An Evaluation of the Linux Scheduler for Single User Workstations

Russ Ross and Deborah Abel
{rross,dabel}@fas.harvard.edu

January 15, 2000

Abstract

We evaluate the Linux scheduler under a single user workstation setting. Using a modified Linux kernel, we collect trace data for all scheduling decisions and process hierarchy information and then trace particularly problematic system loads to evaluate two items. The first is to see how well the default behavior works, and in so doing we employ a metric for quantifying how successfully a process competes for CPU time. From this information we identify a problem with the propagation of dynamic priorities when a process `fork`s that can cause problems under some circumstances. The second evaluation item is to investigate how effectively `nice` functions as a process priority control mechanism. We find that the default behavior of the Linux scheduler could benefit from hinting about what processes are important, either through X Windows focus information or through some other mechanism. We also identify and explain several situations in which the existing priority mechanism, when used in an intuitive way, gives insufficient control to allow for a highly interactive single user environment which still makes efficient use of idle CPU cycles.

1 Introduction

A large portion of systems research is focused on improving performance for servers and clients on a network with the assumption of widespread time-sharing of important resources. However, with the low cost of a reasonably powerful workstation, another computing model is that of a networked computer in which the bulk of its work done and resources used are on the local machine. In this model, the notion of which processes have a higher priority may be different from that of a time-sharing system. Because only one user is assumed to be interacting with the machine at any given time, all running processes may not deserve equal priority from the point of view of the user. This metric, referred to as "user-perceived performance," measures system performance by how fast a user thinks it is rather than by how fast it performs according to any other set of benchmarks.

This model becomes relevant in the realm of graphical applications, where efficiency is routinely sacrificed for a nicer aesthetic. Interactive jobs can create bursty instances of high CPU demand or even occasional prolonged sessions of intensive work, while still spending the majority of the time waiting for user input. Overall system throughput, average time to completion, and other measures of performance mean very little to an X Windows user whose word processor seems to stick when scrolling because she is running other jobs in the background, or who has to wait an excessively long time for a print preview because of a long compile in the background. The resources available in a typical modern workstation are certainly sufficient to scroll a document smoothly, and from a desktop user's perspective, the operating system should be smart enough to make sure this happens. One should not be forced to underuse the system resources, reserving some fraction of CPU time in anticipation of a burst of activity in the interactive application.

The situation is complicated, however, because the measure of what is important is completely subjective. A user may start a large compile in the background and then surf the internet while waiting for it to complete. In this case, the browser should have ready access to the CPU–it will require a fixed amount of time to complete each user request, so making it wait in order to be fair to the compile job won't even increase total throughput, it will just annoy the user. Even if the compile job suffered slightly to better serve the interactive user's foreground application, the user will usually be willing to take this penalty in order to enjoy the illusion of a machine completely devoted to the task she is waiting on at a given time. On the other hand, the user may have other CPU bound jobs running in the background

and then start a compile and sit, staring at the screen, waiting for the compile to complete. In this case the responsiveness of the compile isn't terribly important, but the user will want it to finish quickly, even if the background job takes a performance hit. When treating all processes with equal status, there is no way for a scheduler to know how to please the user. The only way to tell that a particular compile is the process that should be cheated or favored is to know what the user cares about at a particular time.

Does biasing the scheduler toward some definition of "important" processes improve the responsiveness and performance of the applications the user cares about? What are heuristics that we can use to determine which processes a user thinks is important? For this paper, we assume that in an X Windows system, the process that is running in the currently focused window is the process that the user cares about and that should be responsive or complete quickly. Before this question can be addressed directly, we must first evaluate the existing system, and determine whether or not the scheduler provides adequate control to respond to such hints in an appropriate way.

Our traces have shown that nonresponsiveness and excessive latency in the process a user cares about are real phenomena that can be detected and quantified by measuring each block of time that the process spends being ready to run but not running. In the case of interactive processes, the scheduling events of the X server are also important to the user-perceived performance.

In addition, attempting to fix these latency issues while working within the existing scheduler mechanisms creates new problems. For example, manually setting a video game to the highest priority possible (via `nice`) causes it not to run smoother but to become jerky and unplayable because of interactions with the X server. Therefore, it is not likely that there exists a simple fix of these user-perceived performance problems, and the scheduler itself may require adjustments. Possible fixes involve shortening overall time slices in response to longer run queues; treating X as a special case because it does work on behalf of others; and using hints about focus information to influence priority and time slice size of certain processes.

The rest of the paper is outlined as follows: Section 2 discusses related work in the area of scheduling and user-perceived performance. Section 3 describes our assumptions and background information for our scheduling research, and we describe our research techniques in section 4. Section 5 discusses the results, Section 6 draws conclusions, and Section 7 discusses extensions to this research.

2 Previous Work

Yasuhiro Endo et al. have investigated the use of latency as a measure of interactive performance[1][3]. The authors noted that traditional benchmarks failed to address performance issues from the perspective of a user, instead relying on counters and clock cycles and total throughput as performance measures. These studies make the assumption, which we share, that system throughput is less important in a workstation setting than the latency of certain events. Endo took this metric of latency even further and explored the area of "user-perceived performance"[4]. Here, the authors rely on the user as the starting point for determining which events to study. They built a tool called TIPME which allowed users to indicate when something annoying occurred. This would signal an automated tool that would gather relevant information so that the researchers could deduce the cause of annoying events. These issues often involved unresponsive keyboards and mice.

Much work has been done recently to address the problem that priority-based scheduling does not perform well in real-time situations. In such situations, priority inversion and starvation can occur[5][6][2]. These papers advocate against working within the current scheduling framework to solve the performance problems. Jones et al.[5] propose the concept of CPU reservations, which involves reserving a certain proportion of the CPU for a given process. Steere et al.[6] try to automate this idea by making scheduling decisions based on the progress a task has made toward its goal. Duda and Cheriton[2] propose allowing a process to borrow from its future CPU allowance in order to get rid of latency now. These are all interesting ideas to explore when developing alternate scheduling algorithms to combat the user-perceived performance problem.

3 Assumptions and Background Information

3.1 Desired Behavior/Assumptions

For our experiments, we make several assumptions; we document some of the more important ones here:

- The user is running on a workstation devoted almost exclusively to a single interactive user.
- The user is interacting primarily with a single process at any given time; if that process is blocked, then the user waits for it to finish (or replaces it with a new foreground job).
- The user makes efficient use of the processor. If the primary application uses 50% of the processor time, the user would benefit from using the remaining 50% of the CPU time doing some background task.
- The user desires to have the foreground process run as though it were (almost) the only process running. If a user is running a long job in the background and loads a CPU intensive game to pass the time while the background job completes, he/she has implicitly relegated the background job to a less important status. For our purposes, the foreground game will become the most important thing in the system. If the user doesn't want to disturb the background process too much, he/she should play Minesweeper instead of Quake II to pass the time.
- The user is running under the X Windows environment.

3.2 Problems we don't address

We ignore certain classes of problems in our experiments. We do not consider real-time processes which run in the background (such as an mp3 player). We feel this is justified because the current default scheduler doesn't address them (i.e., we aren't making it worse than it was before), and the solutions that exist now are still valid under our proposed systems. For example, the Linux scheduler allows for real-time priority processes which run FIFO or round-robin with strict priority levels. No normal process will run when a real-time priority process is runnable. Also, these processes do not meet our criteria: they are not interactive, and the user is not working with them directly; background processes with special requirements are beyond the scope of this paper.

3.3 The Linux Scheduler

The Linux scheduler honors the traditional Unix system of `nice` values (set by the user) and dynamic priorities which vary depending on CPU usage. Internally, however, the implementation is different. Each process has two values that are maintained in the `task_struct` associated with that process. The first is called `priority`, and is computed from the user specified `nice` value as `priority = 20 - nice`. So the lowest fixed `priority` is 1 (corresponding to `nice` value 19) and the highest `priority` is 40 (corresponding to `nice` value -20). The second value, called `counter`, represents the length of the time slice that will be assigned to this process when it receives the CPU. Values for `counter` range from 0 (this process has exhausted its time slice) to $2 \times$ `priority` with CPU bound processes having a maximum `counter` of `priority`. Scheduling of processes ready to run is based upon the notion of `goodness`, which is computed as `counter + priority`. The process that is ready to run and has the highest `goodness` value will be scheduled next.

Normally, when a process is scheduled to run it will run until one of three things happens: its time slice runs out, it blocks, or a process with a higher `goodness` value becomes ready to run and preempts the currently running process. When the scheduler cannot find any runnable processes with non-zero `counter` values, it boosts the `counter` for each process in the system (including non-runnable processes) using `counter` $= \frac{1}{2}$`(counter) + priority`. A process that has used up its time slice will be given a new time slice of `priority`, while processes that are not running with receive higher `counter` values that converge toward $2 \times$ `priority`. The default time slice length (for a `counter` value of 20) is 210 ms.

This scheduler has some desirable properties. A process that blocks will accrue a higher `counter` value and thus a higher `goodness` value so it will tend to be more responsive, and it will also receive a longer time slice the next time it runs, thus preserving some fairness based on CPU usage history. When two CPU-bound processes are competing with each other, a process with `nice` value -20 (`priority` value 20) will receive twice as much CPU time as a process with the default `nice` value of 0 (`priority` value 20), so important processes can be made to run more aggressively without monopolizing the system. A process running at the maximum `nice` setting of 19 (`priority` value 1) will receive very little CPU time when competing with a normal process of `nice` value 0, so low priority background tasks can be made to run without detracting too much from normal jobs.

It is important to note that the Linux scheduler makes no attempt to shorten time slices when the system load is high. Therefore, many CPU-bound processes compet-

ing with one another will simply round-robin every 210 ms, with the actual wait time linear in the number of processes running.

For our scenario, however, the scheduler is not ideal. In an interactive setting, low latency on foreground jobs is very important. The Linux scheduler does an excellent job of maintaining fairness over time, but it suffers from latency problems that sometimes behave in counter-intuitive ways.

4 Experimental Setup

We collected data in order to establish two assertions:

1. As it stands, the scheduler does not cater to an interactive user as we have described. If there is idle CPU time left when a foreground job is running, starting a second job to use the remaining CPU time adversely affects the foreground process. If the user wishes to run an interactive foreground process and enjoy the illusion of exclusive use of the machine while still using the idle CPU cycles to do something else, he/she must take explicit action to change the default behavior.
2. The existing mechanism for process priority control (`nice`) does not work in an intuitive way to correct the problem. Setting the `nice` value of processes to indicate which process is important (in the foreground by our definition) does not always give the expected behavior.

4.1 Tracing software

We instrumented the Linux 2.2.13 kernel to trace scheduling events and record when processes become blocked, ready, or running. In addition, we wrote an X application to trace which windows were in focus at a given time. By linking windows to processes in memory, we can analyze scheduling decisions to see how the interactive user fared. While our X traces do not link the processes to windows in real time, they are sufficient to do post-trace analysis and to determine if a real-time X hinting system would be worth implementing.

The second piece of the software involves simply recording important process data such as PID, creation time, and parent information. This gives us a connection between PIDs and executables and also parent/children information for determining which processes are related to each other (e.g. descendants of `make` are all part of the same compile job).

Finally, the focus information gathered from the X server acts as a heuristic for determining which processes are important. It acts based on the assumption that a user will focus his/her attention on the window containing the process that he/she deems to be important.

We work from two traces, the first of which lasted 31 minutes and produced 11,127 process accounting records, 2,763,197 scheduling events, and 104 focus events. For the second trace, we concentrated more on different `nice` values and ignored focus events. This trace lasted 17 minutes and produced 916 process accounting records and 539,558 scheduling events.

4.1.1 Kernel Modifications

We instrumented a single-processor version of the Linux kernel to trace every instance where a process's state changes, primarily between blocked, runnable, and running states. This involved hooks in a few dozen places in the core kernel code, and several hundred places in drivers and related subsystems. The information was stored in a ring buffer in kernel memory and passed to user space via a special device driver that we wrote. For the first trace, we recorded the PID, the previous state, the new state, and a time stamp for each state change. From this information we could infer the length of the ready queue at any given time as well as infer information about when and how long a given process spent in each state. For the second trace we also collected information about process priorities, both static and dynamic.

4.1.2 BSD Process Accounting

The Linux kernel includes support for BSD style process accounting which records information about processes when they terminate. It records information about creation time, the name of the executable if the process did an `exec` call, and several other fields which we didn't use. It lacked some information that we needed, however, so we added that functionality. It didn't record the PIDs of processes, so we included that. In addition, we needed information about the process hierarchy which was not recorded. At first we modified the records to save the PID of the parent process, but we found that this was insufficient. Too many processes became orphans by the time they terminated, and they became the children of `init`, which didn't help us much. We had to add a field

to the main task structure to record the original parent of each process in order to record it with the rest of our trace data.

To finish our traces, we concluded by going to single user mode before ceasing process accounting, terminating most processes so that we had all the information we needed recorded. The primary purpose of the process information was to build a process tree.

4.1.3 X Event Logging

To log X focus events, we wrote a utility to query the X server five times per second to see which window was in focus. Whenever it changed, we recorded a time stamp, the X window ID, and the name of the window. Unfortunately, X does not have information about which process is controlling a given window. X is network transparent, and a PID is not meaningful in a networked environment, but we considered the special case of a single user running everything locally so we needed a way to link PIDs to window IDs. Focus changes are relatively infrequent, and we found that with the augmented BSD process accounting information and the information we got from the X server we could identify which process owned the focused window and associate it with a PID. We set our windowing environment to automatically focus new windows, so most windows were created at about the same time as the process was created and having window titles and creation times was sufficient. Noteably, this scheme does not link PIDs to windows in real time, but it was sufficient for our post-trace analysis, because fewer than 20 unique windows required hand matching to processes.

Actually instrumenting focus event hinting from the X server would require modification of the X server. Most likely, such modifications would take the form of matching processes to windows via the sockets they use or getting processes to transmit their PIDs, which is the approach taken by Endo[4].

4.2 The traces we ran

We began with a 31-minute trace on a machine with an AMD K6-2 300 MHz processor with 128 MB RAM running our trace-enabled Linux 2.2.13 kernel based on RedHat Linux 6.1. This trace contained an intentionally heavy workload to find and amplify latency issues. The three most important process families were a kernel compile, an mp3 encoding, and a session with LyX (a word processor that uses LaTeX as its back end) including a print preview for the built-in user manual (119 pages). The latter involved a large LaTeX job which converted the document to postscript (invoking many Metafont processes to render the text) and then displayed it to the screen. While these intensive tasks ran, we did some internet surfing using Netscape. Interspersed in the trace were also the occasional shell-based applications such as `top`, `ps`, and `ls`.

We also tested the effectiveness of `nice` as a process priority control mechanism. For this test we ran a CPU-bound background process, the X server, and an Atari emulator, trying different combinations of `nice` values for the three applications. The CPU-bound process was the `distributed.net` client which works on cracking an rc5 encryption key, and uses all available CPU time. The emulator emulates the Atari 2600 video game console and requires about 90% of the CPU time to achieve its target rate of 60 frames/second. We modified the trace module to collect `priority` and `counter` information with each scheduling event. This trace information allowed us to analyze how different `nice` values affected scheduling decisions.

5 Results

5.1 Some observations

From informal trials, we observed two main situations where the user wants something to happen and the system is not necessarily cooperating. The first is low latency for interactive tasks, i.e. jumpy scrolling, mouse catching, and menus that don't open up quickly after clicking on them. An entirely different situation arises when the user is doing a CPU-intensive task such as a compile or print preview and he/she keeps that window in focus, willing the event to finish as quickly as possible. Both of these situations can be detected anecdotally with focus events.

Although the situations are quite different, the remedies on the surface are quite similar. Both involve telling the scheduler that the process that is in focus "deserves" the CPU more frequently. However, in the second case a desirable behavior is for the scheduler to also give the process longer time slices when it does get the CPU so that it can finish faster.

5.2 Initial trace

5.2.1 Subjective Results

With the CPU-intensive tasks running, scrolling in Netscape suffered noticeably. In addition, when we first asked LyX to output to postscript for a print preview, it took such a long time without any noticeable results (and no screen redrawing), that we thought the process had locked up, so we simply killed it. We started the application again and tried the same thing, but this time we were more patient. However, Metafont was in great contention with the compile going on, and the compile won. Metafont didn't really get to run until after the compile had completed; at that point it ran at normal speed and produced output in a reasonable amount of time. However, under the user-perceived performance model, if the LyX window was in focus, Metafont should have taken precedence over the compile since we knew the compile would take a long time to finish and just wanted to allow it to finish eventually.

5.2.2 Latency

Our first impulse was to measure suboptimal performance in terms of latency. We defined latency as the amount of time between when a process became ready to run until it actually did run. Figure 1 contains a frequency distribution of number of latency events for a given wait time, with the x axis on a logarithmic scale. Notice that a great majority of these events are acceptably short; from this graph it appears as though processes were able to run as soon as they became ready. Separating latency events by the length of the run queue didn't have much effect either, which was puzzling, because the Linux scheduler has no provision for shortening time slices when the run queue gets long, so we would expect longer latencies in those cases, when in fact, we saw the opposite. It is also possible that our traces didn't collect quite the right information because we traced each time the scheduler changed the state of a process. The kernel often briefly changes a process's state when it acquires a lock or other important resource. These events could have interfered with our data. In addition, we believe that latency may not be the correct measure because of preemption. Even when a process does get the CPU, it can be preempted by another process with a higher priority who has just become unblocked. In general, it appeared as though latency didn't tell us anything useful.

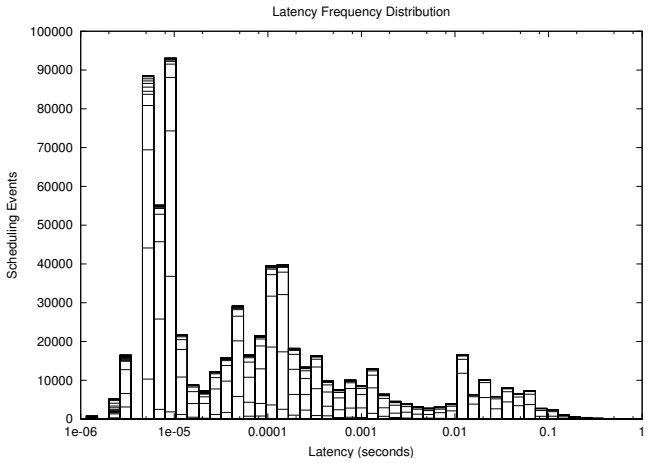

Figure 1: Frequency distribution of latency events. The horizontal bars represent run queue lengths. For example, the space from the bottom to the first bar represents latency events when the run queue was 1. The distance between the first bar and the second bar represents the number of latency events of the specified length when the run queue was 2, etc.

5.2.3 Runtime ratio

Next we created a metric which we called runtime ratio, which is a measure of how well that process competed for the CPU computed over the entire life of the process. A low runtime ratio means that when a process was ready to run, it spent a large percentage of its time waiting for the CPU. A high runtime ratio indicates that when a process was ready to run, it was able to run soon after. We define runtime ratio as

$$\text{runtime ratio} = \frac{\text{CPU time}}{\text{CPU time} + \text{ready time}}$$

Figure 2 contains a frequency distribution of runtime ratio events, with the height of the bars representing the number of processes who had the x axis's particular runtime ratio. However, we soon realized that measuring by the number of processes was misleading; LaTeX spawned off so many processes that it dominated the graph, as seen in Figure 3. When we extracted only LaTeX and related processes, the shape of the frequency distribution looked suspiciously like that of all processes. In addition, the actual lifetime of the process seemed relevant. Does the user notice a problem when a process with only a lifetime of 0.01 seconds spends most of its lifetime waiting? In Figure 4 we show the total amount of time in seconds

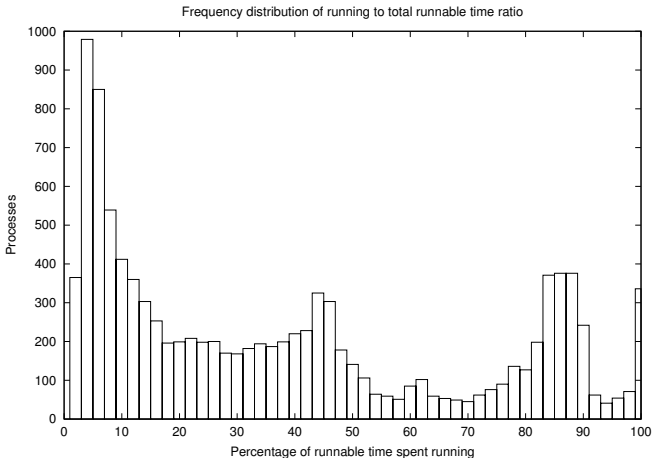

Figure 2: Frequency distribution of runtime ratios, all traced processes

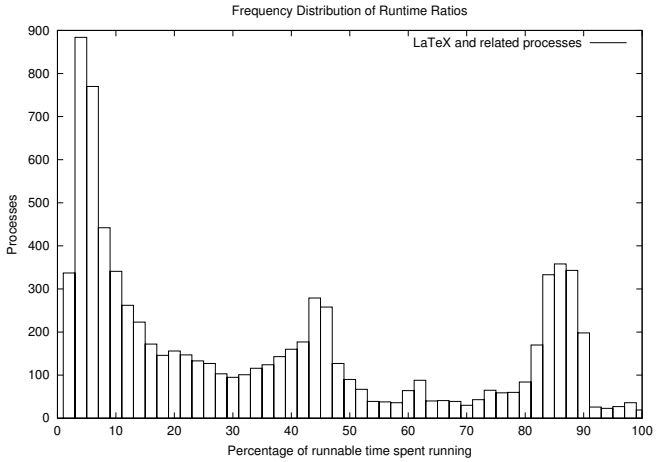

Figure 3: Frequency distribution of runtime ratios, LaTeX and related processes only

that all processes with a given runtime ratio spend either running or being ready to run. The large peak on the left hand side of Figure 2 becomes somewhat shorter, indicating that most of the processes with low runtime ratios were short-lived. However, so many of these processes existed that the total time began to add up, enough to make a difference in the print preview situation. In addition, the right tail of the graph is essentially nonexistent; processes that spent most of their life running were so short-lived as to not make up a noticeable portion of overall running time.

The kernel build and the print preview job seem to be similar in that they both run a series of relatively short, CPU-bound processes. We expected these two process families to exhibit similar behavior, but Figure 5 (a weighted frequency distribution like that of Figure 4) shows that `make` and its related processes did not have the left tail that the LaTeX processes had. The LaTeX job spawned many more processes than `make`, and we were interested to see if the behavior was similar to that described by Endo in BSD systems[4]. He found that the when processes `fork` in BSD Unix, their CPU usage history is not copied, so the low dynamic priority that CPU-bound processes normally acquire is soon forgiven for the child process. In CPU-bound jobs like `make` and LaTeX that `fork` frequently, this translates into an unwarranted priority boost.

In Linux, the only CPU usage history that is kept is the `counter` value, and when a process `fork`s, half of its `counter` goes to the parent process and half to the child. This means that whatever remains of a process's

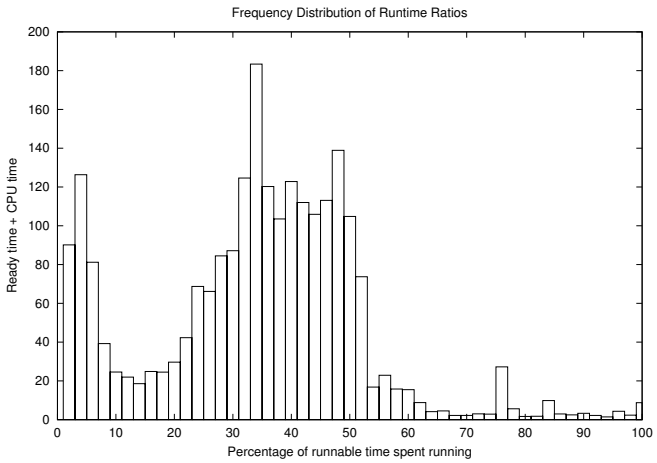

Figure 4: Total time spent ready to run or running for processes with specified runtime ratios

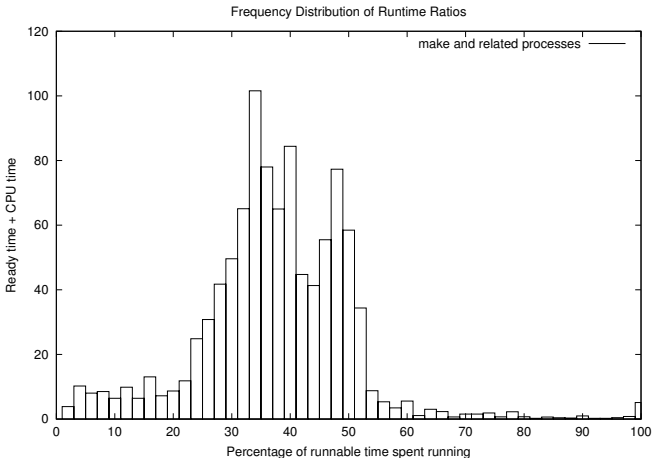

Figure 5: Total time spent ready to run or running for `make` and related processes with specified runtime ratios

time slice when it `fork`s is divided evenly between the original process and the new child process. In a typical scenario, a parent process and its child belong to the same overall job, and by `fork`ing, the parent passes control to the job. The parent process blocks while the child process `exec`s the next phase of the job. When this is the case, `fork`ing can actually give a scheduling penalty to the job as a whole. From the application's standpoint, execution has been transferred from the parent to the child, and half of the current time slice is lost when this happens. The shorter time slice also yields a lower `goodness` value, so the new process typically has to wait longer to be scheduled the first time as well. In the LaTeX job we are examining, the processes were generally very short lived, so added latency at the beginning and a short initial time slice had a very strong negative effect on the runtime ratios of these processes. The effect was cumulative over the life of thousands of short processes, and the end result was an unacceptably long wait for a routine print preview.

5.3 Atari emulator trace

The print preview job was a clear case where the user wanted the foreground job to run more aggressively than it normally would. Even if a mechanism is found for automatically picking out the important processes, the question remains about how to inform the scheduler of this preference in a way that will yield the desired result. The obvious method is to lower the `nice` value of the foreground job, but unfortunately this does not always behave as one might expect.

rc5	X	Atari	Frames/sec	comments
-	0	0	59.9	smooth
-	0	19	59.8	smooth
-	0	-20	59.8	jerky, unplayable
-	-20	0	59.9	smooth
-	-20	-20	59.9	smooth
19	0	0	57.4	smooth
0	0	0	36.0	slow, but smooth
0	0	-20	40.2	jerky, unplayable
0	-20	0	38.3	slow, but smooth
0	-20	-20	40.8	jerky, unplayable*

*not as bad as the (0,0,-20) case

Table 1: Frame rates and empirical observations for different priority combinations of the Atari emulator, X Server, and rc5 cracker

5.3.1 Subjective results

When the emulator and X were running at the same relative `nice` values, the emulator performed smoothly. However, when we `reniced` it down, the game did not run more smoothly as one might expect. To the contrary, motion became jerky, and the game was completely unplayable. Setting X's `nice` value down to what the emulator was running at fixed the problem.

However, things became more complicated when we inserted a CPU-bound process running at default `nice` value (0) into the mix. It garnered enough CPU time to affect the performance of the emulator. Lowering the emulator's `nice` value had the same result as before; however, with the CPU-bound process contending in the background, changing X's priority no longer fixed the problem. The effects of changing priorities seems to run counter to intuition.

5.3.2 Priority control using `nice`

The problems we encountered here are mainly relevant when the foreground job uses enough CPU time that it routinely uses up its time slice. Table 1 gives a summary of the various `nice` values with their frame rates and subjective comments.

When the emulator and the X server were essentially the only processes running on the system, the emulator was able to run at full speed, producing the desired

sixty frames per second. Even when the emulator was given the lowest priority, it still ran smoothly. Examining the trace records for this revealed that X had a `goodness` value which was consistently higher than that of the emulator process. Because X was using relatively little CPU time, it's `counter` value was high, whereas the CPU-bound emulator always had a `counter` less than or equal to its `priority`. Whenever it sent an asynchronous request to X, X would become ready to run and it's `goodness` value would be high enough that X would preempt the emulator and update the screen. This is by design; the scheduler is intended to favor processes that block over those that don't, but the relationship between these two processes was a bit precarious.

When we boosted the emulator's priority to be above the X server's, then the game became jerky and unplayable. Examining the trace results revealed that, although the emulator itself reported that it was rendering nearly sixty frames per second, the X server was only being scheduled one or two times per second, and so nowhere near the 60 rendered frames were being displayed. Groups of frames were either dropped or were being rendered too quickly to see any but the last in the group. Not only did the rendering suffer, but everything else that X did suffered as well, including mouse movements. After moving a single CPU intensive process to a higher priority, the computer became almost unusable. A process with `nice` value of -20 gets a 400 m.s. time slice, and since nothing else had high enough `goodness` to preempt it, every other process suffered a 400 m.s. latency whenever it got scheduled.

We fixed the problem by lowering the X server's `nice` value to match that of the emulator. With nothing else running on the system, the situation was basically the same as when both processes were running at `nice` value 0. When the emulator was running at full frame rate, there was still idle time on the CPU, and using this idle time presents some difficulties. The most effective way under the existing system is to ensure that the background process is running at maximum (19) `nice` value. This way, it gets very short time slices and can be preempted by almost any other process needing to run. Even this is not a perfect solution, however, for reasons both technical and practical.

When running the background job at maximum `nice` value, it still gets to run its time slice in most instances without interruption. Its low `goodness` value ensures that it doesn't run until all other runnable processes have exhausted their `counters` to zero. With one other CPU bound process at `nice` value 0 this happens 4-5 times per second. As Table 1 shows, this doesn't cut significantly into the emulator's frame rate or usability. It does take some time, though, and forces a fixed latency into the foreground process several times per second. There is not way to schedule a process that acts purely as an idle task replacement.

The more serious problem with this for our purposes is that this solution requires the background process to be at low priority. It is not sufficient to boost the priority of the important foreground process and let all other processes move implicitly into the background, as another test illustrated. When we gave X and the emulator `nice` values of -20 and the background task the default `nice` value of 0, the system became very difficult to use. This time, whenever the background process was scheduled (again, only when the emulator had exhausted its time slice) it was allowed to use its entire 210 m.s. time slice without interruption. Scheduling it a single time took enough time that over 10 frames were missed, and it was scheduled one or two times per second (about 400 m.s. for the emulator and 210 m.s. for the background job per time slice). During each of these pauses the emulator was frozen and the mouse was unresponsive.

This is a problematic situation. Our intuition is that the X server should always have a high priority, and our tests seem to confirm this. Beyond that, optimizing the system for a particular process or family of processes cannot be done in the obvious way using the existing scheduler. Giving all the background processes high `nice` values is much more difficult than lowering the `nice` value of a few selected processes. It is not clear what the effect would be of haphazardly increasing the `nice` value of every process that tries to interfere with the foreground job. One could find it suddenly difficult to move a window under X because the window manager is too nice, or a background mp3 player that worked fine at `nice` value zero and didn't interfere too much with the rest of the system might start skipping, etc. The existing Linux scheduler does not adapt time slice duration at all beyond the `nice` value of the process, and as a result it is difficult to have a CPU intensive/latency sensitive foreground job and still use the idle CPU cycles for something else.

6 Conclusion

6.1 Current Limitations

The default Linux scheduler works well most of the time under a relatively light load. When two or more CPU

bound jobs are competing with each other, however, latency issues start to become a problem. This is unfortunate became CPU intensive jobs are natural candidates for background processing, and many of todays user applications require a lot of CPU time either constantly or during certain activities. A workstation user who wishes to enjoy a responsive system that dedicates all needed resources to the job on which the user is currently working must either leave the remaining CPU time idle or take careful and specific action to push other jobs into the background, and even then the results may not be as expected.

Our runtime ratio gives a quantitative measure of how well a given process or family of processes compete for CPU time, and we are able to use this to back up our subjective observations that the foreground processes were receiving too little CPU time. We found a problem in the way dynamic priority is handled when a process `fork`s that is particularly problematic in the Unix paradigm of running many small executables to accomplish a large task. We also showed that `nice`, the existing priority control mechanism, does not give sufficient control to facilitate a simple and intuitive system of boosting priorities for foreground processes.

6.2 Proposed Solutions

One objective in any proposed fixes is to avoid starvation of any processes, and we don't want one process to take control of the system and lock out other processes, which sometimes happens in popular commercial operating systems. Ideally, we'd like to make as few changes as possible to the scheduler to avoid creating new problematic situations while fixing others. We hypothesize that the following changes may help to fix the identified user-perceived performance problems:

- Boost the priority of X. It is a special case process which does work on behalf of essentially all interactive processes. Making its priority helps in many situations, and it doesn't seem to hurt processes because it spends so much time blocking.
- Decrease the size of time slices as run queues get longer. Otherwise, with multiple CPU-bound jobs, the scheduler becomes essentially a round-robin with no preemption, and latency becomes a problem. A CPU-bound interactive process may have to wait for several 200 ms time slices before it gets to run again. Often, concern about context switch overhead is used to justify long time slices. However, context switch overhead is mainly an issue at a much finer granularity, as often happens with two processes that are sharing memory or some other resource that requires synchronization. The extra overhead of breaking a 200 ms time slice into several time slices is not significant.
- Allow for limited hard priorities. For example, one could change the computation of `goodness` such that a process α with `nice` value at least 20 lower than another process β will never have to wait on β. This would allow for some explicit prioritizing. Processes running at the default priority would be unaffected, but those whose `nice` values are explicitly set could compete as they do now with processes whose `nice` values are within a certain range but have a fixed scheduling priority when that is the desired behavior. This would allow for true background processes and more predictable behavior for very high priority processes.
- The Linux scheduler could potentially borrow some ideas from Lottery Scheduling[7]. In particular, unlike some Unix schedulers, there is a clear mechanism for transferring priority between processes. For example, a process that depends on X for on-screen rendering could transfer its `counter` value to X. This would give X a higher `goodness` value so it would have a better chance of preempting other processes, and it would also give it a longer time slice to perform the work.

7 Future Work

This paper identifies various user-perceived performance problems with the Linux scheduler and uses that information to hypothesize possible fixes. Actually implementing these policies and tracing similar events to those discussed here would give a more concrete answer as to whether the proposed scheduling changes would make a difference. In addition, these implementations would need to be tested on other usage patterns to see if they introduce new scheduling problems.

A more difficult problem is to modify the X server so that it can give hints to the scheduler based on window focus information. As mentioned in Section 4.1.3, possible approaches involve creating an X server which matches processes to windows via their shared sockets (this would

require kernel modifications) or getting the the X clients to transmit their PIDs so that the X server actually has that information. Even with such a mechanism in place, further research needs to be done to translate such hints into meaningful information based on that process and information about which of its children is actually running. For example, `xterm` may be the process identified with a particular window, but `gcc` may be the process that needs to be favored. Other considerations come in as well; for example, the child of one window could create an entirely new window, in which case it should no longer be associated with the original window.

More system specific solutions could also prove fruitful; hints generated by a specific window manager may provide more accurate information than can come from the X server alone. We have focused on evaluating the existing system and identifying work that needs to be done before hints can be acted upon effectively. Further investigation of both parts of the problem could yield interesting and useful results.

References

[1] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Michael Smith. The Measured Performance of Personal Computer Operating Systems. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 3-6, 1995. Also in *ACM Transactions on Computer Systems*, January 1996.

[2] Kenneth J. Duda and David R. Cheriton. Borrowed Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general purpose scheduler. *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[3] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using Latency to Evaluate Interactive System Performance. *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, Seattle, WA, October 1996.

[4] Yasuhiro Endo and Margo Seltzer. Improving Interactive System Performance Using TIPME, unpublished, 1999.

[5] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 198-211, October 1997.

[6] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 145-158, March 1999.

[7] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.