USENIX Association

Proceedings of BSDCon ’03

San Mateo, CA, USA
September 8–12, 2003

ULE: A Modern Scheduler For FreeBSD

Jeff Roberson

The FreeBSD Project
jeff@FreeBSD.org

Abstract

The existing thread scheduler in FreeBSD was well suited towards the computing environment that it was developed in. As the priorities and hardware targets of the project have changed, new features and scheduling properties were required. This paper presents ULE, a scheduler that is designed with modern hardware and requirements in mind. Prior to discussing ULE, the designs of several other schedulers are presented to provide some context for comparison. A simple scheduler profiling tool is also discussed, the results of which provide a basis for making simple comparisons between important aspects of several schedulers.

1 Motivation

The FreeBSD project began a considerable shift in kernel architecture for the 5.0 release. The new architecture is geared towards SMP (Symmetric Multi-Processing) scalability along with low interrupt latency and a real-time-like preemptable kernel. This new architecture presents opportunities and challenges for intelligent management of processor resources in both SMP and UP (Uni-Processor) systems.

The current FreeBSD scheduler has its roots in the 4.3BSD scheduler. It has excellent interactive performance and efficient algorithms with small loads. It does not, however, take full advantage of multiple CPUs. It has no support for processor affinity or binding. It also has no mechanism for distinguishing between CPUs of varying capability, which is important for SMT (Symmetric Multi-Threading).

The core scheduling algorithms are also O(n) with the number of processes in the system. This is undesirable in environments with very large numbers of processes or where deterministic run time is important.

These factors were sufficient motivation to rewrite the core scheduling algorithms. While the new features are important, the basic interactive and 'nice' behavior of the old scheduler were required to be met as closely as possible. In conjunction with this project, the scheduler was abstracted and placed behind a contained API so that compile time selection was feasible.

2 Prior Work

There have been several other notable efforts to make schedulers more scalable and efficient for SMP and UP. These efforts were examined while ULE was under development and many of them had some influence in its design. ULE was intended to be a production quality modern scheduler from the start. As such evolution was preferred over revolution.

Several schedulers are discussed below. The 4BSD scheduler provided refinements over the traditional UNIX scheduler. It was the basis for FreeBSD's current scheduler which ULE used as a target to match for its interactive and nice behavior. UNIX System V release 4 solved the O(n) problem by using a table driven scheduler. This approach was later refined in Solaris[4] and other SVR4 derived operating systems. Linux has also recently received a totally rewritten event driven scheduler from which ULE's queuing system was derived.

2.1 4.3 BSD

The 4.3BSD scheduler is an adaptation of the traditional UNIX scheduler[1]. It provides excellent interactive response on general purpose time sharing systems. It does not, however, support multiple scheduling classes or SMP. This scheduler is discussed in greater detail in [2].

The scheduler derives priorities from a simple estimation of recent CPU usage and the nice setting of the process. The CPU usage estimation, *estcpu*, is calculated as the sum of the number of ticks that have occurred while the process has been running. This value is decayed once a second by a factor that is determined by the current load average of the system. The value is also decayed when processes sleep and wakeup. This decay improves the priority of a process and may allow it to run.

The decay function is the primary reason that the scheduler performs poorly with large numbers of processes. In order for the scheduler to operate on processes which may have been sleeping, it must iterate over every process in the system to decay their *estcpu*. Without this regular decay, processes would not trigger any events that would affect their priority and so they would be indefinitely starved.

Contrary to many newer designs, there are no fixed time slices other than a round robin interval. Every 100ms the scheduler switches between threads of equal priority. This ensures some fairness among these threads.

Nice values have a fixed impact on the priority of a process. Processes with a high negative nice value can prohibit other processes from running at all because the effect of *estcpu* is not great enough to elevate their priority above that of the un-nice process.

Since this scheduler was developed before multiprocessors were common, it has no special support for multiple CPUs. BSD implementations that still use this scheduler simply use a global run queue for all CPUs. This does not provide any CPU affinity for threads. The lack of CPU affinity can actually cause some workloads to complete slower on SMP than they would on UP!

2.2 FreeBSD

FreeBSD inherited the traditional BSD scheduler when it branched off from 4.3BSD. FreeBSD extended the scheduler's functionality, adding scheduling classes and basic SMP support.

Two new classes, real-time and idle, were added early on in FreeBSD. Idle priority threads are only run when there are no time sharing or real-time threads to run. Real-time threads are allowed to run until they block or until a higher priority real-time thread becomes available.

When the SMP project was introduced, an interrupt class was added as well. Interrupt class threads have the same properties as real-time except that their priorities are lower, where lower priorities are given preference in BSD. The classes are simply implemented as subdivisions of the available priority space. The time sharing class is the only subdivision which adjusts priorities based on CPU usage.

Much effort went into tuning the various parameters of the 4BSD scheduler to achieve good interactive performance under heavy load as was required by BSD's primary user base. It was very important that systems remain responsive while being used as a server.

In addition to this, the nice concept was further refined. To facilitate the use of programs that wish to only consume idle CPU slices, processes with a nice setting more than 20 higher than the least nice currently running process will not be permitted to run at all. This allows distributed programs such as SETI or the rc4 cracking project to run without impacting the normal workload of a machine.

2.3 System V Release 4

The SVR4 scheduler provided many improvements over the traditional UNIX scheduler[3]. It added several scheduling classes which include time sharing, real-time, idle and fixed priorities. Each class is supported by a run time selectable module. The entire scheduler was also made order one by a table driven priority system.

The individual classes map their own internal priorities to a global priority. A non-class-specific portion of the scheduler always picks the highest priority thread to run.

Fairness was implemented by penalizing threads for consuming their full time slice without sleeping and applying some priority boost for sleeping without using a full time slice. The current priority of the thread is used as an index into a table that has one entry for each event that will affect the priority of a thread. The entry in the table determines the boost or penalty that will be applied to the priority.

In addition to this the table provides the slice size that a thread will be granted. Higher priorities are granted smaller slices since they are typically reserved for interactive threads. Lower priority threads are likely to be CPU bound and so their efficiency is increased by granting them a larger slice.

2.4 Linux

In January 2002 Linux received a totally rewritten scheduler that was designed with many of the same goals as ULE[5]. It features O(1) algorithms, CPU affinity, per CPU run queues and locks and interactive performance that is said to be on par with their earlier scheduler.

This scheduler uses two priority array queues to achieve fairness. After a task exhausts its slice it is placed on an expired queue. There is an active queue for tasks with slices. The scheduler removes tasks from this queue in priority order. The scheduler switches between the two queues as they are emptied. This prevents high priority threads from starving lower priority threads by forcing them onto the secondary queue after they have run. This mechanism is covered in more detail in the ULE section.

The priority is determined from a simple *estcpu* like counter that is incremented when a task sleeps and decremented when it runs. Slice sizes are dynamic with larger slices given to tasks with higher priorities(numerically lower). This is contrary to many scheduler designs which give smaller slices to higher priority tasks as they are likely to be interactive. Lower priority threads are given larger slices so that negative nice values will positively affect the CPU time given to a thread.

3 The ULE Scheduler

The ULE scheduler was designed to address the growing needs of FreeBSD on SMP/SMT platforms and under heavy workloads. It supports CPU affinity and has constant execution time regardless of the number of threads. In addition to these primary performance related goals, it also is careful to identify interactive tasks and give them the lowest latency response possible.

The core scheduling components include several queues, two CPU load-balancing algorithms, an interactivity scorer, a CPU usage estimator, a slice calculator, and a priority calculator. These components each are discussed in detail in the following sections.

3.1 Queue Mechanisms

Each CPU has a kseq (kse queue) structure, which is named after a component in FreeBSD's new threading architecture. Each kseq contains three arrays of run queues that are indexed by bucketed priorities. Two run queues are used to implement the interrupt, real-time, and time sharing scheduling classes. The last is for the idle class. In addition to these queues the kseq also keeps track of load statistics and the current nice window, which will be discussed in the section on nice calculation.

Since ULE is an event driven scheduler there is no periodic timer that adjusts thread priority to prevent starvation. Fairness is implemented by keeping two queues; current and next. Each thread that is granted a slice is assigned to either the current queue or the next queue. Threads are picked from the current queue in priority order until the current queue is empty. At this time the next and current queues are switched. This guarantees that each thread will be given use of its slice once every two queue switches regardless of priority.

A thread is assigned to a queue until it sleeps, or for the duration of a slice. The base priority, slice size, and interactivity score are recalculated each time a slice expires. The thread is assigned to the current queue if it is interactive or to the next queue otherwise. Inserting interactive tasks onto the current queue and giving them a higher priority results in a very low latency response.

Interrupt and real-time threads are also always inserted onto the current queue, as are threads which have had their priorities raised to interrupt or real-time levels via priority propagation. Without this, a non-interactive thread on the next queue that is holding some important resource, such as the giant lock, could prevent a high priority thread from running.

The idle class has its own separate queue. This queue is checked only when there are no other runnable tasks. Idle tasks are always inserted onto this queue unless they have had their priority raised via priority propagation.

Traditionally in BSD a running thread is not on any run queue. ULE partially preserves this behavior. The running thread is not linked into a run queue, but its load and nice settings are accounted for in the kseq. Accounting for the load is important for kseq load-balancing. It is desirable to distinguish between a kseq that is busy running one thread and one that has no load at all when picking the least loaded kseq in the system.

3.2 Interactivity Scoring

The interactivity scoring mechanism has a substantial affect on the responsiveness of the system. As a result of this, it has the most impact on user experience. In ULE the interactivity of a thread is determined using its voluntary sleep time and run time. Interactive threads typically have high sleep times as they wait for user input. These sleeps are followed by short bursts of CPU activity as they process the user's request.

Voluntary sleep time is used so that the scheduler may more accurately model the intended behavior of the application. If involuntary sleep time was taken into consideration, a non-interactive task could achieve an interactive score simply because it was not permitted to run for a long time due to system load.

The voluntary sleep time is recorded by counting the number of ticks that have passed between a sleep() and wakeup() or while sleeping on a condition variable. The run time is simply the number of ticks while the thread is running.

An interactivity score is computed from the relationship between the sleep time and run time. If the sleep time exceeds the run time, the interactivity score is the ratio of sleep to run time scaled to half the interactive score range. If the run time exceeds the sleep time, the ratio of run time to sleep time is scaled to half of the range and then added to half the range. This equation is illustrated in Figure 1 below.

$$m = \frac{(Maximum\, Interactive\, Score)}{2}$$

$$if(sleep > run)\, score = \frac{m}{(\frac{sleep}{run})}$$

$$else\, score = \frac{m}{(\frac{run}{sleep})} + m$$

Figure 1: Interactivity scoring algorithm

This produces scores in the lower half of the range for threads whose sleep time exceeds their run time. Scores in the upper half of the range are produced for threads whose run time exceeds their sleep time.

These two numbers are not allowed to grow unbounded. When the sum of the two reaches a configurable limit, they are both reduced to a fraction of their values. This preserves their relative sizes while remembering only a fraction of the thread's past behavior. This is important so that a thread which changes from interactive to non-interactive will quickly be discovered. This is actually quite a common case since many processes are forked from shells that have interactive scores.

Keeping too large or small of a history yields poor interactive performance. Many interactive applications exhibit bursty CPU usage. Some applications may perform expensive actions as a result of user input. For example, rendering an image or a web page. If the application's history of waiting for user input was not retained for long enough, it would immediately be marked as non-interactive when it did some expensive processing. If the time spent sleeping was remembered for too long, a previously interactive process would be allowed to abuse the system.

The scheduler uses the interactivity score to determine whether or not a thread should be assigned to the current queue when it becomes runnable. A threshold on the interactivity score is set and threads which score below this threshold are marked as interactive.

This threshold and the amount of history kept are two of the most important factors in keeping the system interactive under load. If the threshold is set too low, expensive interactive applications such as graphical editors, web browsers, and office suites would not be marked as interactive. If it is set too high, compilers, scientific applications, periodic tasks etc. would be marked interactive.

3.3 Priority Calculator

The priority is used in ULE to indicate the order in which threads on the run queue should be selected. It is not used to implement fairness as it is in some other schedulers. Only time sharing threads have calculated priorities; the rest are assigned statically.

A fixed part of the priority range in FreeBSD is used for time sharing threads. ULE uses the interactivity score to derive the priority. After this the nice value is added to the priority, which may actually lower the priority in the case of negative nice values.

This generally gives interactive tasks a chance to run sooner than non-interactive tasks when they are placed on the same queue. It may, however, allow for non-interactive negative nice processes to receive a better priority than an interactive, yet still expensive, process. This is desirable so that nice may have a positive effect on response time as well as the distribution of CPU time.

3.4 Nice Impact / Slice Calculator

One of the more difficult decisions in ULE was how to properly deal with nice. This was difficult because, as was previously discussed, ULE runs every thread at least once per two queue switches and, given certain conditions, some threads should not run at all. The final implementation of nice involves a moving window of nice values that are allowed slices.

ULE keeps track of the number of threads in the kseq with each nice value. It also keeps the current minimum nice value, that is, the least nice process. To be compatible with the 4BSD scheduler ULE only allows threads that have nice values within 20 of the least nice thread to obtain slices. The remaining threads receive a zero slice and are still inserted onto the run queue. When they are selected to be run their slice is reevaluated and then they are placed directly onto the next queue.

The threads that are within the nice window are given a slice value that is inversely proportional to the difference between their nice value and the least nice value currently recorded in the kseq. This gives nicer threads smaller slices which very granularly and deterministically defines the amount of CPU time given to competing processes of varying nice values.

On x86, FreeBSD has a default HZ of 100. This leaves us with a minimum slice value of 10ms. ULE chooses 140ms as the maximum slice value. Larger values are impractical since they would unacceptably increase the latency for non-interactive tasks. This means that the least nice thread will receive 14 times the CPU of the most nice thread that is still granted a slice. Since there are 40 nice values, differences of less than three do not impact the slice size although they do have a minimal affect on priority.

Interactive tasks receive the minimum slice value. This allows us to more quickly discover that an interactive task is no longer interactive. The slice value is meaningless for non-time-sharing threads. Real-time and interrupt threads are allowed to run so long as they are not preempted by a higher priority real-time or interrupt thread. Idle threads are allowed to run as long as there are no other runnable threads in the system.

3.5 CPU Usage Estimator

The CPU usage estimator is used only for ps(1), top(1), and similar tools. It is intended to show roughly how much CPU a thread is currently using. Often summing all of the CPU usage percentages in a system can yield numbers over 100%. This is because the numbers are smoothed over a short period of time.

ULE keeps track of the number of statistics clock ticks that occurred within a sliding window of the thread's execution time. The window grows up to one second past a threshold and then is scaled back down again. This process keeps the ratio of run time to sleep time the same after scaling while making the actual preserved count smaller. This is so that new ticks or sleeps will have a greater impact than old behavior since old behavior accounts for a smaller percent of the total available history.

Keeping ULE O(1) required implementing a CPU usage estimator that operated on an event driven basis. Since a thread may go to sleep for a long time it will have no regular event to keep its CPU usage up to date. To account for this, a hook was added to the scheduler API that is used whenever the CPU usage is read. This hook adjusts the current tick window and tick counts to keep the statistics sane.

There is a rate limiter on this hook to prevent rounding errors from eroding away some of the CPU usage. Before this limit was implemented, top(1) caused the CPU usage to reach zero if it was constantly refreshed.

3.6 SMP

The primary goal of ULE on SMP is to prevent unnecessary CPU migration while making good use of available CPU resources. The notion of trying to schedule threads onto the last CPU that they were run on is commonly called CPU affinity. It is important to balance the cost of migrating a CPU with the cost of leaving a CPU idle.

Modern processors have various large caches that have significant impacts on the performance of threads and processes. CPU affinity is important because a thread may still have data in the caches of the CPU that it last ran on. When a thread migrates to a new CPU, not only does it have to load this data into the cache of the CPU that it is running on but cache lines on the previous processor must be invalidated.

ULE supports two mechanisms for CPU load-balancing. The first is a pull method, where an idle CPU steals a thread from a non-idle CPU. The second is a push method where a periodic task evaluates the current load situation and evens it out. These two mechanisms work in tandem to keep the CPUs evenly balanced with a variety of workloads.

The other situation that ULE takes into consideration is SMT (Symmetric Multi-Threading), or Hyper-Threading as it is called on Intel Pentium4 CPUs. ULE treats SMT as a specific case of NUMA (Non-Uniform Memory Architecture). Based on information provided by the machine dependent code, ULE is able to make scheduling decisions where migrating between different sets of CPUs have different costs and all CPUs are not equal.

3.6.1 Pull CPU Migration

Pull CPU migration is designed to prevent any CPU from idling. It is mostly useful in scenarios where you have light or sporadic load, or in situations where processes are starting and exiting very frequently.

With small numbers of processors it is less expensive to lock the run queue of another processor and check it for runnable threads than it is to idle. Because of this, ULE simply implements CPU migration by checking the queues of other CPUs for runnable threads when a CPU idles.

All of the available kseqs are compared and the highest priority thread is selected from the most loaded kseq. Some alternate algorithms were considered, such as selecting the thread that ran the least recently. Some attempt at balancing the interactive and non-interactive load was also made. Neither of these two approaches showed any measurable gain in any test workloads.

This pull method ends up being effective for short running, high turnover tasks such as batch compiles. It is not quite as good at evenly distributing load for very short lived threads as having a single run queue. The other advantages of the scheduler should outweigh this minor disadvantage.

3.6.2 Push CPU Migration

The pull model for CPU migration is not effective if all CPUs have some work to do but they have an uneven distribution of work. Without push migration, a system with several long running compute-bound tasks could end up with a severe imbalance. For example, it takes just one thread using 100% of the CPU to prevent the pull method from working even if the other processors have many runnable threads.

To prevent this scenario ULE has a timeout that runs twice a second. Each time it runs it picks the two most unbalanced kseqs and migrates some threads to the kseq with the lowest load. If the kseqs are imbalanced by only one thread one thread is still moved from one kseq to the next. This is useful to ensure total fairness among processes.

Consider a two processor system with three compute-bound processes. One thread has an affinity for the first processor while the remaining two threads have an affinity for the second. If we did not periodically move one thread, the thread on the first processor would complete twice as quickly as the threads on the second!

This timeout does not attempt to completely balance all CPUs in the system. There are two advantages to this. The first is that FreeBSD is likely to see more dual processor systems than any other SMP configuration. So the algorithm is simple and perfectly effective for the common case.

On systems with more than two processors it will only take slightly longer to balance the CPUs. This is advantageous because CPU load is very rarely regular. Attempts at over aggressive balancing are likely to ruin caches and not resolve real load imbalances.

3.6.3 SMT Support

Symmetric Multi-Threading presents the scheduler with a slight variation on SMP. Since the logical CPUs in a SMT system share some resources they are not as powerful as another physical CPU. To take full advantage of SMT, the scheduler must be aware of this.

ULE effectively takes advantage of the lack of penalty for migrating threads to a CPU on the same core as well as treating the logical CPUs as less capable than true physical cores. ULE accomplishes this by mapping multiple logical CPUs to the same kseq. This ensures that in a system with multiple SMT capable cores the load-balancing algorithms will prefer to distribute load evenly across groups of CPUs. Since logical CPUs on the same core typically share cache and TLB resources a thread may have run on either without paying any penalty for migration.

A more naive implementation would treat all of the cores as equal. This could lead to many logical cores on the same CPU being used while other physical processors with more resources go unused.

SMT introduces a concept of non-uniform processors into ULE which could be extended to support NUMA. The concept of expressing the penalty for migration through the use of separate queues could be further developed to include a local and global load-balancing policy. At the time of this writing, however, FreeBSD does not support any true NUMA capable machines and so this is left until such time that it does.

4 Late: A Workload Simulation Tool

Late was developed as a means for creating synthetic CPU intensive workloads for use in analyzing scheduler behavior. It collects a variety of metrics that are useful when comparing scheduler implementations.

Late works by running and sleeping over a configurable period. CPU intensive work is simulated by using memcpy to transfer data between two fixed buffers. There is a calibration loop that is run on an idle system prior to any test that determines the number of loops required to occupy the processor for a number of microseconds. The sleep is simply implemented via nanosleep(3).

The time required to execute each work period is averaged out over the course of a run. The maximum time is also kept. The difference between the requested wakeup time from nanosleep(3) and the actual wakeup time is also averaged. These statistics may optionally be reported once a second along with the current priority of the late process. At the end of each run, CPU time, real time, sleep time, and %CPU are displayed. The number of voluntary and involuntary context switches are also displayed when they are provided by the operating system.

Late has the ability to wait for a SIGUSR1 before performing its configured task. This makes setting up

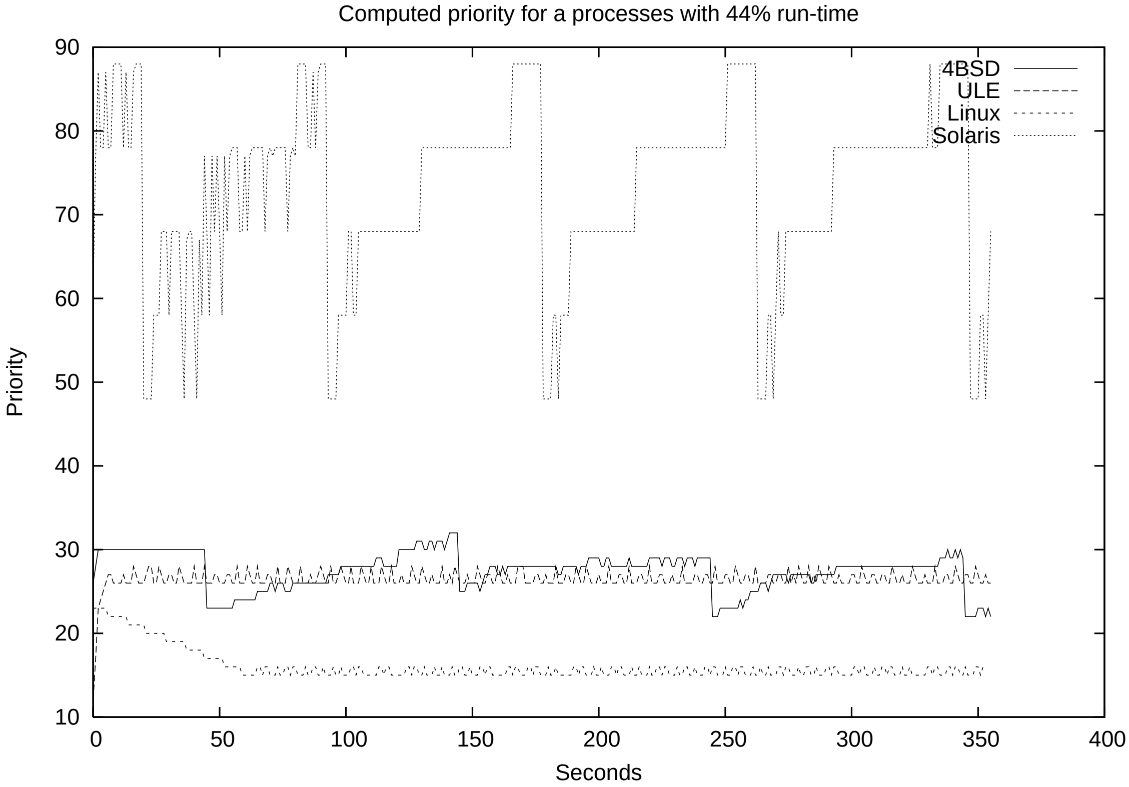

Figure 2: This graph depicts the fluctuations in priority for a process with a constant run-time. The priorities assigned relative to the other schedulers are meaningless. The function of priority and mechanisms of its assignment, however, become evident when it is shown over time.

and coordinating several late processes easy to do from a shell script. Once the late process is permitted to run it may also set its nice value for nice related tests. Setting the nice value from within the late process is a good way to guarantee that all processes were given a chance to proceed up to that point, which they may not have if their priority was too low from the start.

Using these simple facilities late processes can be combined to illustrate various scheduler behaviors by simulating different workloads. While this tool proved to be indispensable during development and for producing benchmarks, it is no substitute for real user experience.

5 Benchmarks

The benchmarks were all created using late to generate synthetic workloads. Combinations of processes simulating compute-bound tasks, batch compiles, vi editors and web browsing were used. A mix of sleep and run time for each simulated application was determined through observing typical runs of these processes on the test system. The benchmarks are intended to give an indication of how a scheduler might perform under load. Due to their artificial nature they are not an absolute representation of real user experience.

The system running the tests was a dual Athlon MP 1900+ with 512mb of memory. Three operating systems were installed; FreeBSD 5.1-CURRENT with the 4BSD and ULE schedulers, Mandrake Linux 9.1 with the 2.5.73 kernel, and Solaris/x86 9.1. The second processor was disabled during each test.

To measure interactivity, all tests, except where noted, ran 4 simulated vi sessions and 2 simulated Mozilla sessions. This mix was chosen to emulate typical use of a server or workstation machine with an administrator or user present in a graphical environment. This should be fairly representative because shells and vi sessions exhibit similar CPU usage characteristics and a graphical administration tool, word processor, or image editor would be represented by the Mozilla simulation. It is important to have expensive interactive applications as well as cheap ones to mimic the use of systems as workstations.

Late keeps track of the time it took for the process to wake up after a timer fired and the time taken to

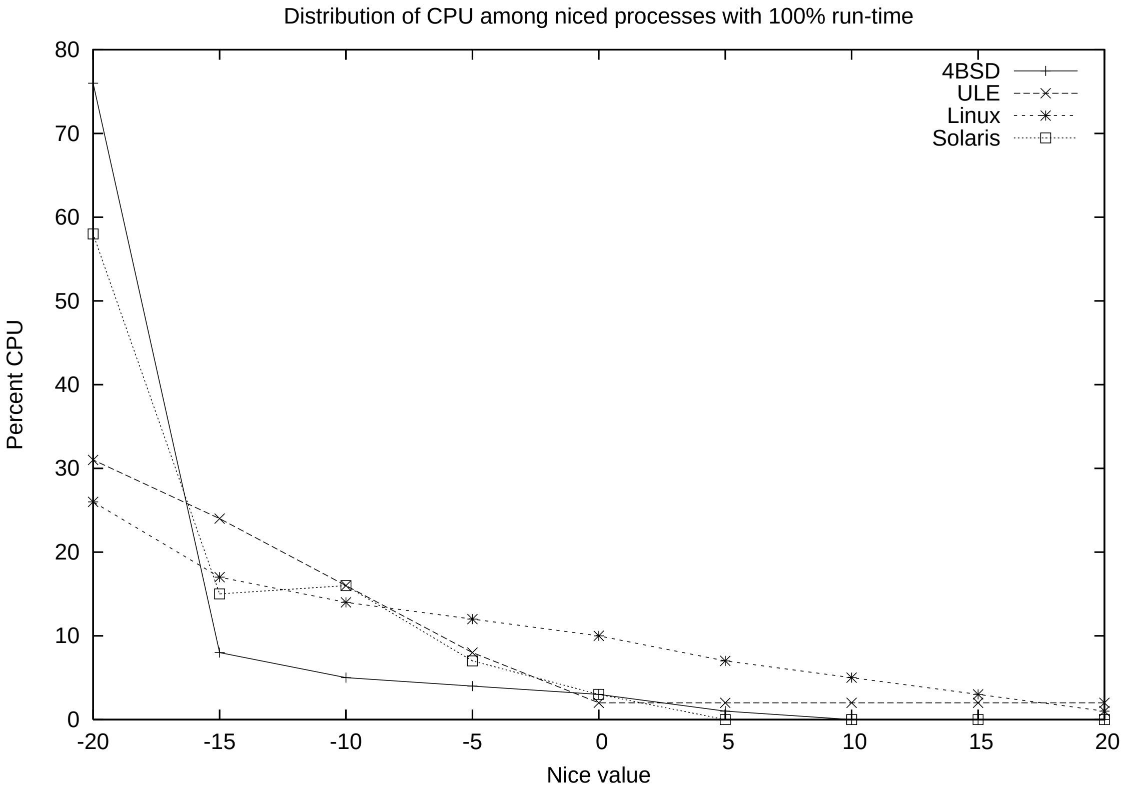

Figure 3: This graph depicts the distribution of CPU granted vs. nice assignment. Two important things are visible here. Firstly, the Solaris and 4BSD schedulers, which always run the highest priority task, give an non-proportional amount of CPU time to nice -20 processes. Secondly, Linux has no cut-off after which processes receive no CPU time.

execute the workload. These two are summed together to indicate the response time for user input in the various interactive tasks.

For the interactivity graphs, the sum of the latency for all simulated interactive processes is graphed. This is not significantly different from the graph of any individual process in most cases. It does, however, widen the gap between the different schedulers in some tests which leads to more easily interpreted results.

5.1 Priority Calculations

In this test a single late process was used. This process runs for 40ms and then sleeps for 50, and thus simulates 44% CPU utilization. Various methods were used to collect the priority of this late process once a second over a 6 minute run. It is important to note that the relative priorities between schedulers are meaningless. We are only concerned with the priority of the process relative to the others in the system and how it is adjusted with CPU usage. This is useful to understand when analyzing the behavior of subsequent tests. The results of this test are shown in figure 2 above.

The periodic decay loop is evident in the cyclic priority assignments of the 4BSD scheduler. This is ensures that even high priority processes will eventually get CPU time by decaying them to a lower priority. Solaris mimics this behavior even without a periodic decay algorithm. It is likely that once the priority reaches a certain value it is reduced significantly to ensure that the process gets some time to execute and thus mimicking the behavior of the 4BSD scheduler.

ULE and Linux do not base their fairness on priority and so they tend to find a stable state. ULE starts at an interactive priority as it inherited it from the shell and then moves up to the priority determined by the CPU utilization of the process. The priority in ULE oscillates around a small range as the process runs and sleeps but it averages out to a level line.

Linux steadily decays the priority until it hits the minimum value. This means that this process, which is using 44% of the CPU, reaches the best user priority possible without a negative nice setting, after approximately 60 seconds. Due to the simple way that Linux tracks CPU utilization, given enough time the

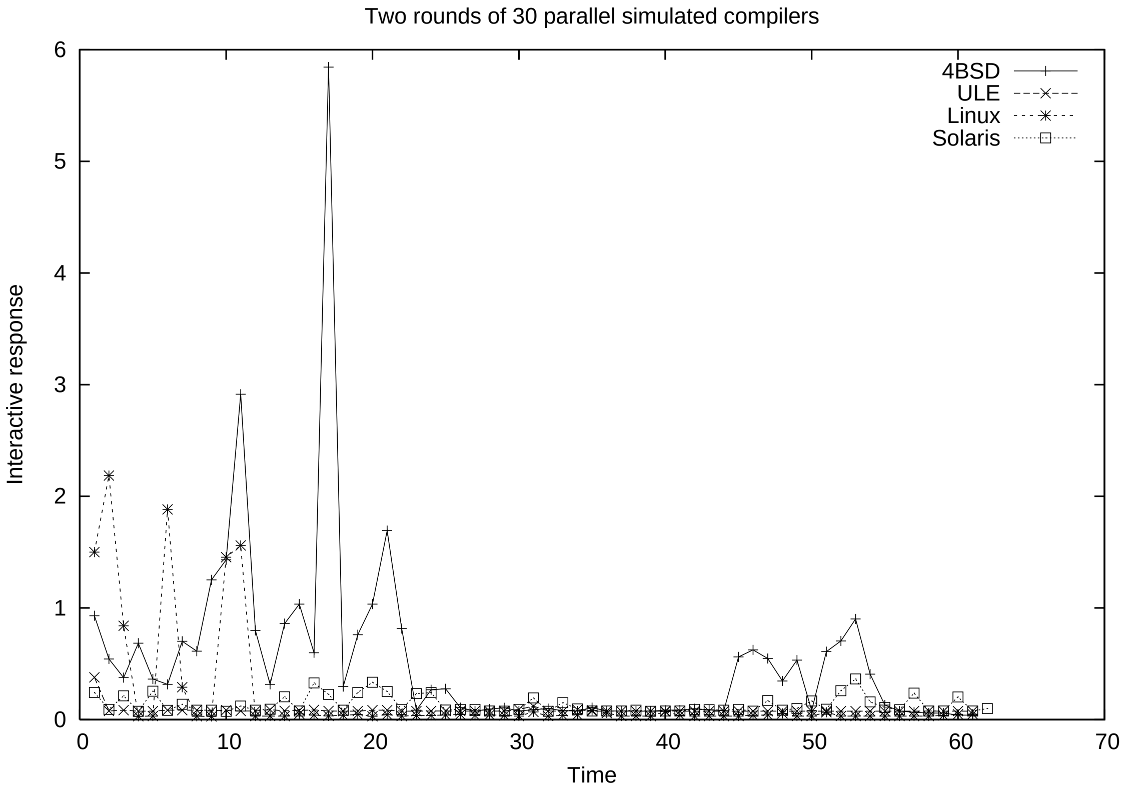

Figure 4: This graph shows the sum of the response time for several interactive applications while the system is running 2 batches of 30 parallel compilers. All of the schedulers do fairly well once the test has been running for some time although they vary significantly in how long they take to stabilize.

priority of a process with a very regular run-time will reach the minimum priority if it runs less than it sleeps and the maximum priority otherwise. This leads to some pathological scenarios which are demonstrated later.

5.2 Effects of Nice

The test in Figure 3 shows the distribution of CPU time among several nice processes running simultaneously. Nice values from 20 to -20 in steps of 5 were chosen to illustrate a wide range of activity on the machine. The results of these tests only illustrate differences in behavior. There is no correct or well defined way to handle nice values.

The processes in this test were never yielding the CPU. This illustrates an important feature of all the schedulers other than Linux. Processes that exceed some nice threshold are not allowed to run at all when there is other load on the system. This was most useful before idle classes were introduced although it is still a common practice to nice a process so that it will take only idle cycles. Curiously, Linux does not have an idle scheduling class either and so this class of applications will always consume CPU time on a Linux machine regardless of load.

Here both of the older schedulers show a clearer bias for -20 processes than the newer two. This is probably more of an artifact of how difficult it is for processes to travel 20 priority points from their CPU usage than it is any design goal. Deriving nice based CPU distribution from the slice size yields a much more even slope in both ULE and Linux.

This data looks somewhat irregular for ULE. The processes that were past the cutoff point were still given slightly less than 1% of the CPU. This is due to an interaction between the simulated workload and ULE's queuing mechanism. The nice late processes were allowed to run for one slice after the un-nice processes exited.

5.3 Interactivity Tests

Figures 4 and 5 illustrate the sum of the response time for several interactive applications. These tests are designed to illustrate how responsive the system remains under several workloads. The synthetic benchmarks do a reasonable job of pointing out what

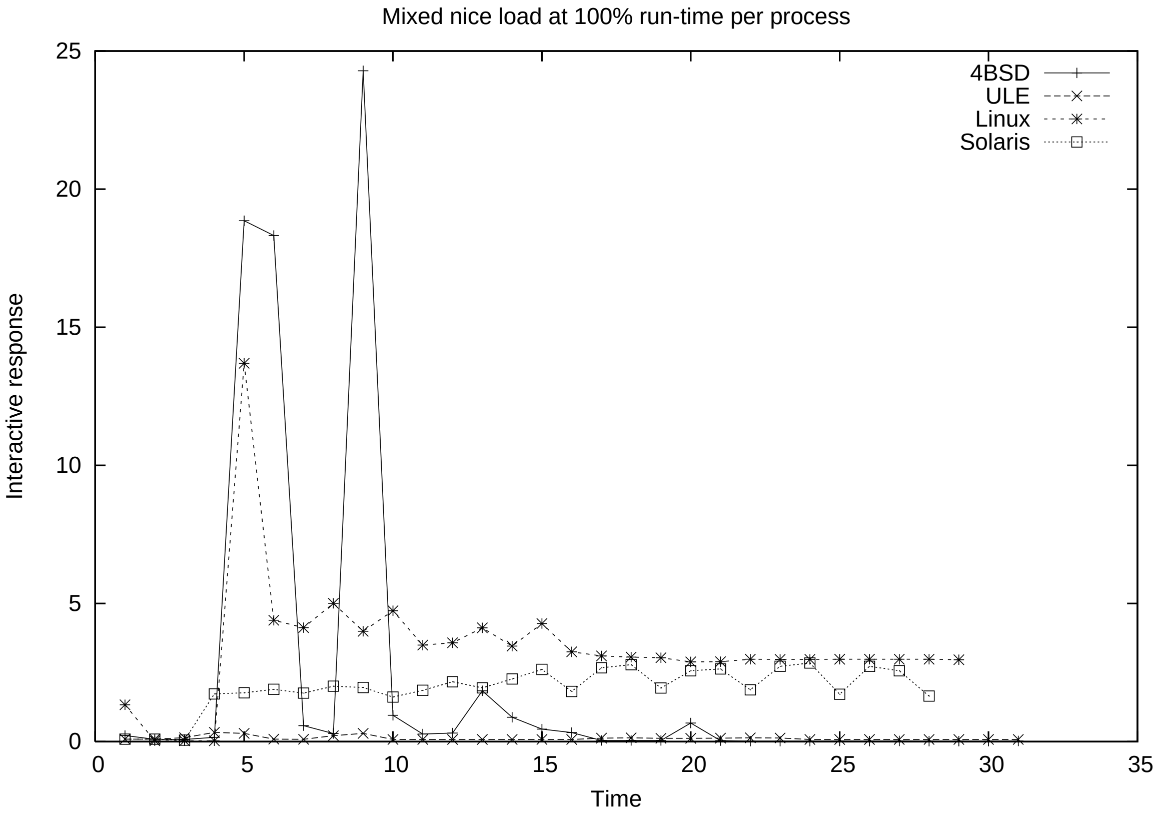

Figure 5: The nice test from Figure 3 was run while measuring interactive latency. All schedulers other than ULE allow a few negative nice processes to have an impact on interactive response.

areas may be problematic. Subjective tests have shown, however, that minor problems exposed by the synthetic tests typically result in much worse interactive behavior than is suggested.

The data for figure 4 was gathered during a test that simulated two sets of 30 parallel compiles. The second set was gated by the completion of all tests in the first set as you would see with a typical build dependency. This illustrates a more realistic workload than the simulations involving nice processes. We can see that all of the schedulers do fairly well once the compilers have been running for a short while. This is due to the time it takes the schedulers to learn the behavior of the compiler processes.

The 4BSD scheduler has the worst spikes due to the high priority inherited from the shell and the time it takes to overcome that. ULE is hardly visible on this graph because it has no spikes in interactive response time and very regularly exhibits almost no latency whatsoever. Solaris has only a few small bursts of latency throughout the test. Linux has a few significant peaks in the beginning and then settles out and performs quite well. Only 4BSD and Solaris show signs of the second batch of compilers starting near 45 seconds. Linux and ULE both distribute more of the learned behavior from child processes to parents when they exit.

Figure 5 was generated from data gathered during the nice related test illustrated above. The nice processes began running 3 seconds after the interactive tasks. In this graph ULE is as responsive as it is under no load. 4BSD is generally responsive after the test is underway but it suffers from several extreme spikes in latency at the begining.

Solaris is the next most responsive after 4BSD. Although an average response time of 2.5 seconds is not acceptable for editor use, it would be enough to use a shell to kill the offending processes. Linux has the worst average latency in this test. This is probably due to the effect of nice on the priority of the process and the time slice granted. The negative nice processes regularly exceed the priority of the interactive processes and starve them for CPU. Even cheap, extremely interactive processes, such as vi and shells, suffer as a result.

All of the schedulers other than ULE exhibit such extreme latency in responding to the interactive tasks

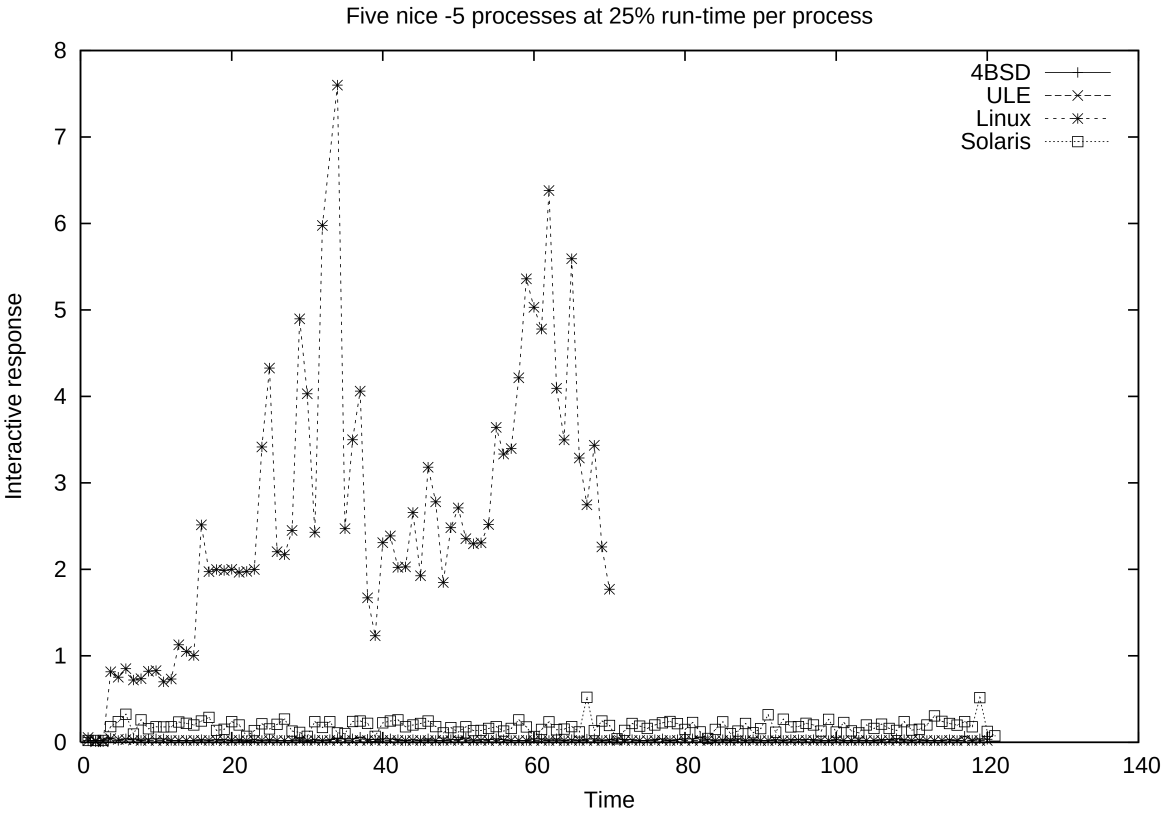

Figure 6: A pathological case in the Linux scheduler is exposed. Priorities for nice -5 processes using 25% of the CPU severely exceed the priority of a simulated vi session.

that they were not able to complete their runs within the time constraints provided. This is why Linux and Solaris seem to have lost several seconds from their graph. ULE stands out as having clearly lower response time and higher regularity than any other scheduler in this test.

Figure 6 illustrates a pathological case for the Linux scheduler which early versions of ULE fell victim to. The setup is 5 nice -5 processes each attempting to use 25% of the CPU. This over-commits the CPU by 25%, which should not be a problem. However, since Linux gradually reduces the priority until it hits the minimum, the nice value is enough to prevent even normal interactive tasks from running with reasonable latency. This was solved in ULE by using the interactivity scoring algorithm presented above.

5.4 Performance

Development of ULE thus far has primarily focused on the interactivity and 'nice' behavior of the scheduler. Now that these algorithms are stabilizing, the focus is shifting to performance tuning. An in-depth analysis of ULE's performance is not within the scope of this paper. However, some initial numbers can be obtained from running the tests performed above with SMP enabled kernels.

The parallel compile test completed four times faster on ULE than it did on 4BSD. This is entirely due to the effects of CPU affinity. This data will not be representative of the results seen with real compilers for several reasons. The synthetic load is entirely memory bound and heavily impacted by the CPU cache. Also, the amount of memory transferred is small and so it is likely that many, if not all, of the late processes can be held in the cache of the two CPUs simultaneously. Late also rarely enters the kernel to do more than a nanosleep(2). Real compilers often contend on kernel resources which reduce the possible parallelization and cause more frequent context switches.

While late demonstrates what is probably close to the best case for CPU affinity gains, early tests with apache may give more realistic results. ULE bests 4BSD by 30% on apache throughput benchmarks on a dual processor Xeon system. This is likely to be more representative of the gains that can be expected from CPU affinity in real applications.

6 Conclusions

While the design of ULE primarily had performance-related goals, it quickly became clear that the architecture provided advantages in other areas as well. Other attempts in the past, such as SVR4/Solaris, have sacrificed interactive behavior for order one execution time. ULE achieves both by borrowing a novel approach from the 2.5 Linux scheduler and the development of new algorithms surrounding that mechanism.

The advantages in interactivity come from several key differences over earlier schedulers. Firstly, interactivity, priority, and slice size are separate concepts in ULE. Other schedulers closely tie these three parameters and are left with many side effects. In ULE each one can be adjusted mostly independently from the rest so that desirable behaviors may be achieved. As a result of this, another one of the key advantages of ULE is possible. Nice is viewed only as a hint by the administrator for non-interactive jobs. As such, the system remains completely responsive despite the nice load. Livelock under nice load has been a constant problem for UNIX schedulers which ULE now avoids entirely.

While the benchmark results are encouraging, a significant amount of time must be spent with ULE in real environments before it will be accepted as the default scheduler by the FreeBSD community.

7 Future Work

ULE was written over a weekend and refined over the course of a year. The 4BSD scheduler, by comparison, has seen a decade of refinement in the FreeBSD project alone. Hopefully, much of that refinement was captured in the new design. Despite the differences in algorithms, ULE should represent another step in the evolution of the UNIX scheduler. Aside from further tuning for a wider variety of workloads, ULE has some areas that are still in need of more analysis and development.

The SMP load-balancing algorithms need to be studied in at least as much detail as the interactivity algorithms have been. They are currently quite primitive, although they may stay that way. Attempts at making them more intelligent have only led to minor improvements in some areas to the extreme detriment of others. This topic is worthy of a paper on its own.

Non-uniform CPU architectures are starting to become popular. Effective support for this in ULE will provide many users with a more compelling reason to switch. At this time enabling Hyper-Threading logical cores leads to worse performance than having them disabled. Recent changes to ULE close that gap but more work is required.

Finally, late has been a very useful tool for scheduler development. However, there are several behaviors of real applications that it fails to capture. As a result of this, its effectiveness in comparing schedulers is reduced. Adding support for variable and bursty workloads as would be seen by real users seems to be the next logical step. After this, giving it multi-threading capabilities would allow us to analyze methods of scheduling multiple threads within the same process.

Acknowledgments

Thank you Dave Anderson, Jagdeep Johal, Mike Karels, Jonathan Mini, Donn Seeley, and Donya Shirzad for providing feedback on this paper. Thanks to Matthew Dillon for his review of the early code and Steve Kargl and FreeBSD-current for their constant testing. Also, thank you Donya for putting up with my long nights writing this paper and hacking on FreeBSD.

Availability

ULE is available via the FreeBSD source repository in any branch after 5.0. Please refer to http://www.FreeBSD.org to obtain the source or an installable FreeBSD image.

Late is available via the FreeBSD source repository in any branch after 5.1. The test scripts are available along with late.

References

[1] M. J. Bach, "*The Design of the UNIX Operating System*", Bell Telephone Laboratories, Inc. (1986)

[2] M. McKusick, K. Bostic, M. Karels, & J. Quarterman, "*The Design and Implementation of the 4.4BSD Operating System*", Addison Wesley Publishing Company (1996)

[3] U. Vahalia, "*UNIX Internals The New frontiers*", Prentice Hall (1996)

[4] J. Mauro, R. McDougall, "*Solaris Internals Core Kernel Architecture*", Sun Microsystems, Inc. (2001)

[5]I. Molnar, Linux O(1) Scheduler design document, http://lxr.linux.no/source/Documentation/sched-design.txt?v=2.5.56