SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:347–364
Published online 24 October 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.772

A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler

Lisa A. Torrey*,†, Joyce Coleman and Barton P. Miller

Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706, U.S.A.

SUMMARY

We implemented a simple multilevel feedback queue scheduler in the Linux 2.6 kernel and compared its response to interactive tasks with that of the new Linux 2.6 scheduler. Our objectives were to evaluate whether Linux 2.6 accomplished its goal of improved interactivity, and to see whether a simpler model could do as well without all of the special cases and exceptions that the new Linux 2.6 scheduler acquired. We describe the two algorithms in detail, report their average interactive response times under different kinds of background workloads, and compare their methods of deciding whether a task is interactive. The MLFQ scheduler performs comparably to the Linux 2.6 scheduler in all response time tests and displays some inadvertent improvements in turnaround time, while avoiding the complex task of explicitly defining interactivity. We maintain an inverse relationship between priority and time slice length, and this seems to be the primary reason that the MLFQ remains simple, yet performs comparably to the Linux 2.6 scheduler. These results may provide some guidelines for designers of new scheduling systems. Copyright © 2006 John Wiley & Sons, Ltd.

Received 27 April 2005; Revised 7 June 2006; Accepted 7 June 2006

KEY WORDS: scheduling; interactivity; Linux; multilevel feedback queue

INTRODUCTION

Linux 2.6 contains a redesigned scheduler, one goal of which was to improve *interactivity*—the responsiveness of latency-sensitive tasks. The resulting algorithm most closely resembles a multilevel feedback queue, but its treatment of feedback is more complicated than in the traditional model [1]. The core algorithm also received several optimizations to handle special cases, which further added to its complexity.

*Correspondence to: Lisa A. Torrey, Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706, U.S.A.
†E-mail: ltorrey@cs.wisc.edu

Interactivity was a driving force in Linux 2.6 because of several acknowledged shortcomings in the previous algorithm. In Linux 2.4 and earlier, the selection of the next process to run scaled up linearly with the number of runnable processes, causing increasingly longer response times in busier systems. Furthermore, the algorithm included a periodic time slice recalculation loop that was also linear in the number of processes. The time quantum given to each process was also too long for high-load systems, and the kernel was not preemptible. Bovet and Cesati [2] and Kolivas [3] provide further information on the motivation to improve interactivity in Linux 2.6.

We are focusing on several questions about this scheduler. First, how well does the algorithm achieve its goal of improved interactivity? Second, could a simpler algorithm do as well? In particular, since it is similar in nature, could a conventionally structured multilevel feedback queue accomplish the same goals without resorting to similarly piecemeal optimizations? We accept the Linux 2.6 scheduler designers' emphasis on interactivity, but if a simpler algorithm would accomplish the same level of interactivity, we would prefer it over the current complex implementation.

THE LINUX 2.6 SCHEDULER

In December 2003, the Linux 2.6 kernel [4] became publicly available. A major feature was the entirely new scheduling algorithm. Contributors identified the following among the main goals of this new scheduler [3,5].

- An O(1) bound on the time to select the next process to run.
- Immediate response to interactive processes, even under considerable load.
- Reasonable prevention of both starving and hogging.
- Symmetric multiprocessing scalability and task affinity.
- Maintenance of good performance in the common case of a small number of processes.

We are concentrating on the scheduler's response to interactive processes, which is directly related to the first three of these goals. Several aspects of the Linux 2.6 kernel impact interactivity (such as the I/O scheduler); in this paper, we concentrate only on the core scheduler. This algorithm is largely centered around the concept of interactivity, as are many of the additional optimizations.

Description of the Linux 2.6 scheduling algorithm

The central data structure of the Linux 2.6 scheduler is the run queue, which contains all of the tasks that are ready to run. It keeps them in two arrays: one *active*, consisting of tasks that are currently eligible to run, and one *expired*, with temporarily ineligible tasks. In general, when a task on the active array uses up its allowed time slice, it waits on the expired array until all of the other active tasks have done the same; at this point the arrays are switched and the expired array becomes active.

The active array holds a list of tasks for each of 140 possible priority levels. The top 100 levels are used only for real-time processes, which we do not address in this study, so this discussion will involve only the last 40 levels. Tasks enter a priority level list at the tail and are taken from the head to be run. When the scheduler chooses the next task to run, it draws from the highest non-empty priority level. Whenever a scheduler clock tick or interrupt occurs, if a higher-priority task has become available,

DOI: 10.1002/spe

it will preempt the running task as long as the running task is not holding any kernel locks. A task's priority level is the sum of its *static priority*, which is based directly on its *nice value*, and its *dynamic priority bonus*, which the scheduler assigns as an estimate of its interactivity. Niceness, the value that the user specifies for the task's priority, has a range of 40 values. Dynamic priority bonuses range from -5 to $+5$, effectively adjusting the user-specified niceness value over one quarter of the potential priority range.

A task runs for a block of time called a time slice. The scheduler assigns time slices based on static priority, giving higher-priority tasks larger time slices and lower-priority tasks shorter time slices. A task might not use all of its time slice at once, since it could block or be preempted before it finishes, but eventually it consumes the entire amount. When that happens, the task is usually placed on the expired array with a new time slice and a recalculated priority.

Several special cases change this process for interactive tasks. An interactive task receives the same time slice as others at the same static priority, but the slice is divided into smaller pieces. When it finishes a piece, the task will round robin with other tasks at the same priority level. This way execution will rotate more frequently among interactive tasks of the same priority, but higher-priority tasks will still run for longer before expiring. Furthermore, an interactive task that finishes its entire time slice still does not expire. It receives a new time slice and remains on the active array, unless the scheduler has detected that the expired array is either starving or holding a task with higher priority than the current task. The expired array is said to be starving when an amount of time proportional to the number of tasks in the run queue has passed since the arrays last switched.

Interactivity is a Boolean classification in the Linux 2.6 scheduler. At each niceness value, a task needs to have a certain dynamic priority bonus to qualify as interactive. It is difficult for low-priority processes to qualify, and difficult for high-priority processes not to qualify. The scheduler calculates a task's dynamic priority bonus by keeping track of a *sleep average*: in general, it adds credit for the time the task spends sleeping and subtracts penalties for the time it spends running (more details follow in the next section).

Complexities and special cases in the Linux 2.6 scheduling algorithm

The Linux 2.6 scheduler has some characteristics of a multilevel feedback queue (MLFQ). Its array of priority lists is a conventional structure, and the concept of awarding resources based on task behavior is a key element of MLFQ scheduling. Using two arrays is an innovative approach to preventing starvation, and is reasonably straightforward. However, the details of the implementation include several complicated mechanisms and special cases to optimize interactive response. The three main issues that we will discuss are the labeling of tasks as interactive, the calculation of sleep average, and the distribution of time slices.

The two-array approach is not ideal for interactivity, since if an interactive task were to expire, there could be a long period of unresponsiveness while the rest of the active array cleared out. The designers addressed this problem by making an exception for interactive tasks, as we mentioned before; they do not expire unless the scheduler has detected starvation and is about to force an array switch anyway. It is difficult to find reasonable situations in which interactive tasks are inactive for long under this policy, but of course it only works if interactive tasks are always labeled correctly.

Since users do not specify whether tasks are interactive or not, the only way to classify a task is to look at its behavior. The developers define as interactive those tasks that sleep for long periods waiting

DOI: 10.1002/spe

for user input and then run for short periods before blocking again. They reduce this behavior to a single number, the sleep average, and use it to calculate the dynamic priority bonus for a task. To allow the Linux 2.6 scheduler to classify a process as interactive or non-interactive, they have set a threshold on the dynamic priority bonus at each static priority level, based on empirical observations of what produced a useful classification.

The basic idea of the sleep average is straightforward: they credit a task for its sleep time and penalize it for its runtime. However, there are several cases that require special treatment to make the interactivity classification effective. For example, since processes can change behavior quickly, it is possible for a task to be treated inappropriately before its label changes to reflect its new behavior. For example, a moderately CPU-intensive task could suddenly become more interactive but take a long time to be reclassified, possibly even expiring, leading to poor interactive response. To make this problem occur less often, the scheduler weights both sleep time and run time so that tasks on the extreme ends of the dynamic priority bonus range lose their current label slowly, while tasks in the middle can shift more quickly between labels. Another case involves tasks that block for disk I/O, which would gain large dynamic bonuses under the general policy, but are not usually interactive with the user. The sleep average of such a task is set at a threshold: high enough to classify as interactive, but low enough to lose interactive status quickly if the task starts to use up its time slices. Other special treatments deal with tasks that are woken by interrupts, which get the time they spend waiting on the run queue credited to their sleep averages because they are likely to be interactive, and child processes, which get their sleep averages decreased to prevent them from hogging in case they are more computationally intensive than their parents. The need for many of these special cases is due to the way that time slices are assigned.

Time slices in a conventional MLFQ scheduler are assigned inversely to priority, so that higher-priority tasks are chosen more often but run for less time. An interesting aspect of the core Linux 2.6 scheduling algorithm is that this relationship is reversed: higher-priority tasks actually get larger time slices. We believe that this reversal causes instabilities in the algorithm, and that many of the special cases had to be developed to compensate for these instabilities.

Each of the exceptions and optimizations seems reasonable, but together they add substantial complexity to the original concept. When the designers encountered all of these issues, should they have concluded that their model was not suitable for their goals? Or are special cases inevitable with a concept as complex as interactivity? The next section presents a well-known model that addresses many of these situations naturally. The following sections evaluate the relative effectiveness of the two models.

THE MLFQ SCHEDULER

An MLFQ [9] has multiple, usually round-robin, queues with a priority and a time slice associated with each queue. The time slice length is inversely related to the priority level. A task moves to a lower priority if it uses up its entire time slice, and runs for a longer interval next time; it moves to a higher priority if it blocks before using its entire time slice, and runs for a shorter interval next time. The scheduler chooses a process from the highest-priority non-empty queue to run, and an arriving higher-priority process preempts a running lower-priority task.

DOI: 10.1002/spe

MLFQ design decisions

We implemented an algorithm that was very simple and conventional, without any special features, and that had parameters similar to those of the Linux 2.6 scheduler whenever possible. For instance, the time slices in the Linux 2.6 scheduler range from 10 to 200 ms, so we used the same range. Increasing time slices by intervals of 10 ms, as the Linux 2.6 scheduler does, gave us a total of 20 queues. MLFQ algorithms often use exponentially increasing time slices, but we decided not to do so because it would force us to change either the time slice range or the number of queues more drastically. We move tasks up and down the queue levels in the traditional manner described above; every time a task becomes ready to run, we check whether it used up its previous time slice and place it either above or below the queue it was in last time. To keep the process completely simple, we did not provide conditions under which a task would remain in the same queue.

Some additional decisions relate to process creation. When one task forks to create a new task, the parent's remaining time slice is split in half between the parent and the child, so that forking divides a task's resources. The child starts on the highest-priority queue, and if it has a time slice too large for that queue we return the extra amount back to the parent. If the child is now on a higher queue than the parent, or if the parent has no time left, the child runs next. Otherwise, the parent continues. Most of this policy is identical to process creation in the Linux 2.6 scheduler, except that there the child always begins on the same queue as the parent. We decided that it would be better to start a CPU-intensive child too high and let it sink quickly to the level its behavior merits, rather than start an interactive child too low and wait for it to rise through the queue levels.

The MLFQ scheduler does not differentiate between tasks based on their nice values, nor does it have a starvation policy. Our tests use only nice 0 user tasks, and we have pointed out any differences that our lack of starvation handling might cause. Since we are testing interactivity, rather than designing a fully functional scheduler for Linux 2.6, niceness and starvation handling were not particularly important issues for our study.

Comparison of the algorithms

One significant difference between the two algorithms is that the MLFQ scheduler makes no attempt to estimate or classify interactivity. It decides what resources to give to a task based on a simple heuristic: whether or not the task used up its most recent time slice. It holds no long-term information about the task's behavior and never decides whether the task is interactive. It is based on the idea that a task's behavior will lead it to a queue that has a suitable time slice, and also that tasks that take up less time before they block should always run first. Another major difference is the opposite relationship that the schedulers give to priority and time slice.

Owing to these differences, the MLFQ scheduler performs well in many important situations without treating them as special cases. Many of the special cases in the Linux 2.6 scheduler are necessary because a task that is labeled as interactive might actually use up its long time slice and therefore hog the CPU, since it is also high priority. This type of hogging is not a danger in MLFQ because the high priority tasks have the smallest time slices, and MLFQ cannot label a task inappropriately since it does not label tasks at all.

There are some potential problems that MLFQ does not address. For example, the Linux 2.6 scheduler limits the sleep averages of tasks that block for disk I/O, because otherwise they would be treated like highly interactive processes, and in most cases they are not. In MLFQ, there is less of

DOI: 10.1002/spe

a problem because we evaluate tasks based only on how long they run and not at all on how long they sleep, so processes doing disk I/O will only look interactive if they also run for very short intervals. However, the general issue remains: MLFQ has no innate way to differentiate between tasks that just run for short periods and tasks that are really interactive with the user. Of course, there is only a need to differentiate between them if user-interactive tasks actually suffer because of this lack of distinction, which is unlikely because of the small time slices they both receive.

The conceptual simplicity of MLFQ is an asset for several reasons. Simple code is, of course, easier to maintain and adapt. Excluding code that is common to both, the Linux 2.6 scheduler takes about 750 lines while MLFQ takes about 450 lines. It also has fewer parameters to set, which means that it requires less experimental tuning; in fact, we did not perform any tuning to achieve the results below. Perhaps most importantly, its generic design makes it more likely to perform reasonably in situations that its designers did not specifically anticipate.

INTERACTIVITY TESTS

We added three simple system calls to the Linux 2.6 kernel that allow us to trace the scheduler's treatment of a single task. These calls provide access into the internal workings of the kernel without significantly changing its behavior.

System calls for scheduler tracing

The first system call we added is `trace_pid`. It takes a process ID as an argument and stores it. The scheduler can then record the time whenever the process with that ID gets placed on the run queue, record it again when the process is actually about to run, and print the intervening time—which is called the *response time* or *scheduling latency*—to the kernel log.

We modified both the Linux 2.6 scheduler and our MLFQ scheduler to perform fine-grained timing using the Pentium `rdtsc` instruction, which reads the value of a clock cycle counter. Recording the time involves a couple of assembly language instructions to read and store the counter value. We execute these instructions in the function `try_to_wake_up`, which is where a task is added back onto the run queue after having slept, blocked, or waited on another queue. In the function `schedule`, where one task passes control of the CPU to another, we read the value again and subtract to find the number of clock cycles that have occurred in the meantime.

The second system call we added is `trace_queue`. It also takes a process ID as an argument and stores it, but instead of making measurements, it prints the contents of the run queue just after the traced process returns to the queue. Furthermore, it logs any scheduling events that occur before the traced process runs.

The last system call is `trace_status`. It also takes a process ID, and it prints information each time that process runs. In both schedulers, it prints the number of the task's current queue. In the Linux 2.6 scheduler, it also prints whether or not the task is currently labeled interactive.

Test descriptions

We performed a variety of tests on both schedulers. Some confirmed properties that we expected both algorithms to have, and some investigated situations that highlight differences between the two schedulers. Several also attempted to measure their performance under realistic workloads.

DOI: 10.1002/spe

The first test we conducted was to evaluate whether the Linux 2.6 scheduler met a basic goal—to respond immediately to interactive processes, regardless of the CPU load—and whether our scheduler did as well. We controlled a background workload of tasks that performed floating-point operations, and increased the number of these tasks while taking response time measurements for the bash shell. To provoke responses, we simply held down a key for several minutes, causing the shell to echo a long string of characters. While not stunningly sophisticated, this method allowed us to collect enough data to examine a characteristic range of response times. We expected that the average time from when the interactive task was placed on the run queue to when it actually ran would remain constant regardless of the background load in both schedulers.

The second test allowed us to examine situations where the interactive process we were timing was not the only interactive task in the system, and also to evaluate scheduling performance during a common and realistic workload. We used compilations of the Linux 2.6 kernel, which perform some CPU-intensive work and also some disk I/O. Again, we were interested in the response times of the bash shell. In fact, we suspected that the Linux 2.6 scheduler might give the user-interactive task better response times than MLFQ because it lowers the priority of tasks that perform disk I/O.

The next two tests addressed some differences in the ways that the algorithms treat interactive processes. In one test, we traced the status of a task that alternates between sleep and bursts of floating-point division over regular intervals, and tried to characterize what it takes to be classified as interactive in the Linux 2.6 scheduler. By varying the proportion of sleep to work, we were able to show roughly how a task's behavior corresponds to its treatment. In the other test, we traced the status of a task that made a sudden behavior change, either from CPU-intensive to interactive or *vice versa*. By recording its progress through queues and classifications, we evaluated how quickly the two schedulers could adapt to processes that behave differently over their lifetimes.

We also performed one test that was unrelated to interactivity. Using a mixture of three types of tasks in batch workloads—CPU-intensive, interactive, and compilation—we measured the total turnaround time of varying combinations. The purpose of this test was to assess, in a general sense, how the two schedulers perform in aspects other than interactivity. We did this to verify that remodeling the scheduling algorithm with a focus on interactivity did not have drastic effects on other important performance characteristics.

UNIPROCESSOR RESULTS

The hardware that we used to perform the tests contained a 425 MHz Pentium III processor with 128 MB of memory. We used a Gentoo Linux distribution with the 2.6.3 kernel. The following results are all based on that system. The next section presents further results on a multiprocessor system.

CPU-intensive workload

Response time, or scheduling latency, is the interval between when a task is placed on the run queue and when it actually runs. Varying the number of CPU-intensive processes in the background had the expected effect on the interactive task's response time: none. As we repeatedly doubled the number of CPU-intensive tasks, average response times remained constant in both schedulers, because the CPU-intensive tasks were placed at a lower priority level than the interactive task.

DOI: 10.1002/spe

Table I. Shell response times with a CPU-intensive workload without daemon activity.

	Number of CPU-intensive tasks							
	1	2	4	8	16	32	64	128
Average response time (Linux 2.6)	4.0 μs	4.0 μs	4.0 μs	4.0 μs	3.9 μs	4.0 μs	4.0 μs	4.1 μs
Average response time (MLFQ)	3.4 μs	3.4 μs	3.4 μs	3.4 μs	3.4 μs	3.4 μs	3.4 μs	3.4 μs
Minimum response time (Linux 2.6)	3.8 μs	3.8 μs	3.8 μs	3.8 μs	3.8 μs	3.8 μs	3.7 μs	3.9 μs
Minimum response time (MLFQ)	3.2 μs	3.2 μs	3.2 μs	3.2 μs	3.2 μs	3.2 μs	3.2 μs	3.2 μs
Data points collected (Linux 2.6)	5703	5705	5703	5714	5714	5714	5714	5709
Data points collected (MLFQ)	5698	5701	5697	5705	5715	5714	5714	5710

Table II. Shell response times with a CPU-intensive workload including daemon activity.

	Number of CPU-intensive tasks							
	1	2	4	8	16	32	64	128
Average response time (Linux 2.6)	22.3 μs	22.1 μs	22.2 μs	22.2 μs	22.2 μs	25.0 μs	22.2 μs	22.2 μs
Average response time (MLFQ)	21.2 μs	21.1 μs	21.2 μs	21.2 μs	21.3 μs	20.4 μs	21.1 μs	21.1 μs
Maximum response time (Linux 2.6)	80.5 μs	23.6 μs	81.8 μs	23.5 μs	23.8 μs	103 μs	85.6 μs	23.5 μs
Maximum response time (MLFQ)	22.1 μs	22.0 μs	22.0 μs	21.6 μs	23.4 μs	21.9 μs	21.5 μs	21.9 μs
Daemon interference frequency (Linux 2.6)	0.4%	0.4%	0.4%	0.4%	0.3%	0.5%	0.3%	0.6%
Daemon interference frequency (MLFQ)	0.4%	0.4%	0.4%	0.4%	0.3%	0.4%	0.3%	0.5%

The average response times were slightly lower by a constant amount for the MLFQ scheduler because it did less work to reschedule a task than the Linux 2.6 scheduler did, but this difference is minor in terms of performance. Under both schedulers, kernel daemons occasionally shared the priority level of the interactive task. Sometimes they would run while the interactive task waited on the run queue, causing a longer response time than usual, although never long enough to be noticeable to the user. This variance often appears to be higher in the Linux 2.6 scheduler than in MLFQ. Tables I and II report measurements with and without daemon activity separately to give a clearer picture of this bimodal distribution.

DOI: 10.1002/spe

Table III. Shell response times with a compilation workload without daemon activity.

	Number of compile tasks	
	1	2
Average response time (Linux 2.6)	5695 μs	6462 μs
Average response time (MLFQ)	1670 μs	1268 μs
Minimum response time (Linux 2.6)	3.9 μs	3.9 μs
Minimum response time (MLFQ)	3.2 μs	3.3 μs
Maximum response time (Linux 2.6)	20 610 μs	28 890 μs
Maximum response time (MLFQ)	9414 μs	10 000 μs
Compile task interference frequency (Linux 2.6)	0.5%	0.7%
Compile task interference frequency (MLFQ)	1.5%	2.4%
Data points collected (Linux 2.6)	6502	7478
Data points collected (MLFQ)	6607	7544

Table IV. Shell response times with a compilation workload including daemon activity.

	Number of compile tasks	
	1	2
Average response time (Linux 2.6)	947 μs	878 μs
Average response time (MLFQ)	277 μs	151 μs
Daemon interference frequency (Linux 2.6)	1.6%	2.0%
Daemon interference frequency (MLFQ)	1.3%	3.7%

Compilation workload

Timing the interactive response with a background of compile tasks produced similar results. The average response time for the Linux 2.6 scheduler remained constant whether there was one compile task or two. Sometimes compile tasks shared the priority level of the interactive task, running while the interactive task waited on the run queue, and then the response time of the interactive task increased by several orders of magnitude. However, these times are all still too small to be noticeable to the user. These data are summarized in Tables III and IV, again with daemon activity separated out for clarity.

The MLFQ scheduler data also displays a constant average response time that is comparable to that of the Linux 2.6 scheduler, with the same small decrease due to smaller scheduling overhead. As expected, it shows more frequent interference from compile tasks than the Linux 2.6 scheduler does. However, when the interference did occur, both the average and maximum resulting response times were smaller than those in the Linux 2.6 scheduler.

DOI: 10.1002/spe

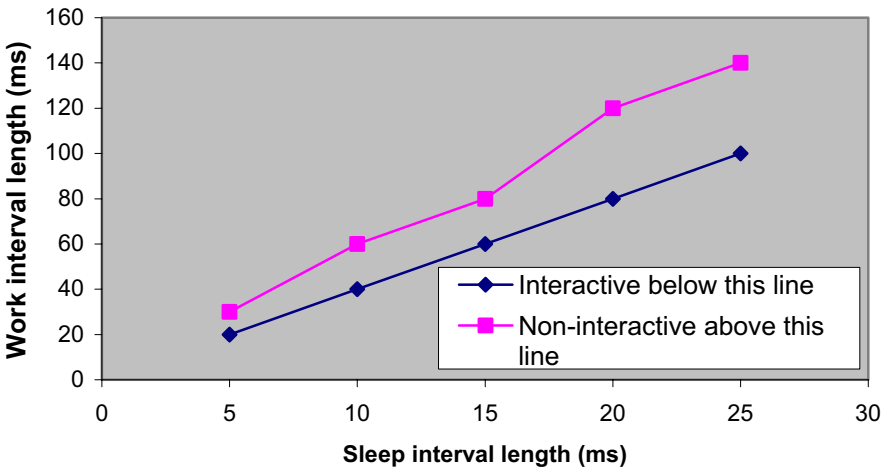

Figure 1. Interactivity by sleep and work intervals in Linux 2.6.

So the Linux 2.6 scheduler appears to keep compile tasks from competing with the maximally interactive shell more often, because it directly limits the interactivity of tasks that perform disk I/O, but when it fails to do so these tasks run for relatively long periods. In contrast, the MLFQ scheduler does nothing to prevent compile tasks from interfering with the shell, but when they do, their run times are more limited. Again, though, both schedulers succeeded in that none of the delays were large enough to be noticeable to the user.

Boundaries of interactivity

We tracked the interactivity classification of a task that alternated between work and sleep, and repeated the test with different lengths of work and sleep intervals. Figure 1 shows how the ratio of work to sleep determines a task's interactivity in the Linux 2.6 scheduler. Below the bottom line the task was labeled interactive, and above the top line it was labeled non-interactive. Between the lines, its label switched back and forth, so it must have been in the center of the interactivity range where a classification change can occur easily.

In the MLFQ scheduler, this test is unnecessary. There is no interactivity classification, so the only way to measure how the scheduler treats a process is to track its queue level over time. Doing so is also unnecessary, though, because MLFQ behaves deterministically based on the length of the work interval. A task that works for less than 10 ms at a time will always be in the top queue, and a task that works for 25 ms at a time will move back and forth between the second and third queue, regardless of how long it sleeps.

We have demonstrated here an ideological difference between the two schedulers' definitions of interactivity. The MLFQ scheduler only takes into account the amount of time a task works, while the Linux 2.6 scheduler also uses the amount of time it sleeps. The data also provide an interesting

DOI: 10.1002/spe

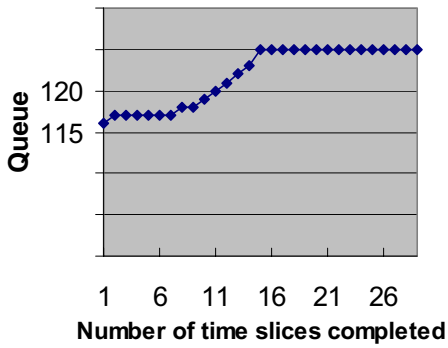

Figure 2. Priority after switching from interactive to non-interactive in Linux 2.6.

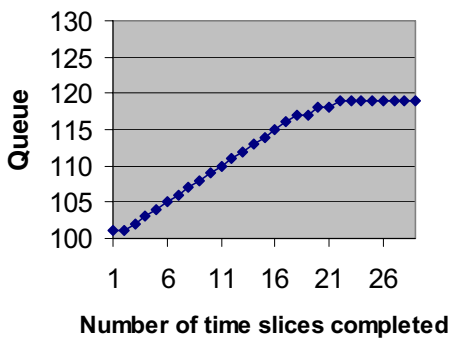

Figure 3. Priority after switching from interactive to non-interactive in MLFQ.

characterization of how work time and sleep time combine into an interactivity classification in Linux 2.6: a task tends to be labeled interactive if its sleep time is at least one quarter of its work time, according to our analysis.

The next figures show how the two schedulers react to sudden behavior changes. In Figures 2 and 3, a task establishes itself as highly interactive by sleeping for several seconds, and then begins to do floating-point operations. In Figures 4 and 5, a task starts by doing floating-point operations and then begins sleeping for 10 ms at a time. Both schedulers have the task in one queue before the change, and afterwards they move it towards a more appropriate level. Note that a lower queue number corresponds to a *higher* priority, and the numbers start at 100 because real-time processes occupy levels 0–99.

Each data point tells what queue the task was on a certain number of time slices after its behavior change. The MLFQ scheduler moves the task across its entire range, but the Linux 2.6 scheduler only allows it to move 10 levels, because the dynamic priority range is -5 to 5. The figures also show how the Linux 2.6 scheduler delays changes in interactivity.

DOI: 10.1002/spe

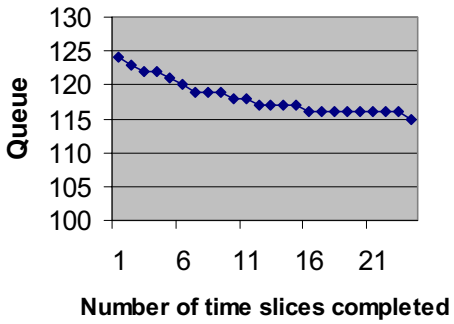

Figure 4. Priority after switching from non-interactive to interactive in Linux 2.6.

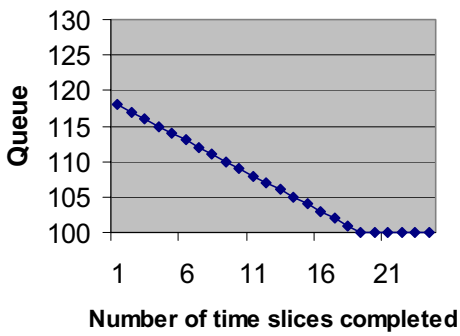

Figure 5. Priority after switching from non-interactive to interactive in MLFQ.

The advantage of the Linux 2.6 approach is that an interactive task that has a temporary CPU burst before returning to its normal behavior does not fall in priority and need to climb back up again. The disadvantage is that while it stays at high priority and uses up its time slice, it may be hogging the CPU, because the time slices at high priorities are long. The advantage to allowing tasks to move across the entire range of queues is that they are not limited to a small range of time slices; the disadvantages is that it ignores the users' nice values. To decide authoritatively which approach is better, we would need to investigate user behavior and weigh the common scenarios that would be affected by these choices.

Side effects

Using the test programs we had already created, we measured the total turnaround times of several workloads. The individual run times of these programs are listed in Table V, averaged over three trials.

DOI: 10.1002/spe

Table V. Test program run times in seconds.

	Program type	
	MLFQ	Linux 2.6
CPU-intensive	8.36	8.41
Interactive	0.41	0.41
Compilation	103.07	101.86

Table VI. Turnaround times of combined workloads.

	Number of CPU-intensive tasks		
	16	32	128
Number of interactive tasks	16	32	128
Number of compilation tasks	1	2	2
Turnaround time (Linux 2.6) (s)	229	468	1283
Turnaround time (MLFQ) (s)	194	412	1189

Table VI reports the turnaround time for combinations of these tasks, each also averaged over three trials. Here there are some differences: the MLFQ scheduler finishes all of these workloads noticeably faster than the Linux 2.6 scheduler does. The smallest test takes 85% of the original time, the second takes 88%, and the largest takes 93%.

Although this performance improvement did not directly relate to interactivity, we investigated possible reasons for it. By eliminating one type of task from the workload at a time, we discovered that the discrepancy occurred only when both CPU-intensive and compilation tasks ran concurrently; the interactive tasks did not contribute to the difference. We then found that the CPU-intensive tasks finished earlier under the Linux 2.6 scheduler than they did under the MLFQ scheduler. There are at least two possible causes: the Linux 2.6 scheduler's starvation prevention mechanism, and its limiting of the priority of tasks that perform disk I/O. We removed the special case in the Linux 2.6 scheduling algorithm that limits the priority of these disk I/O tasks, and found that the turnaround times came closer to those of the MLFQ scheduler, but that the gap did not close altogether. Table VII shows these results for a workload with 16 CPU-intensive tasks and two compile tasks.

The results indicate that part of the reason the MLFQ scheduler had better turnaround times was that it did not artificially limit the compile tasks to lower queues. The rest of the difference may be attributed to the Linux 2.6 scheduler's starvation handling. Since the MLFQ scheduler does not have starvation handling at all, the comparison is not altogether fair to the Linux 2.6 scheduler. If the Linux 2.6 scheduler's starvation limits were relaxed, we would expect its turnaround times to improve. This issue is really about the balance between efficiency and fairness, which is beyond our scope here. We will only point out that the MLFQ scheduler may achieve some performance improvements simply

DOI: 10.1002/spe

Table VII. Turnaround times and task completion (in seconds).

Scheduler tested	Total completion time	CPU-intensive completion time
MLFQ	274.50	223.13
Linux 2.6	355.58	158.81
Linux 2.6 without special case	336.82	182.94

Table VIII. Shell response times (in microseconds) with a CPU-intensive workload on a multiprocessor system.

	Number of CPU-intensive tasks			
	4	16	64	256
Average response time without daemons (Linux 2.6)	7.8 μs	7.8 μs	7.8 μs	7.6 μs
Average response time without daemons (MLFQ)	5.8 μs	5.8 μs	5.8 μs	5.8 μs
Average response time including daemons (Linux 2.6)	11.9 μs	11.9 μs	12.7 μs	13.6 μs
Average response time including daemons (MLFQ)	14.0 μs	19.4 μs	9.0 μs	17.9 μs

because it does not differentiate between categories of tasks. Switching to the simpler model not only maintains good interactivity, but may improve other aspects of scheduling as well.

MULTIPROCESSOR RESULTS

Another goal of the Linux 2.6 scheduler was to improve SMP scalability, so its code contains many changes specific to multiprocessor systems. We ran some of our tests on a multiprocessor system, both to ensure that our changes had not affected the SMP-related features and to compare the two schedulers in another environment. The hardware for this section was a four-processor 200 MHz Pentium Pro system with 512 MB of memory. We used a Red Hat Linux distribution that we upgraded to include the 2.6.3 kernel. The MLFQ scheduler on this system uses the same load-balancing algorithms as Linux.

Response times during CPU-intensive background loads again remained constant, as shown in Table VIII. The effects of daemon interference varied more than they did in the uniprocessor system, but the delays all remained unnoticeable to the user.

In turnaround tests for loads of CPU-intensive and compile tasks, we found again that the CPU-intensive tasks finished later in MLFQ while the entire workload finished earlier. However, when we scaled up the size of the workload, the performance gains of the MLFQ scheduler nearly disappeared. The differences that cause MLFQ to finish workloads more quickly seem to decrease with the workload size. Table IX summarizes these results.

DOI: 10.1002/spe

Table IX. Turnaround times and task completion on a multiprocessor system.

	Number of CPU-intensive tasks	
	16	64
Number of compilation tasks	2	8
Total completion time (Linux 2.6)	148.23 s	463.80 s
Total completion time (MLFQ)	138.59 s	460.04 s
CPU-intensive completion time (Linux 2.6)	79.45 s	320.55 s
CPU-intensive completion time (MLFQ)	86.21 s	349.96 s

RELATED WORK

Scheduler interactivity has been a design issue since the beginning of timeshare scheduling. An early multilevel feedback queue approach that dealt with the topic was CTSS [6], which used priority queues and exponentially increasing time slices in an attempt to provide good interactivity without forcing too many context switches. The designers tried an approach that took user behavior into account: when a user typed a carriage return at a terminal, that terminal's process was promoted to the highest priority. Since the user was interacting with it, they reasoned, that process was likely to be interactive. Of course, on a multi-user system, this policy was widely abused once users discovered it [7].

The FreeBSD operating system, which is based on the Unix kernel, uses a MLFQ scheduler [8,9]. The Solaris operating system [10] also uses one for its time sharing class of processes. Both of these systems maintain the traditional inverse relationship between priorities and time slices. Solaris 2.4 introduced an 'interactive' scheduling class that uses an MLFQ scheduler, but additionally boosts the priority of the task in the active window [11]. This approach takes both user and task behavior into account.

The Windows 2000 operating system [11] contains a scheduler that shares the multilevel queue structure, but like the Linux 2.6 scheduler, it has some interesting differences. Task priorities get boosted only when the task performs certain actions, such as an I/O operation or completing a wait on a kernel object. A boosted task decays back to its base priority one level at a time. There are only two different time slices: one of default length for background processes, and a longer one for the process that owns the active window. Windows XP [12] uses the same type of structure, but schedules threads rather than processes. Threads can be boosted up to two priority levels above their base priority and decay to two levels below, and some threads classified as *real-time* have fixed priorities. The active window is still given a priority boost and a longer time slice.

Duda and Cheriton [13] have proposed *borrowed-virtual-time* scheduling, which is designed to handle a diverse process load containing interactive tasks. It allows real-time and interactive tasks to 'borrow' time from their future CPU allocation to reduce latency, while batch tasks consume their time slices regularly. The authors show that this policy maintains fairness while improving response time. Their approach shares with MLFQ the goal of balancing interactive and batch tasks without incurring a high level of scheduler complexity.

DOI: 10.1002/spe

There has been ongoing work in the Linux community to evaluate the interactivity of the Linux 2.6 scheduler. Molnar posted the original announcement and benchmark evaluations of the new scheduler [14]. White demonstrated improvements in response times from the 2.4 to the 2.6 kernel [15]. Linux developer Con Kolivas developed a *staircase process scheduler* [16], which comes very close to a multi-level feedback queue by eliminating the expired array and moving processes up when they sleep and down when they run. Williams, also designing for Linux, developed a *single priority array* [17] that also eliminates the expired array and replaces the interactivity computations with a simpler analysis of a task's scheduling statistics. These recent developments show that the Linux community has recognized the excessive complexity of the original Linux 2.6 scheduling algorithm.

CONCLUSIONS

Our first objective was to decide whether the Linux 2.6 scheduler succeeded in meeting its interactivity goals. The tests we performed indicate that it did respond well to interactive tasks regardless of the background load. Purely computational loads had no effect on interactivity, and on average, neither did more diverse loads such as compilation. Even the occasional longer delays caused by compile tasks were unnoticeable to a user. Therefore, we conclude that the Linux 2.6 scheduler did achieve its goals regarding interactivity.

Another objective was to decide whether the simpler model of the MLFQ scheduler could succeed as well, handling the important situations without treating them as special cases. Again, the tests we performed indicate that it would. Its average response times were consistently comparable to the Linux 2.6 scheduler's response times, both on a uniprocessor and on a four-processor SMP. Even in the case of compilation, where the MLFQ scheduler had more frequent interference of compile tasks with the interactive shell, the response times remained acceptably small. We could decrease such interference by treating tasks that block on disk I/O specially, as the Linux 2.6 scheduler does, but since we did not find that these tasks caused any substantial neglect of more traditionally interactive tasks, we see no reason not to leave them at their natural queue levels. Finally, our turnaround tests indicate that the MLFQ scheduler may perform favorably to the Linux 2.6 scheduler in ways other than interactivity. So far, our Occam's razor hypothesis seems to have been confirmed.

We would also argue that the MLFQ scheduler has other desirable properties. In particular, we believe that its definition of interactivity is more appropriate. Under the Linux 2.6 scheduler, a process can become more or less interactive depending on how long it sleeps, as well as on how long it runs, while in MLFQ interactivity depends solely on run time. If two processes each do the same amount of work before requesting input, and one receives a response sooner, is either process more or less interactive than the other? If there is any differentiation, then it actually seems more natural to categorize the one that receives a faster response as more interactive, because the faster response could suggest that the user cares about it more. The Linux 2.6 scheduler, however, treats the task that waits longer as more interactive. The MLFQ scheduler treats them equally.

In fact, rather than saying that the definitions of interactivity in the Linux 2.6 scheduler and in the MLFQ scheduler are different, it would be more accurate to say that one of them defines interactivity and one does not. The Linux 2.6 scheduler measures and decides the interactive status of each task, while MLFQ makes simple-minded local decisions with little information. This may be one reason why the Linux 2.6 scheduler requires special cases and complex mechanisms that control specific situations,

DOI: 10.1002/spe

while the MLFQ scheduler does not. The other reason, as we have discussed, is that the Linux 2.6 scheduler needs to compensate for the instabilities created by assigning time slices proportionally to priority.

One issue that we have ignored in this study is starvation, because it was not directly relevant to interactivity. However, the MLFQ algorithm can easily incorporate starvation handling. A natural approach would be to move tasks up to higher levels if they have spent too much time waiting. The Solaris scheduler, for example, increases a task's priority when a one-second starvation timer expires [10]. A task moved in this way would get to run, but only for the short time slice associated with that higher level.

The practical use of this study is the guidance it may provide to developers of new scheduling algorithms. One lesson is that maintaining an inverse relationship between time slice and priority is important for the stability and robustness of a scheduler. Another is that a simple algorithm that avoids defining complex concepts such as interactivity is likely to be more adaptable and more easily tuned than a complicated algorithm.

FUTURE WORK

Along with starvation handling, the MLFQ scheduler could handle niceness values so that users can affect the scheduling of their tasks. One possibility is to limit a task to a certain range of queues based on its niceness. Tasks with increasingly extreme values could be limited to smaller and smaller ranges on the appropriate end of the run queue.

The MLFQ algorithm has several tunable parameters, such as the number of queues, the difference in their time slices, and the distance a task moves when it consumes its time slice or blocks. We were interested in comparing the Linux 2.6 scheduler to an MLFQ scheduler with similar parameter settings, so we did not investigate the effects of changing those settings. It would be informative to do so, though, and especially to compare the performance of MLFQ using exponentially increasing time slices to its performance using linearly increasing time slices.

We also have several suggestions for further investigation of the properties of both schedulers. One is to explore the effects of allowing or preventing quick changes in interactivity, to see which is preferable under common situations. Another is to investigate the effects of starvation handling, to determine which policies are likely to do users the most good. Finally, as new hardware arrives and the Linux scheduler changes, further study on interactivity will continue to be relevant.

REFERENCES

1. Silberschatz A, Galvin PB, Gagne G. *Operating System Concepts* (6th edn). Wiley: New York, 2003.
2. Bovet D, Cesati M. *Understanding the Linux Kernel*. O'Reilly: Sebastopol, CA, 2003.
3. Kolivas C. Interactivity in the Linux 2.6 scheduler, 2003. http://kerneltrap.org/node/view/780 [9 July 2006].
4. Linux 2.6.3 source code. http://www.kernelhq.cc/.
5. Love R. *Linux Kernel Development*. SAMS Publishing: Indianapolis, IN, 2004.
6. Corbató FJ, Daggett M, Daley R. An experimental time-sharing system. *Proceedings of the Spring Joint Computer Conference*, Baltimore, 1962; 335–344.
7. Tanenbaum AS. *Modern Operating Systems* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1992.
8. Lehey G. *The Complete FreeBSD*. Walnut Creek: Concord, CA, 1998.

DOI: 10.1002/spe

9. Ritchie DM, Thompson K. The UNIX time-sharing system. *Communications of the ACM* 1974; **17**(7):365–375.
10. McDougall R, Mauro J. *Solaris Internals*. Sun Microsystems Press/Prentice-Hall: Englewood Cliffs, NJ, 2000.
11. Solomon DA, Russinovich ME. *Inside Microsoft Windows 2000* (3rd edn). Microsoft Press: Redwood, WA, 2000.
12. Russinovich ME, Solomon DA. *Windows Internals* (4th edn). Microsoft Press: Redwood, WA, 2005.
13. Duda K, Cheriton D. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charleston, SC, 1999; 261–276.
14. Molnar I. Ultra-scalable O(1) SMP and UP scheduler, 2002. http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html [9 July 2006].
15. White B. Linux 2.6: A breakthrough for embedded systems, 2003. http://www.linuxdevices.com/articles/AT7751365763.html [9 July 2006].
16. Kolivas C. Linux: Staircase process scheduler, 2004. http://kerneltrap.org/node/2744/7844 [9 July 2006].
17. Williams P. Single Priority Array (SPA) O(1) CPU scheduler, 2004. http://kerneltrap.org/node/3870 [9 July 2006].

DOI: 10.1002/spe