The witness module also verifies that no locks that prohibit sleeping are held when requesting a sleep lock or voluntarily going to sleep.

The witness module can be configured to either panic or drop into the kernel debugger when an order violation occurs or some other witness check fails. When running the debugger, the witness module can output the list of locks held by the current thread to the console along with the filename and line number at which each lock was last acquired. It can also dump the current order list to the console. The code first displays the lock order tree for all the sleep locks. Then it displays the lock order tree for all the spin mutexes. Finally, it displays a list of locks that have not yet been acquired.

4.4 Thread Scheduling

The FreeBSD scheduler has a well-defined set of kernel-application programming interfaces (kernel APIs) that allow it to support different schedulers. Since FreeBSD 5.0, the kernel has had two schedulers available:

• The ULE scheduler first introduced in FreeBSD 5.0 and found in the file **/sys/kern/sched_ule.c** [Roberson, 2003]. The name is not an acronym. If the underscore in its filename is removed, the rationale for its name becomes apparent. This scheduler is used by default and is described later in this section.

• The traditional 4.4BSD scheduler found in the file **/sys/kern/sched_4bsd.c**. This scheduler is still maintained but no longer used by default.

Because a busy system makes millions of scheduling decisions per second, the speed with which scheduling decisions are made is critical to the performance of the system as a whole. 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.

The Low-Level Scheduler

Scheduling is divided into two parts: a simple low-level scheduler that runs frequently and a more complex high-level scheduler that runs at most a few times per second. The low-level scheduler runs every time a thread blocks and a new thread must be selected to run. For efficiency when running thousands of times per second, it must make its decision quickly with a

minimal amount of information. To simplify its task, the kernel maintains a set of ***run queue***s for each CPU in the system that are organized from high to low priority. When a task blocks on a CPU, the low-level scheduler's sole responsibility is to select the thread from the highest-priority non-empty run queue for that CPU. The high-level scheduler is responsible for setting the thread priorities and deciding on which CPU's run queue they should be placed. Each CPU has its own set of run queues to avoid contention for access when two CPUs both need to select a new thread to run at the same time. Contention between run queues occurs only when the high-level scheduler decides to move a thread from the run queue of one CPU to the run queue of another CPU. The kernel tries to avoid moving threads between CPUs as the loss of its CPU-local caches slows it down.

All threads that are runnable are assigned a scheduling priority and a CPU by the high-level scheduler that determines in which run queue they are placed. In selecting a new thread to run, the low-level scheduler scans the run queues of the CPU needing a new thread 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. Instead, it is placed on a *turnstile* or a *sleepqueue*. If a thread uses up the ***time quantum*** (or ***time slice***) allowed it, 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 incur less overhead from doing context switches and processor caches will be flushed less often. The time quantum used by FreeBSD is adjusted by the high-level scheduler as described later in this subsection.

Thread Run Queues and Context Switching

The kernel has a single set of run queues to manage all the thread scheduling classes shown in Table 4.2. The scheduling-priority calculations described in the previous section are used to order the set of timesharing threads into the priority ranges between 120 and 223. The real-time threads and the idle threads priorities are set by the applications themselves but are constrained by the kernel to be within the ranges 48 to 79 and 224 to 255, respectively. The number of queues used to hold the collection of all runnable threads in the system affects the cost of managing the queues. If only a single (ordered) queue is maintained, then selecting the next runnable thread becomes simple but other operations become expensive. Using 256 different queues can significantly increase the cost of identifying the next thread to run. The system uses 64 run queues, selecting a run queue for a thread by dividing the thread's priority by 4. To save time, the threads on each queue are not further sorted by their priorities.

The run queues contain all the runnable threads in main memory except the currently running thread. Figure 4.5 shows how each queue is organized as a doubly linked list of thread structures. The head of each run queue is kept in an array. Associated with this array is a bit vector, *rq_status*, that is used in identifying the nonempty run queues. Two routines, *runq_add*() and *runq_remove*(), are used to place a thread at the tail of a run queue, and to take a thread off the head of a run queue. The heart of the scheduling algorithm is the *runq_choose*() routine. The *runq_choose*() routine is responsible for selecting a new thread to run; it operates as follows:

1. Ensures that our caller acquired the lock associated with the run queue.

2. Locates a nonempty run queue by finding the location of the first nonzero bit in the *rq_status* bit vector. If *rq_status* is zero, there are no threads to run, so selects an ***idle loop*** thread.

3. Given a nonempty run queue, removes the first thread on the queue.

4. If this run queue is now empty as a result of removing the thread, clears the appropriate bit in *rq_status*.

5. Returns the selected thread.

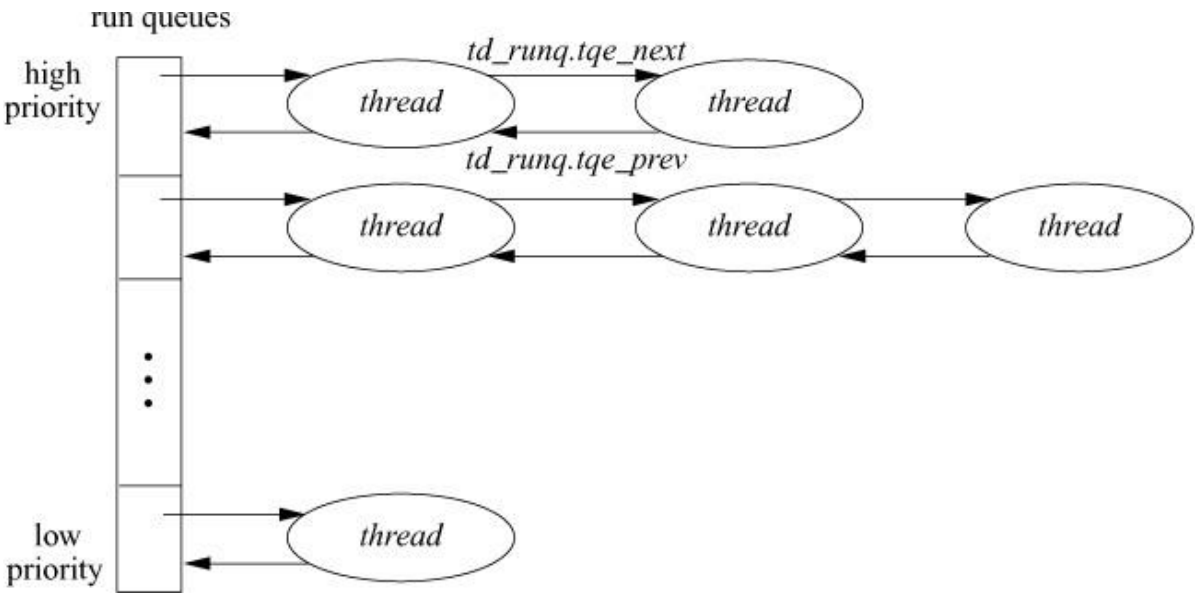

Figure 4.5 Queueing structure for runnable threads.

The context-switch code is broken into two parts. The machine-independent code resides in *mi_switch*(); the machine-dependent part resides in *cpu_switch*(). On most architectures, *cpu_switch*() is coded in assembly language for efficiency.

Given the *mi_switch*() routine and the thread-priority calculations, the only missing piece in the scheduling facility is how the system forces an involuntary context switch. Remember that voluntary context switches occur when a thread calls the *sleep*() routine. *Sleep*() can be invoked only by a runnable thread, so *sleep*() needs only to place the thread on a sleep queue and to invoke *mi_switch*() to schedule the next thread to run. Often, an interrupt thread will not want to *sleep*() itself but will be delivering data that will cause the kernel to want to run a different thread than the one that was running before the interrupt. Thus, the kernel needs a mechanism to request that an involuntary context switch be done at the conclusion of the interrupt.

This mechanism is handled by setting the currently running thread's TDF_NEEDRESCHED flag and then posting an ***asynchronous system trap*** (***AST***). An AST is a trap that is delivered to a thread the next time that thread is preparing to return from an interrupt, a trap, or a system call. Some architectures support ASTs directly in hardware; other systems emulate ASTs by checking an AST flag at the end of every system call, trap, and interrupt. When the hardware AST trap occurs or the AST flag is set, the *mi_switch*() routine is called instead of the current thread resuming execution. Rescheduling requests are made by the *sched_lend_user_prio*(), *sched_clock*(), *sched_setpreempt*(), and *sched_affinity*() routines.

With the advent of multiprocessor support, FreeBSD can preempt threads executing in kernel mode. However, such preemption is generally not done for threads running in the timesharing class, so the worst-case real-time response to events when running with the timeshare scheduler is defined by the longest path through the top half of the kernel. Since the system guarantees no upper bounds on the duration of a system call, when running with just the timeshare scheduler FreeBSD is decidedly not a hard real-time system.

Real-time and interrupt threads do preempt lower-priority threads. The longest path that preemption is disabled for real-time and interrupt threads is defined by the longest time a spinlock is held or a critical section is entered. Thus, when using real-time threads, microsecond real-time deadlines can be met. The kernel can be configured to preempt timeshare threads executing in the kernel with other higher-priority timeshare threads. This option is not used by default as the increase in context switches adds overhead and does not help make timeshare threads response time more predictable.

Timeshare Thread Scheduling

The goal of a multiprocessing 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, there are

many runnable threads competing for a few processors. One job of the scheduler is to ensure that the CPUs are always busy and are 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 CPU's memory cache. Migrating a thread has a cost. When a thread is moved from one CPU to another, its CPU-cache working set is lost and must be removed from the CPU on which it was running and then loaded into the new CPU to which it has been migrated. The performance of a multiprocessing 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 only migrates threads when necessary to give an idle processor something to do.

A multiprocessing system may be built with multiple processor chips. Each processor chip may have multiple CPU cores, each of which can execute a thread. The CPU cores on a single processor chip share many of the processor's resources, such as memory caches and access to main memory, so they are more tightly synchronized than the CPUs on other processor chips.

Handling processor chips with multiple CPUs is a derivative form of load balancing among CPUs on different chips. It is handled by maintaining a hierarchy of CPUs. The CPUs on the same chip are the cheapest between which to migrate threads. Next down in the hierarchy are processor chips on the same motherboard. Below them are chips connected by the same backplane. The scheduler supports an arbitrary depth hierarchy as dictated by the hardware. When the scheduler is deciding to which processor to migrate a thread, it will try to pick a new processor higher in the hierarchy because that is the lowest-cost migration path.

From a thread's perspective, it does not know that there are other threads 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 CPUs is the scheduling algorithm. In particular, the scheduler treats each CPU on a chip as one on which it is cheaper to migrate threads than it would be to migrate the thread to a CPU on another chip. The mechanism for getting tighter affinity between CPUs on the same processor chip versus CPUs on other processor chips is described later in this section.

The traditional FreeBSD scheduler maintains a global list of runnable threads that it traverses once per second to recalculate their priorities. The use of a single list for all runnable threads means that the performance of the scheduler is dependent on the number of tasks in the system, and as the number of tasks grow, more CPU time must be spent in the scheduler maintaining the list.

The ULE scheduler was developed during FreeBSD 5.0 with major work continuing into FreeBSD 9.0, spanning 10 years of development. The scheduler was developed to address shortcomings of the traditional BSD scheduler on multiprocessor systems. A new scheduler was undertaken for several reasons:

• To address the need for processor affinity in multiprocessor systems

• To supply equitable distribution of load between CPUs on multiprocessor systems

• To provide better support for processors with multiple CPU cores on a single chip

• To improve the performance of the scheduling algorithm so that it is no longer dependent on the number of threads in the system

• To provide interactivity and timesharing performance similar to the traditional BSD scheduler.

The traditional BSD scheduler had good interactivity on large timeshare systems and single-user desktop and laptop systems. However, it had a single global run queue and consequently a single global scheduler lock. Having a single global run queue was slowed both by contention for the global lock and by difficulties implementing CPU affinity.

The priority computation relied on a single global timer that iterated over every runnable thread in the system and evaluated its priority while holding several highly contended locks. This approach became slower as the number of runnable threads increased. While the priority calculations were being done, processes could not *fork* or *exit* and CPUs could not context switch.

The ULE scheduler can logically be thought of as two largely orthogonal sets of algorithms; those that manage the affinity and distribution of threads among CPUs and those that are responsible for the order and duration of a thread's runtime. These two sets of algorithms work in concert to provide a balance of low latency, high throughput, and good resource utilization. The remainder of the scheduler is event driven and uses these algorithms to implement various decisions according to changes in system state.

The goal of equalling the exceptional interactive behavior and throughput of the traditional BSD scheduler in a multiprocessor-friendly and constant-time implementation was the most challenging and time consuming part of ULE's development. The interactivity, CPU utilization estimation, priority, and time slice algorithms together implement the timeshare scheduling policy.

The behavior of threads is evaluated by ULE on an event-driven basis to differentiate interactive and batch threads. Interactive threads are those that are thought to be waiting for and responding to user input. They require low latency to achieve a good user experience. Batch threads are those that tend to consume as much CPU as they are given and may be background jobs. A good example of the former is a text editor, and for the latter, a compiler. The scheduler must use imperfect heuristics to provide a gradient of behaviors based on a best guess of the category to which a given thread fits. This categorization may change frequently during the lifetime of a thread and must be responsive on timescales relevant to people using the system.

The algorithm that evaluates interactivity is called the interactivity score. The interactivity score is the ratio of voluntary sleep time to run time normalized to a number between 0 and 100. This score does not include time waiting on the run queue while the thread is not yet the highest priority thread in the queue. By requiring explicit voluntary sleeps, we can differentiate threads that are not running because of inferior priority versus those that are periodically waiting for user input. This requirement also makes it more challenging for a thread to be marked interactive as system load increases, which is desirable because it prevents the system from becoming swamped with interactive threads while keeping things like shells and simple text editors available to administrators. When plotted, the interactivity scores derived from a matrix of possible sleep and run times becomes a three-dimensional sigmoid function. Using this approach means that interactive tasks tend to stay interactive and batch tasks tend to stay batched.

A particular challenge is complex X Window applications such as Web browsers and office productivity packages. These applications may consume significant resources for brief periods of time, however the user expects them to remain interactive. To resolve this issue, a several-second history of the sleep and run behavior is kept and gradually decayed. Thus, the scheduler keeps a moving average that can tolerate bursts of behavior but will quickly penalize timeshare threads that abuse their elevated status. A longer history allows longer bursts but learns more slowly.

The interactivity score is compared to the interactivity threshold, which is the cutoff point for considering a thread interactive. The interactivity threshold is modified by the process ***nice*** value. Positive nice values make it more challenging for a thread to be considered interactive, while negative values make it easier. Thus, the nice value gives the user some control over the primary mechanism of reducing thread-scheduling latency.

A thread is considered to be interactive if the ratio of its voluntary sleep time versus its run time 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 run time, Eq 4.1 is used:

$$interactivity\ score = \frac{scaling\ factor}{sleep\ /\ run} \qquad \text{(Eq. 4.1)}$$

When a thread's run time exceeds its sleep time, Eq. 4.2 is used instead:

$$interactivity\ score = \frac{scaling\ factor}{run\ /\ sleep} + scaling\ factor \qquad \text{(Eq. 4.2)}$$

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 threads existence—for example, when the thread is awakened by a *wakeup*() call—to update the thread's run time and sleep time. The sleep- and run-time values are only allowed to grow to a certain limit. When the sum of the run time and sleep time pass the limit, they 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 get 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.

Priorities are assigned according to the thread's interactivity status. Interactive threads have a priority that is derived from the interactivity score and are placed in a priority band above batch threads. They are scheduled like real-time round-robin threads. Batch threads have their priorities determined by the estimated CPU utilization modified according to their process nice value. In both cases, the available priority range is equally divided among possible interactive scores or percent-cpu calculations, both of which are values between 0 and 100. Since there are fewer than 100 priorities available for each class, some values share priorities. Both computations roughly assign priorities according to a history of CPU utilization but with different longevities and scaling factors.

The CPU utilization estimator accumulates run time as a thread runs and decays it as a thread sleeps. The utilization estimator provides the percent-cpu values displayed in **top** and **ps**. ULE delays the decay until a thread wakes to avoid periodically scanning every thread in the system. Since this delay leaves values unchanged for the duration of sleeps, the values must also be decayed before any user process inspects them. This approach preserves the constant-time and event-driven nature of the scheduler.

The CPU utilization is recorded in the thread as the number of ticks, typically 1 millisecond, during which a thread has been running, along with window of time defined as a first and last tick. The scheduler attempts to keep roughly 10 seconds of history. To accomplish decay, it waits until there are 11 seconds of history and then subtracts one-tenth of the tick value while moving the first tick forward 1 second. This inexpensive, estimated moving-average algorithm has the property of allowing arbitrary update intervals. If the utilization information is inspected after more than the update interval has passed, the tick value is zeroed. Otherwise, the number of seconds that have passed divided by the update interval is subtracted.

The scheduler implements round-robin through the assignment of time slices. A time slice is a fixed interval of allowed run time before the scheduler will select another thread of equal priority to run. The time slice prevents starvation among equal priority threads. The time slice times the number of runnable threads in a given priority defines the maximum latency a thread of that priority will experience before it can run. To bound this latency, ULE dynamically adjusts the size of slices it dispenses based on system load. The time slice has a minimum value to prevent thrashing and balance throughput with latency. An interrupt handler calls the scheduler to evaluate the time slice during every statclock tick. Using the stat-clock to evaluate the time slice is a stochastic approach to slice accounting that is efficient but only grossly accurate.

The scheduler must also work to prevent starvation of low-priority batch jobs by higher-priority batch jobs. The traditional BSD scheduler avoided starvation by periodically iterating over all threads waiting on the run queue to elevate the low-priority threads and decrease the priority of higher-priority threads that had been monopolizing the CPU. This algorithm violates the desire to run in constant time independent of the number of system threads. As a result, the run queue for batch-policy timeshare threads is kept in a similar fashion to the system callwheel, also known as a calendar queue. A calendar queue is one in which the queue's head and tail rotate according to a clock or period. An element can be inserted into a calendar queue many positions away from the head and gradually migrate toward the head. Because this run queue is special purpose, it is kept separately from the real-time and idle queues while interactive threads are kept along with the real-time threads until they are no longer considered interactive.

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

One array of queues is the ***idle queue***, where all idle threads are stored. The array is arranged from highest to lowest priority. The second array of queues is designated the realtime queue. Like the idle queue, it is arranged from highest to lowest priority.

The third array of queues is designated the timeshare queue. Rather than being arranged in priority order, the timeshare queues are managed as a calendar queue. A pointer references the

current entry. The pointer is advanced once per system tick, although it may not advance on a tick until the currently selected queue is empty. Since each thread is given a maximum time slice and no threads may be added to the current position, the queue will drain in a bounded amount of time. This requirement to empty the queue before advancing to the next queue means that the wait time a thread experiences is not only a function of its priority but also the system load.

Insertion into the timeshare queue is defined by the relative difference between a thread's priority and the best possible timeshare priority. High-priority threads will be placed soon after the current position. Low-priority threads will be placed far from the current position. This algorithm ensures that even the lowest-priority timeshare thread will eventually make it to the selected queue and execute in spite of higher-priority timeshare threads being available in other queues. The difference in priorities of two threads will determine their ratio of run-time. The higher-priority thread may be inserted ahead of the lower-priority thread multiple times before the queue position catches up. This run-time ratio is what grants timeshare CPU hogs with different nice values, different proportional shares of the CPU.

These algorithms taken together determine the priorities and run times of timesharing threads. They implement a dynamic tradeoff between latency and throughput based on system load, thread behavior, and a range of effects based on user-scheduling decisions made with nice. Many of the parameters governing the limits of these algorithms can be explored in real time with the *sysctl kern.sched* tree. The rest are compile-time constants that are documented at the top of the scheduler source file (**/sys/kern/sched_ule.c**).

Threads are picked to run, in priority order, from the realtime queue until it is empty, at which point threads from the currently selected timeshare queue will be run. Threads in the idle queues are run only when the other two arrays of queues are empty. Real-time and interrupt threads are always inserted into the realtime queues so that they will have the least possible scheduling latency. Interactive threads are also inserted into the realtime queue to keep the interactive response of the system acceptable.

Noninteractive threads are put into the timeshare queues and are scheduled to run when the queues are switched. Switching the queues guarantees that a thread gets to run at least once every pass around the array of the timeshare queues regardless of priority, thus ensuring fair sharing of the processor.

Multiprocessor Scheduling

A principal goal behind the development of ULE was improving performance on multiprocessor systems. Good multiprocessing performance involves balancing affinity with utilization and the

preservation of the illusion of global scheduling in a system with local scheduling queues. These decisions are implemented using a CPU topology supplied by machine-dependent code that describes the relationships between CPUs in the system. The state is evaluated whenever a thread becomes runnable, a CPU idles, or a periodic task runs to rebalance the load. These events form the entirety of the multiprocessor-aware scheduling decisions.

The topology system was devised to identify which CPUs were symmetric multi-threading peers and then made generic to support other relationships. Some examples are CPUs within a package, CPUs sharing a layer of cache, CPUs that are local to a particular memory, or CPUs that share execution units such as in symmetric multi-threading. This topology is implemented as a tree of arbitrary depth where each level describes some shared resource with a cost value and a bitmask of CPUs sharing that resource. The root of the tree holds CPUs in a system with branches to each socket, then shared cache, shared functional unit, etc. Since the system is generic, it should be extensible to describe any future processor arrangement. There is no restriction on the depth of the tree or requirement that all levels are implemented.

Parsing this topology is a single recursive function called *cpu_search*(). It is a path-aware, goal-based, tree-traversal function that may be started from arbitrary subtrees. It may be asked to find the least- or most-loaded CPU that meets a given criteria, such as a priority or load threshold. When considering load, it will consider the load of the entire path, thus giving the potential for balancing sockets, caches, chips, etc. This function is used as the basis for all multiprocessing-related scheduling decisions. Typically, recursive functions are avoided in kernel programming because there is potential for stack exhaustion. However, the depth is fixed by the depth of the processor topology that typically does not exceed three.

When a thread becomes runnable as a result of a wakeup, unlock, thread creation, or other event, the *sched_pickcpu*() function is called to decide where it will run. ULE determines the best CPU based on the following criteria:

• Threads with hard affinity to a single CPU or short-term binding pick the only allowed CPU.

• Interrupt threads that are being scheduled by their hardware interrupt handlers are scheduled on the current CPU if their priority is high enough to run immediately.

• Thread affinity is evaluated by walking backwards up the tree starting from the last CPU on which it was scheduled until a package or CPU is found with valid affinity that can run the thread immediately.

• The whole system is searched for the least-loaded CPU that is running a lower-priority thread than the one to be scheduled.

• The whole system is searched for the least-loaded CPU.

• The results of these searches are compared to the current CPU to see if that would give a preferable decision to improve locality among the sleeping and waking threads as they may share some state.

This approach orders from most preferential to least preferential. The affinity is valid if the sleep time of the thread was shorter than the product of a time constant and a largest-cache-shared level in the topology. This computation coarsely models the time required to push state out of the cache. Each thread has a bitmap of allowed CPUs that is manipulated by **cpuset** and is passed to *cpu_search*() for every decision. The locality between sleeper and waker can improve producer/consumer type threading situations when they have shared cache state but it can also cause underutilization when each thread would run faster given its own CPU. These examples exemplify the types of decisions that must be made with imperfect information.

The next major multiprocessing algorithm runs when a CPU idles. The CPU sets a bit in a bitmask shared by all processors that says that it is idle. The idle CPU calls *tdq_idled*() to search other CPUs for work that can be migrated, or stolen in ULE terms, to keep the CPU busy. To avoid thrashing and excessive migration, the kernel sets a load threshold that must be exceeded on another CPU before some load will be taken. If any CPU exceeds this threshold, the idle CPU will search its run queues for work to migrate. The highest-priority work that can be scheduled on the idle CPU is then taken. This migration may be detrimental to affinity but improves many latency-sensitive workloads.

Work may also be pushed to an idle CPU. 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 CPU in the system is idle. If an idle CPU is found, then the thread is migrated to the idle CPU using an ***interprocessor interrupt*** (***IPI***). Making a migration decision by inspecting a shared bitmask is much faster than scanning the run queues of all the other processors. Seeking out idle processors when adding a new task works well because it spreads the load when it is presented to the system.

The last major multiprocessing algorithm is the long-term load balancer. This 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. Since the two scheduling events that distribute load only run when a thread is added and when a CPU idles, it is possible to have a long-term imbalance where more threads are running on one CPU than another. 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. To fulfill the goal of emulating a fair global run queue, ULE must periodically shuffle threads to keep the system balanced. 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. An ideal implementation would give each thread an average of 66 percent of the CPU available from a single CPU.

The long-term load balancer balances the worst path pair in the hierarchy to avoid socket-, cache-, and chip-level imbalances. It runs from an interrupt handler in a randomized interval of roughly 1 second. The interval is randomized to prevent harmonic relationships between periodic threads and the periodic load balancer. In much the same way a stochastic sampling profiler works, the balancer picks the most- and least-loaded path from the current tree position and then recursively balances those paths by migrating threads.

The scheduler must decide whether it is necessary to send an IPI when adding a thread to a remote CPU, just as it must decide whether adding a thread to the current CPU should preempt the current thread. The decision is made based on the current priority of the thread running on the target CPU and the priority of the thread being scheduled. Preempting whenever the pushed thread has a higher priority than the currently running thread results in excessive interrupts and preemptions. Thus, a thread must exceed the timesharing priority before an IPI is generated. This requirement trades some latency in batch jobs for improved performance.

A notable omission to the load balancing events is thread preemption. Preempted threads are simply added back to the run queue of the current CPU. An additional load-balancing decision can be made here. However, the runtime of the preempting thread is not known and the preempted thread may maintain affinity. The scheduler optimistically chooses to wait and assume affinity is more valuable than latency.

Each CPU in the system has its own set of run queues, statistics, and a lock to protect these fields in a *thread-queue* structure. During migration or a remote wakeup, a lock may be acquired by a CPU other than the one owning the queue. In practice, contention on these locks is rare unless the workload exhibits grossly overactive context switching and thread migration, typically suggesting a higher-level problem. Whenever a pair of these locks is required, such as for load balancing, a special function locks the pair with a defined lock order. The lock order is the lock with the lowest pointer value first. These per-CPU locks and queues resulted in nearly linear scaling with well-behaved workloads in cases where performance previously did not improve with the addition of new CPUs and occasionally have decreased as new CPUs introduced more contention. The design has scaled well from single CPUs to 512-thread network processors.

Adaptive Idle

Many workloads feature frequent interrupts that do little work but need low latency. These workloads are common in low-throughput, high-packet-rate networking. For these workloads, the cost of waking the CPU from a low-power state, possibly with an IPI from another CPU, is excessive. To improve performance, ULE includes a feature that optimistically spins, waiting for load when the CPU has been context switching at a rate exceeding a set frequency. When this frequency lowers or we exceed the adaptive spin count, the CPU is put into a deeper sleep.

Traditional Timeshare Thread Scheduling

The traditional FreeBSD timeshare-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 1 or more seconds and by lowering the priority of threads that accumulate significant amounts of CPU time.

The time quantum is always 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 remained unchanged over the 30-year lifetime of this scheduler. Although the time quantum was originally selected on centralized timesharing systems with many users, it has remained correct for decentralized laptops. While laptop users expect a response time faster than that anticipated by the original timesharing users, the shorter run queues on the single-user laptop made a shorter quantum unnecessary.

4.5 Process Creation

In FreeBSD, new processes are created with the *fork* family of system calls. The *fork* system call creates a complete copy of the parent process. The *rfork* system call creates a new process entry that shares a selected set of resources from its parent rather than making copies of everything. The *vfork* system call differs from *fork* in how the virtual-memory resources are treated; *vfork*