The amount of history that is kept and the interactivity threshold are the two values that most strongly influence a user's interactive experience on the system.

Noninteractive threads are put into the *next* queue and are scheduled to run when the queues are switched. Switching the queues guarantees that a thread gets to run at least once every two queue switches regardless of priority, which ensures fair sharing of the processor.

Two mechanisms are used to migrate threads among multiple processors. When a CPU has no work to do in any of its queues, it marks a bit in a bitmask shared by all processors that says it is idle. Whenever an active CPU is about to add work to its own run queue, it first checks to see if it has excess work and if another processor in the system is idle. If an idle processor is found, then the thread is migrated to the idle processor using an IPI (*interprocessor interrupt*). Making a migration decision by inspecting a shared bitmask is much faster than scanning the run queues of all the other processors. Seeking

Thread Scheduling in FreeBSD 5.2

out idle processors when adding a new task works well because it spreads the load when it is presented to the system.

The second form of migration, called *push migration*, is done by the system on a periodic basis and more aggressively offloads work to other processors in the system. Twice per second the sched_balance() routine picks the most-loaded and least-loaded processors in the system and equalizes their run queues. The balancing is done between only two processors because it was thought that two-processor systems would be the most common and to prevent the balancing algorithm from being too complex and adversely affecting the performance of the scheduler. Push migration ensures fairness among the runnable threads. For example, with three runnable threads on a two-processor system, it would be unfair for one thread to get a processor to itself while the other two had to share the second processor. By pushing a thread from the processor with two threads to the processor with one thread, no single thread would get to run alone indefinitely.

Handling the SMT case is a derivative form of load balancing among full-fledged CPUs and is handled by *processor groups*. Each CPU core in an SMT processor is given its own kseq structure, and these structures are grouped under a kseq group structure. An example of a single processor with two cores is shown in figure 1. In an SMP system with multiple SMT-capable processors there would be one processor group per CPU. When the scheduler is deciding to which processor or core to migrate a thread, it will try to pick a core on the same processor before picking one on another processor because that is the lowest-cost migration path.

CONCLUSION

The original FreeBSD scheduler continues to work well in many situations. Its elegant simplicity (it is written in fewer than 100 lines of C code) is easy to understand and difficult to cheat. It does not, however, understand the complexity of multiprocessor environments; thus, it cannot be adapted to do affinity processing or assign a share of the processor to a group of threads. The ULE scheduler was developed to handle these needs more effectively.

ACKNOWLEDGMENTS

This article is a partial excerpt from Chapter 4, Process Management, from *The Design and Implementation of the FreeBSD Operating System,* by Marshall Kirk McKusick and George Neville-Neil (Table 1 is from an earlier section, *Process State*, in the same chapter). Reprinted with permission from Pearson Education Inc. (0-201-70245-2). Copyright 2005. To learn more: http://www.awprofessional.com/title/0201702452. Q

REFERENCE

1. Roberson, J. ULE: A modern scheduler for FreeBSD. *Proceedings of BSDCon 2003* (September 2003).

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MARSHALL KIRK McKUSICK has a Berkeley, California-based consultancy, writes books and articles, and teaches classes on Unix- and BSD-related subjects. His work with Unix stretches more than 20 years. While at the University of California at Berkeley, he implemented the 4.2 BSD Fast File System and was the research computer scientist at the Berkeley CSRG (Computer Systems Research Group) overseeing the development and release of 4.3 BSD and 4.4 BSD. His areas of interest are the virtual-memory system and the file system. He earned his undergraduate degree in electrical engineering from Cornell University and did his graduate work at UC Berkeley, where he received master's degrees in computer science and business administration and a doctorate in computer science. He is president of the Usenix Association and a member of ACM and IEEE.

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and networking. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts. He is a member of the ACM, the Usenix Association, and the IEEE. He is an avid bicyclist and traveler who splits his time between Tokyo and San Francisco.