Desktop Scheduling: How Can We Know What the User Wants?

Yoav Etsion[*] Dan Tsafrir Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

ABSTRACT

Current desktop operating systems use CPU utilization (or lack thereof) to prioritize processes for scheduling. This was thought to be beneficial for interactive processes, under the assumption that they spend much of their time waiting for user input. This reasoning fails for modern multimedia applications. For example, playing a movie in parallel with a heavy background job usually leads to poor graphical results, as these jobs are indistinguishable in terms of CPU usage. Suggested solutions involve shifting the burden to the user or programmer, which we claim is unsatisfactory; instead, we seek an automatic solution. Our attempts using new metrics based on CPU usage failed. We therefore propose and implement a novel scheme of identifying interactive and multimedia applications by directly quantifying the I/O between an application and the user (keyboard, mouse, and screen activity). Preliminary results indicate that prioritizing processes according to this metric indeed solves the aforementioned problem, demonstrating that operating systems can indeed provide better support for multimedia and interactive applications. Additionally, once user I/O data is available, it opens intriguing new possibilities to system designers.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling; H.1.2 [**User/Machine Systems**]: Human factors; C.4 [**Performance of Systems**]: Design studies

General Terms

Algorithms, Human Factors, Design

Keywords

Human Centered Computing, Scheduling, Multimedia, Interactive, Frame Rate, User I/O

[*]Supported by a Usenix scholastic grant.

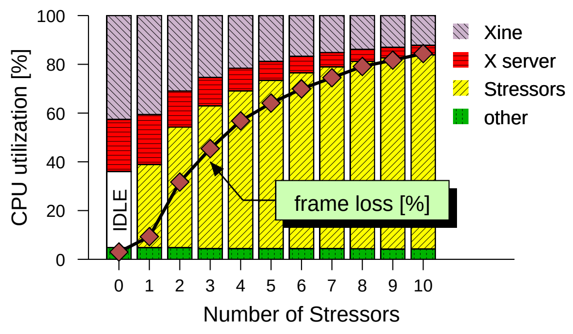

Figure 1: *Background stressors deprive the Xine movie player of required CPU resources, causing increased frame loss rates as more of them are added.*

1. INTRODUCTION

Prevalent commodity operating systems use a simple scheduling scheme that has not changed much in 30 years. Processes are scheduled in priority order, where priority has two main components: static and dynamic. The static component reflects inherent importance differences (e.g. system processes might have higher priority than user processes). The dynamic part depends on CPU usage and ensures that the priority of a process is lowered proportionally to the amount of CPU cycles it has consumed recently.

Tying priority to lack of CPU usage achieves two important goals. The obvious one is fairness: all active processes get a fair share of the CPU. The second one is responsiveness: the priority of a blocked (I/O-bound) process grows with time, so that when it is awakened, it has higher priority than that of other (CPU-bound) processes and is therefore scheduled to run immediately. In fact, in most systems this is the *only* mechanism that provides responsiveness for I/O-bound processes (Windows and Solaris give a boost to interactive processes, but still within the CPU-centric framework). This was sufficient in the past, when user-computer interaction was mainly conducted through text editors, shell consoles, etc. — all applications that exhibit very low CPU consumption. Nowadays, computer workloads (especially on the desktop) contain a significant multimedia component: playing of music and sound effects, displaying video clips and animation, etc. These workloads are not well supported by conventional operating system schedulers [12], as multimedia applications are very demanding in terms of CPU usage and therefore indistinguishable from traditional background (batch) jobs.

Figure 1 is a good example of this deficiency. It demonstrates what happens when a Xine movie-player displays a short clip along with an increasing number of CPU-bound processes (which we call *stressors*) executing in the background. When no such processes are present, Xine gets all the resources it needs (which is about 40% of the CPU). Adding one stressor process is still tolerable since it takes the place of the idle loop. But after that, each additional stressor reduces Xine's relative CPU share, and causes a significant decline in its displayed frame rate. For example, when 4 stressors are present, each gets about 15% of the CPU, and Xine only gets about 20% (half of what it needs), thereby causing the frame rate to drop by a bit more than 50%. A similar effect will occur if the background load is caused by downloading from the net, rather than CPU activity.

In recent years, there has been increasing interest in supporting multimedia applications. Several solutions were proposed to the above problem, which fall into two main categories. The first involves specialized APIs that enable applications to request special treatment, particularly in the area of real-time support, and schedulers that respect these requests [13, 5, 15, 9]. The major drawback of such an approach is that it reduces portability and requires a larger learning and coding effort. The second category implements support for quality of service in the kernel, and allows users to explicitly control the QoS provided to different applications [4, 19, 3]. While this does not require any modifications in the application, it shifts the burden of configuring the system to the user, who must cater for each application individually. This is probably not a good solution for transient interactive and multimedia tasks that come and go during normal work.

Our approach is that the solution should be automatic and transparent. We start by recognizing the fact that some I/O devices can supply us with a fairly good approximation of the user's interests and wishes. We can assume with a reasonable degree of certainty that when the user types on the keyboard, he wants the target application to receive this input and respond to it in a timely manner. Similarly, if some application continuously produces output that spans a significant portion of the screen, it wouldn't be too far fetched to conjecture that the user is interested in this output. By getting such data from the relevant I/O devices, it is possible for the system to identify applications that are of immediate interest to the user, and prioritize them accordingly. In particular, this solves the problem shown in Figure 1; the results, in which Xine is automatically prioritized relative to the stressors and retains its full frame rate are shown in Figure 5.

Importantly, this approach handles both traditional interactive applications (such as text editors) and modern multimedia applications: both types can be identified by tracking user I/O. We collectively denote such applications as being *Human Centered*, or **HuC** for short.

2. METHODOLOGY

Our measurements were conducted on a 664 MHz Pentium 3 machine equipped with 256 MB RAM and a 3DFX Voodoo3 graphics accelerator with 16 MB RAM that supports OpenGL in hardware. The operating system is a 2.4.8 Linux kernel (RedHat 7.0), with the XFree86 4.1 X server. The clock interrupt rate was increased from the default 100Hz to 1,000Hz. This clock rate has been adopted in the newly relesed Linux 2.6 kernel, and is more suitable for multimedia applications which require millisecond timing resolution [13]. We have also verified that the increase in overhead is negligible [6].

As there are numerous different applications in contemporary desktop workloads, we have identified several dominant application classes and chose to focus on a representative or two from each

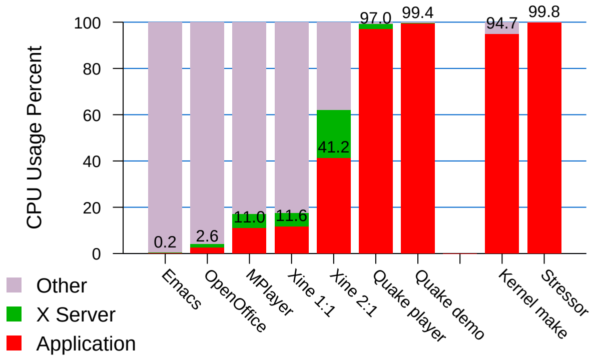

Figure 2: *CPU consumption of different applications (when run alone) expressed as a percentage of the wallclock time.*

class. The representative applications we chose are as follows:

- **Classic interactive applications**: The (traditional) Emacs and the (newer) OpenOffice text editors. During the test, editors were used for standard typing at a rate of about 8 characters per second.
- **Classic batch applications**: Artificial CPU-bound processes (stressors) and a complete compilation of the Linux kernel. Several processes are involved in compilation and therefore the associated data presented in this paper is a summation.
- **Movie players**: MPlayer and the Xine MPEG viewer, which were used to show a short video clip in a loop. While MPlayer is a single threaded application, Xine's implementation is multithreaded, making it a suitable representative of this growing class of applications [8]. In our experiments, audio output was disabled rather than sent to the sound card, to allow focus on interactions with the X server.
- **Modern interactive applications**: The Quake III Arena first-person shooter game. An interesting feature of Quake is that it is adaptive: it can change its frame rate based on how much CPU time it gets. In our experiments, when running alone it is usually ready to run and can use almost all available CPU time.

In addition, the system runs a host of default processes, mostly various daemons. Of these, the most important with regard to interactive processes is obviously the X server (to be described in Section 4.1).

3. IDENTIFYING HUC PROCESSES BASED ON CPU USAGE PATTERNS

Prioritization based on CPU usage can take various forms. In this section we show that all of them do not work, as modern HuC processes may use significant CPU resources, and are essentially indistinguishable from non-HuC work.

3.1 CPU Consumption

The simplest measure of CPU usage is total consumption. Most general purpose schedulers base priority mainly on this metric (see appendix). Processes that use the CPU lose priority, while those that wait in the queue gain priority.

The question, however, is whether low CPU consumption can be used to identify HuC processes. Figure 2 demonstrates that this is

Emacs	Open Office	MPlayer	Xine	Quake user	Quake demo	Kernel make	Stressor
99.6	99.1	98.5	83.1	14.3	1.2	81.6	0.5

Table 1: *Percent of context switches that are voluntary for the various applications.*

not the case. HuC processes are seen to span the full range from very low CPU usage (the Emacs and OpenOffice editors) to very high CPU usage (the Quake role-playing game). Movie players such as Xine provide an especially interesting example: their CPU usage is proportional to the viewing scale. Showing a relatively small movie, taking about 13% of the screen space, required about 15% of the CPU resources for the player and X combined. Using a zoom factor of 2:1, the viewing size quadrupled to about half the screen, and the resource usage also quadrupled to about 60%. Attempting to view the movie on the full screen would overwhelm the CPU. This is despite using an optimization by which the frame data is handed over to X using shared memory.

3.2 Effective Quantum Lengths

While CPU consumption is the main metric used by current schedulers, other (new) metrics are also possible. A promising candidate is the distribution of *effective quantum lengths*. An effective quantum is defined to be the period from when a process is allocated a processor until the processor is relinquished, either because the process has exhausted its allocated quantum, or because it blocks waiting for some event, or because a newly awakened process has a higher priority. The intuition is that although HuC processes may exhibit large CPU consumption, their effective quanta probably remain very small due to their close interaction with I/O devices. Thus we expect to see a difference between the allocated quanta and the effective ones in HuC processes, but expect non-HuC processes to typically use their full allocation.

Figure 3 shows these distributions for different groups of applications. Multimedia applications, in particular, are indistinguishable from other application types: on one hand Quake behaves just like a CPU stressor, both when running alone and when running with a competing process, and on the other hand Xine resembles the well-known kernel-make benchmark.

3.3 Voluntary vs. Forced Context Switches

Another possible metric is the *type* of context switch. HuC processes (such as movie players) often relinquish the processor voluntarily, due to their dependency on I/O devices, through which they communicate with the user. We can therefore classify processes according to the *fraction* of their effective quanta that ended voluntarily, rather than the *duration* of the effective quanta (as described above).

We define a voluntary context switch as one that was induced by the process itself, either explicitly by blocking on a device, or implicitly by performing an action that triggered another process to run (such as releasing a semaphore). We were able to trace such context switches by monitoring the various kernel queues. The results shown in Table 1 indicate that this new metric also fails to make a clear distinction between HuC and other processes. Quake is again similar to stressors, and Xine looks like kernel-make.

4. IDENTIFYING HUC PROCESSES BASED ON USER INTERACTION

Failing to identify HuC processes using the traditional CPU-consumption-based metrics suggests a different approach is needed. It seems there's no alternative other than to actually follow the flow of information between the user and the various processes, and explicitly characterize HuC processes as such according to the magnitude of this flow.

4.1 HuC Devices

Only a subset of the peripheral devices in the system are of interest when trying to quantify the volume of interactions between the user and the various processes. These include the keyboard, mouse, screen, joystick, sound card, etc., and will be referred to collectively as *HuC devices*. The common property of such devices is that they all directly interact with the user. For the purpose of this research we've decided to only monitor the "bare necessities", namely the keyboard, mouse, and screen. The same principles can be applied to the other HuC devices in a straightforward manner.

Unix environments use the X-Windows system [20] to multiplex I/O between the user (as represented by HuC devices) and the various applications. Applications that wish to communicate with HuC devices are referred to as *X-clients*. Clients connect to the *X-server* and communicate with it using the *X-protocol*. The server usually associates a window with each client, such that user input events performed within this window are forwarded to the client (in the form of *X-events*), and output produced by the client (in the form of *X-requests*) is directed to this window. Consequently, the X server centralizes all work concerning the kernel mechanisms that allow communication with the canonical HuC devices, and hence with the user. It is therefore natural to use the X server as a meta-device when monitoring user I/O. We have instrumented the X server to collect per-process I/O data, and forward it to the kernel once a second. Also, since the X server is itself a process, its own user I/O must be reported. As such, the kernel sums the I/O the X server reports about other clients, and considers that sum to be the X server's own I/O production.

We remark that even though the X protocol is the conventional paradigm used to perform user I/O in Unix environments, other mechanisms do exist. The *Direct rendering Infrastructure* (DRI [14]) is an extension to the X protocol that allows direct interaction with the graphics controller. This is the dominant alternative to using the X protocol itself for output since it is used by the OpenGL graphical library [17], which in turn is heavily used by graphical software (such as Quake). In order to make our implementation complete, OpenGL should have also been modified (similarly to the X server) to maintain the per-process output statistics and to periodically report them to the kernel.

4.2 Quantifying User Input

Input events can be perceived as an immediate and explicit expression of the user's wishes. The number of events is typically not so important: dragging with the mouse, which generates multiple events per second, conveys the same amount of user interest as a single mouse button click or the typing of a single character. But given that users are slow, the effect is retained for several seconds.

The most important issue regarding input is recency: the most recent user input should get the highest priority. Therefore, in addition to the regular periodic updates sent from X to the kernel once a second, whenever a client that has not received any input events recently, does receive input, the kernel is immediately notified. This allows the scheduler to maximize responsiveness by promptly handling such events.

4.3 Quantifying Output to the User

Unlike user input that has almost an unary nature (a human user can deliver simultaneous events to very few processes in one sec-

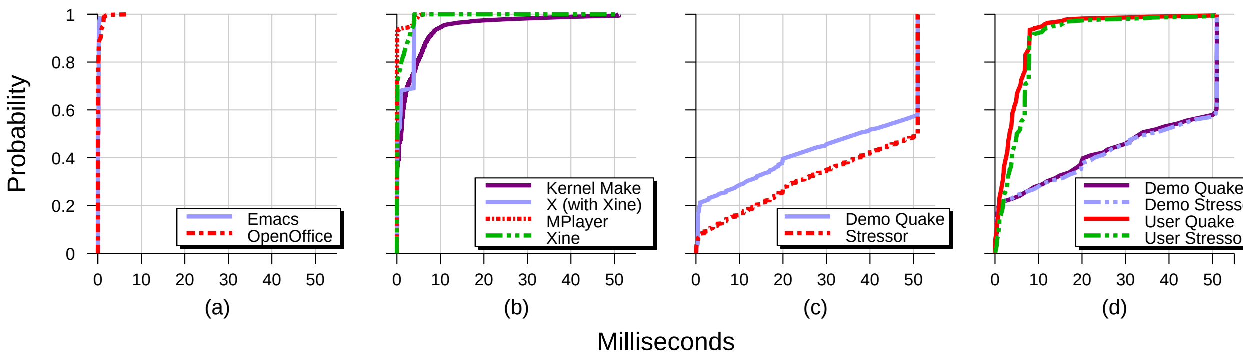

Figure 3: *Cumulative distribution function of the effective quanta when applications are run alone.* **(a)** *Editors have very short effective quanta.* **(b)** *Movie players also have short effective quanta, but this is similar to the profi le of the kernel-make batch job.* **(c)** *Quake can consume all available CPU cycles, so when running in demo mode it behaves like a stressor. Both are occasionally interrupted by various system daemons, causing around 50% of the effective quanta to end prematurely.* **(d)** *When a stressor runs together with Quake , both end up with the same distribution, because Quake interrupts the stressor.*

Request Name	Description
ClearArea	Clear a rectangular area
CopyArea	Copy a rectangular area
CopyPlane	Copy one color plane of a rectangular area
PolyPoint	Draw points
PolyLine	Draw a line through a path of points
PolySegment	Draw multiple separate lines
PolyRectangle	Draw the outline of rectangles
PolyArc	Draw a circular or elliptical arc
FillPoly	Fill the region inside the specified path
PolyFillRectangle	Fill a rectangle
PolyFillArc	Fill a given arc (either Chord or PieSlice)
PutImage	Draw a bitmap
PolyText8	Draw a 1-byte character string
PolyText16	Draw a 2-byte character string
ImageText8	1-byte characters string with background
ImageText16	2-byte characters string with background

Table 2: *X protocol graphical requests*

ond) and which reflects the immediate wishes of the user, quantifying output is a bit more complex: firstly, because various applications may simultaneously produce output to different windows, but more importantly, because we don't know which of these output events is more significant to the user. To cope with this difficulty we exploit a feature in human perception and physiology that is a remnant of our predatory days: human vision is more sensitive to movement [16]. By quantifying the rate of changes produced by each client we get a reasonable idea about which process has the user's attention. We further assume that the user does not like to be distracted and will eliminate any source of interference (e.g. iconify an irrelevant window).

The question remains of how to quantify the rate of screen changes. Simple event counting will not work in this case since an output event can be as small as printing a character or as large as changing the background image. Moreover, output events may refer to hidden portions of windows. Our goal is therefore to approximate the percentage of the screen actually changed due to each output event.

This is a feasible task since the X protocol defines a reduced set of only seventeen graphical X-requests that are available to clients.

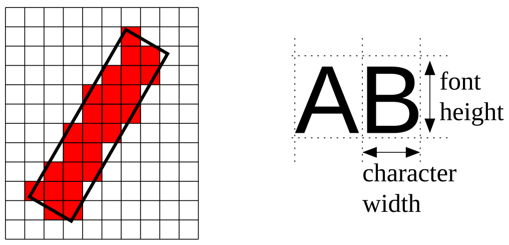

Figure 4: *Estimation of the area of a diagonal line and a text caracter.*

The full list of graphical requests is available in table 2. For each X-request we have implemented a function that approximates the amount of change it introduces to the screen. Additionally, we've hooked to the X clipping mechanism in order to find out how much of the change is indeed visible to the user.

For example, when drawing a diagonal line, the number of pixels drawn is estimated by the line's $width \cdot height$. This number however, is not accurate since the line's boundaries might not coincide with the discrete pixels' boundaries as can be seen in Figure 4. Although in this case the inaccuracy is in the range of 1-2 pixels only, it might be bigger for other lines. The figure also shows the calculation of the area used by a character. When drawing a string the estimate of the area drawn is the sum of the bounding boxes of all the characters used, whereas the actual area used is smaller.

5. EXPERIMENTAL RESULTS

To evaluate the concept of HuC scheduling and our Linux implementation of this concept (described in detail in [7]) we conducted measurements with several workloads. The workloads typically included at least one HuC process, and different numbers of stressor processes that compete for the CPU.

Probably the most striking result is shown in Figure 5. This shows profiles of executing Xine showing a movie at a 2:1 size ratio, with up to 10 stressor processes. Xine and the X server require about 60% of the CPU in this case. Under the original Linux scheduler, they do not get this percentage when there are two or more stressors, resulting in an increasing frame-loss rate as stressors are added (Figure 1). But with the HuC scheduler Xine and X are identified as the focus of attention and given priority over the

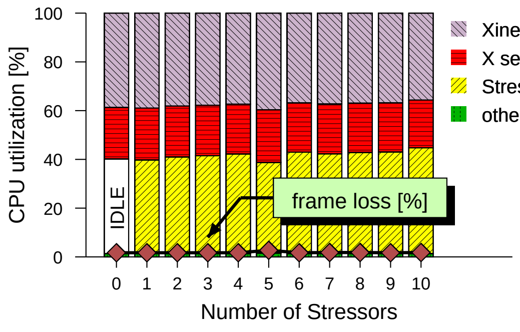

Figure 5: *Division of CPU time and Xine's frame-loss rate under the HuC-scheduler (compare with Figure 1).*

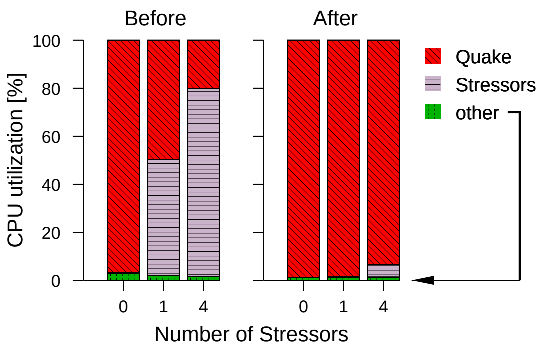

Figure 6: *CPU share given to quake by the default scheduler (left) and by the HuC scheduler (right).*

stressor processes, and they continue to get 60% of the CPU regardless of the number of stressors. As a result the frame loss rate remains negligible.

Similar results are obtained for other applications as well. At the low end of CPU usage, applications like the Emacs editor are unaffected by the HuC scheduler. Emacs only requires maybe 1% of the CPU resources, and gets it even under the default scheduler; the HuC scheduler provides the same. But at the high end, Quake can adaptively use CPU resources to improve its output quality. When run under the default scheduler, its share of the CPU is reduced when stressor processes are added. With the HuC scheduler, it can continue to dominate the CPU (Figure 6).

The HuC scheduler not only allocates CPU time preferentially to HuC processes, it also does so promptly. Figure 7 shows the *dispatch latency* of various process types under loaded conditions when served by the Linux scheduler and by HuC scheduler. The dispatch latencies of HuC-processes remains very low ($\leq$ 1ms), regardless of the background load.

Another point worth mentioning is the improved responsiveness of the window-manger itself. While conducting measurements involving heavy background load under the default scheduler, we have noticed that moving windows around produces extremely jerky and abrupt results. By contrast, the HuC scheduler impressively rectified this misfeature: identifying the window-manager as HuC allowed smooth window movement which (subjectively) felt as if no background load was present.

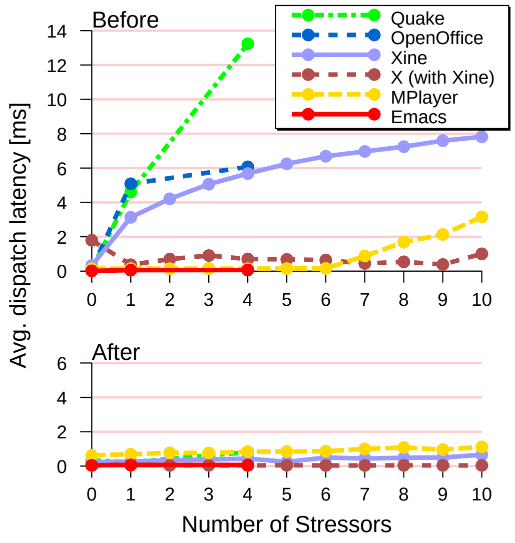

Figure 7: *Average dispatch latency of HuC applications under the Linux scheduler (top) and the HuC-scheduler (bottom).*

6. CONCLUSIONS AND DISCUSSION

To summarize, explicit identification of HuC processes is necessary to correctly prioritize them relative to background jobs. While this cannot be done by using CPU-based metrics, quantifying I/O activity provides a general and automatic solution to the problem. It is effective for both traditional interactive applications that make little use of the CPU, and for modern multimedia applications that use large amount of CPU power. It does not require any modifications to the applications themselves. The price is modifications to the system, to instrument HuC devices (in our implementation, the X server) to collect I/O information regarding the different processes, and then to use this information in the scheduler.

Once these facilities are in place, they open up a whole new spectrum of possibilities. For example, it is possible to adjust CPU allocations so that movie viewers with different sizes achieve the same frame rates. This is done by counting output production in terms of *relative* window change (namely the number of changed pixels divided by the size of the window). Allocating CPU power so as to equalize this metric will give more to viewers with a larger window, as opposed to equal allocations that will enable small viewers to complete more frames, at the same time starving the larger (probably more important) viewers.

Another example is the option to propagate "user-importance" information to other system components. This can be used to offload heavy computations that produce graphical output to other machines, or to prioritize disk/network bandwidth that is used for immediate viewing by a human user, thus improving the support for video streaming.

Finally, the concept of prioritization by I/O production opens interesting research questions: should output be quantified differently for images, text, and sound? Should input from the keyboard count the same as a mouse click/drag? Such questions beg for interdisciplinary research involving not only computer scientists but also human-interface experts and cognitive psychologists.

7. APPENDIX: SURVEY OF SCHEDULING ALGORITHMS

This review demonstrates how CPU usage is factored into the scheduling algorithms of contemporary operating systems.

The simplest example is the **Traditional Unix** scheduler [1]. The scheduler chooses processes based on priority, which is calculated as the sum of three terms: a *base* value that distinguishes between user and kernel priorities, a *nice* value (partially configurable by the ordinary user to reflect relative importance), and a *usage* value. Lower numerical values represent higher priorities. The usage is incremented on each operating system clock tick for the currently running process, so priority is reduced linearly when a process is running. On the other hand the accumulated usage of all processes is divided by a factor once a second, thus raising their priorities. The factor depends on the load: when load is high, and the process gets to run less often, the aging is also slower. **BSD Unix**, which is the basis for FreeBSD and Mac OS-X, uses a similar formula [11].

In **Linux** the priority dictates both which process is chosen to run, and how long it may run [2]. The Linux scheduler partitions time into epochs. In each epoch, every process has an allocation of how long it may run, as measured in ticks. When the process runs, the allocation is decremented on each tick until it reaches zero. Then, the process is preempted in favor of the ready process with the highest positive allocation. When there are no ready processes with an allocation left, a new epoch is started, with all processes getting a new allocation that is inversely proportional to their nice value (the lower the nice value, the higher the priority and thus the higher the allocation). In addition, processes that did not use up all their previous allocation transfer half of it to the new epoch. Thus processes that were blocked for I/O get a higher total allocation, and hence a higher priority.

Solaris is somewhat more sophisticated [10]. The Solaris scheduler supports scheduler modules, so new modules can be loaded at runtime by the administrator, thus changing the behavior of the scheduler. The default classes are time sharing (TS), interactive (IA, which is very similar to TS), system (SYS), and real-time (RT). User threads are usually handled by the TS and IA classes. Priorities and quanta are set according to a scheduling-class-specific table, which sets *(i)* the quantum length for each priority, *(ii)* the priority the thread will have if it finishes its quantum (lower), or *(iii)* if it blocks on I/O (higher). The quanta are in operating system clock tick units, and the values in the tables can be changed by the administrator. The basic idea is that higher priorities get shorter quanta: when a process finishes its quantum it gets a longer one at lower priority, and when it blocks it receives a shorter quantum at a higher priority, as opposed to what might happen under Linux.

The priority of threads in **Windows NT4.0/2000** also has static and dynamic components [18]. The static component depends on the thread's type. The dynamic component is calculated according to a set of rules, that may also give the thread a longer quantum. These rules include the following:

- Threads associated with the focus window get a quantum that is up to three times longer than they would otherwise.
- Threads that seem to be starved get a double quantum at the top possible priority, and then revert to their previous state.
- After waiting for I/O, a thread's priority is boosted by a factor that is inversely proportional to the speed of the I/O device. This is then decremented by one at the end of each quantum, until the original priority is reached again. Thus threads waiting for user input get the biggest boost, as the keyboard and mouse are amongst the slowest devices.
- Users may specify the relative importance of applications.

8. REFERENCES

[1] M. J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[2] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2001.

[3] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz, "*The Eclipse operating system: providing quality of service via reservation domains*". In *USENIX Technical Conf.*, pp. 235–246, 1998.

[4] S. Childs and D. Ingram, "*The Linux-SRT integrated multimedia operating system: bringing QoS to the desktop*". In 7th *Real-Time Tech. & App. Symp.*, p. 135, May 2001.

[5] K. J. Duda and D. R. Cheriton, "*Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*". In 17th *Symp. Operating Systems Principles*, pp. 261–276, Dec 1999.

[6] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.

[7] Y. Etsion, D. Tsafrir, and D. G. Feitelson, *Human-Centered Scheduling of Interactive and Multimedia Applications on a Loaded Desktop*. Technical Report 2003-3, School of Comp. Sci. and Eng., The Hebrew University, Mar 2003.

[8] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "*Thread-level parallelism and interactive performance of desktop applications*". In 9th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 129–138, Nov 2000.

[9] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "*Supporting time-sensitive applications on a commodity OS*". In 5th *Symp. Operating Systems Design & Implementation*, pp. 165–180, Dec 2002.

[10] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, Oct 2001.

[11] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[12] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "*SVR4 UNIX scheduler unacceptable for multimedia applications*". In 4th *Int'l Workshop Network & Operating System Support for Digital Audio and Video*, Nov 1993.

[13] J. Nieh and M. S. Lam, "*The design, implementation and evaluation of SMART: a scheduler for multimedia applications*". In 16th *Symp. Operating Systems Principles*, pp. 184–197, Oct 1997.

[14] B. Paul, *Introduction to the Direct Rendering Infrastructure*. http://dri.sourceforge.net/doc/DRIintro.html, Aug 2000.

[15] M. A. Rau and E. Smirni, "*Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads*". In *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 252–261, Oct 1999.

[16] B. Shneiderman, *Designing the User Interface*. Addison-Wesley, 3rd ed., 1998.

[17] Silicon Graphics Inc., "*OpenGL*". http://www.opengl.org/.

[18] D. A. Solomon and M. E. Russinovich, *Inside Windows 2000*. Microsoft Press, 3rd ed., 2000.

[19] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: flexible proportional-share resource management*". In *Symp. Operating System Design & Implementation*, pp. 1–11, Nov 1994.

[20] X Consortium, "*X Windows System*". www.X.org.