Scheduling of Interactive Processes

"How Can I Play a DVD and Compile Linux at the Same Time?"

Yoav Etsion
etsman@cs.huji.ac.il

School of Computer Science and Engineering
Hebrew University of Jerusalem

A thesis submitted in partial fulfillment
of the requirements for the degree of Master of Science
Supervised by Dr. Dror Feitelson

October 2002

Abstract

When we work with computers, most of the workload we generate is of an interactive nature. Our interactions with the computer are in the form of a conversation — we type something, the computers replies and so on. Applications that work in this manner are considered interactive.

Since the birth of the UNIX operating system in the early 1970s, operating systems' task schedulers regard interactive processes as I/O bound — processes that do not generate substantial CPU load and spend most of their time waiting for the user to respond.

In the last few years however, with the growing popularity of graphical cards capable of massive rendering, CD and DVD drives and the internet, there is a growing acceptance of the workstation as a multimedia console, thus introducing a new type of applications.

Modern interactive and multimedia applications require greater graphical capabilities. This feature makes them consume much more CPU cycles, so modern interactive applications no longer adhere to the interactive workload model introduced in the traditional UNIX scheduler and require a more sophisticated model.

In this thesis we explore the abilities of modern operating systems — especially with regards to the task scheduler and the operating system clock — to satisfy the needs of modern, CPU hungry, interactive applications. We show that the common CPU metric can no longer be used to classify various applications.

Instead, we offer a novel approach to dynamically and autonomously identify interactive processes based on monitoring the information flow between the different processes and the user. A scheduler based on this model can prioritize processes according to the user's desires, without any additional demands from the user.

Finally, we implement a new task scheduler for the Linux kernel based on that model. Our implementation includes some modifications to the X server and a massive rewrite of the Linux kernel's task scheduler. We present benchmarks that show the great improvement in the handling of interactive processes — with regards to minimizing dispatch latency and sufficient CPU time allocation — regardless of the load on the system.

Acknowledgements.

First and foremost, I would like to thank my advisor, Dr. Dror Feitelson, for guiding me through those uncharted waters of academic research and showing me what it means to be both a researcher and a human being.

I would also like to thank my colleague, Dan Tsafrir, for some very helpful tips and insights and for being a wailing wall during the frustrating moments of bug tracking. My thanks are also sent to our system group, mainly Danny Braniss and Tomer Klainer, for supplying me with the hardware used for this research and for stopping me from throwing any computer out the window (even though the computers sometimes deserved that).

Lastly, I would like to thank the USENIX organization for giving a very generous grant that funded this research.

Contents

1 Introduction and Related Work **1**
- 1.1 Introduction 1
- 1.2 Related Work 1
- 1.3 Our Solution 5
- 1.4 Layout of this Document 5

2 What is an Interactive Process? **7**
- 2.1 Is the determination that a process is interactive persistent? 8
- 2.2 Conclusions 9

3 The Classics: A survey of Modern Operating Systems' Schedulers **11**
- 3.1 Traditional UNIX Scheduler 11
- 3.2 BSD 4.4 11
- 3.3 Linux 12
- 3.4 Solaris 12
- 3.5 Windows NT4.0/2000 13
- 3.6 Analysis: No Reference to Interactive Processes 13

4 The Testbed **15**
- 4.1 System 15
- 4.2 Kernel LOGGER (KLogger) 15
- 4.3 Applications Tested 16

5 Operating Systems' Clock Resolution and Its effects on Real-Time and Interactive Processes **19**
- 5.1 The Resolution of Clock Interrupts 19
- 5.2 Preview of Results 20
- 5.3 Clock Resolution and Overheads 21
- 5.4 Clock Resolution and Billing 23
- 5.5 Clock Resolution and Timing 24
- 5.6 Clock Resolution and the Interleaving of Applications 25
- 5.7 Towards Best-Effort Support for Real-Time 27

6 Why is the Classic, CPU Consumption Based, Process Classification No Good anymore? **31**
- 6.1 Total CPU Consumption 31
- 6.2 CPU Consumption Distribution 33
- 6.3 CPU Consumption is Misguiding the Scheduler 34
  - 6.3.1 Interactive Processes Aren't Getting the CPU When They Need It 34

6.3.2 Interactive Processes Aren't Getting Enough CPU 36
6.4 Another Possible CPU Consumption Based Metric (and why it doesn't work) 39
6.5 Indeed, the Times They Are A'Changin... 40

7 An Alternative Metric to Identify Interactive Tasks — Information Flow Tracking 41
7.1 Following the Flow of Information 41
7.2 Interactive Devices 41
7.3 Is the Interactive Devices' Information Complete? 42
7.4 Interprocess Communication Graph 42
7.5 Quantifying Interactions With The User 44
7.6 Scheduling with Positive Feedback — Preventing Starvation 45

8 A New, User Oriented Process Scheduler For Linux 47
8.1 Maintaining the Interactive Device Statistics 47
8.1.1 Acquiring The Interactive Device Statistics — Using the X Server as a Meta-Device 48
8.1.2 Modifying the X Server Data Structures 48
8.1.3 Monitoring the User Input Using the X Server 49
8.1.4 Monitoring the User Output Using the X Server 49
8.2 Maintaining the Interprocess Communication Graph Information . . . 52
8.2.1 Identifying Interprocess Dependency 53
8.2.2 Handling of Multi-Threading/Shared Memory 53
8.3 Putting it all Together 54
8.3.1 The Stacked Scheduler 54
8.3.2 Identifying the Interactive Processes and Prioritizing Them . . 55
8.3.3 Allocating CPU Time and Choosing the Next Interactive Process To Run 57
8.4 Conclusions 57

9 Now Let's See If It Works... 59
9.1 Do Interactive Processes Get the CPU When They Need It? 59
9.2 Do Interactive Processes Get As Much CPU Time As They Need? . . 61
9.3 Conclusions 62

10 Further Research 65
10.1 Extending the Research Presented 65
10.2 Various Application of Information Flow Tracking 65

A API for the Interprocess Statistics Scheduler 67

Chapter 1

Introduction and Related Work

1.1 Introduction

The problem of scheduling the resources of a computer is decades old. While the first machines were simple number crunchers capable of running one program at a time, it soon became apparent that there is a need to allow for several programs to be served concurrently.

A simple solution for this problem was to let the programs wait in a queue and be served in a First-In-First-Out (FIFO) order. A later solution was to let the programs actually run concurrently by slicing the CPU time among them using the round robin algorithm — the multiprocessing computer was created, and gave birth to a variety of resource allocation problems.

Time-slicing also allowed programs to interact with the user in a conversation-like manner, albeit using textual medium at that time. This uncovered the problem of real time scheduling of computer resources among programs while considering the user sitting by the console waiting for a response. The new operating systems of that time, such as UNIX [2], incorporated scheduling algorithms that handled the textual based interactions with the user in a satisfactory manner.

Recent advances in hardware production have made computers cheaper and faster. As such, computers soon became a common household device that offers an abundance of interactive activities, ranging from word processing, through web surfing, and up to playing graphic-rich games and use as a multimedia console. The conversation medium was no longer limited to the textual domain.

A lot of work has been done on the subject of how to divide the CPU time between the various running applications, while keeping the interactive user happy. In the following section we review several of the common solutions for scheduling interactive applications, including modern multimedia applications. We then outline our unique contribution.

1.2 Related Work

In this section we will review the evolution of interactive processes' schedulers:

Classic Operating Systems' Scheduler Design

At first, interactive processes were textual, and all communication between the user and the various applications used the terminal. Task schedulers of common operating systems tried to identify the interactive processes and give them some special treatment, to prevent the user from experiencing the system load through longer response times. The UNIX task scheduler [2] works under the assumption that interactive processes spend most of their time waiting for a response from the user — as the user lives in a temporal domain much slower than that of the processor — and uses a simple policy of prioritizing processes that rarely use the CPU over those that do so often. In a sense, this is a simple fair share approach — the scheduler tries to distribute the processing time evenly among the different processes (with some consideration given to user-assigned weights passed through the *nice* mechanism).

Other operating systems took this support one step further by identifying which device a process was waiting for. Operating systems such as *Sun's Solaris* [29], *Digital's VAX/VMS* [25] and *Microsoft's Windows NT/2000/XP* [48, 49] give a priority boost to processes that were blocked on a device with inverse proportion to the speed of the device — the slower the device, the higher the priority boost. This means the a process waiting for the slower terminal device gets a major boost, in a scheme that implicitly favors interactive processes.

Microsoft added another feature to their *Windows NT/2000/XP* operating systems: the thread owning the focus window gets an additional priority boost. These operating systems thus had the first mainstream scheduler that explicitly recognizes the interactiveness of applications, as opposed to the previous implicit solution.

These solution worked well when human interaction was limited to the textual domain. Contemporary computer workloads however, especially on the desktop, contain a significant multimedia component: use of graphical user interfaces, playing of music and sound effects, displaying video clips and animations, etc. These workloads are not well supported by the mechanisms described so far as shown by Nieh et al. [34] and further investigated in chapter 6.

This deficiency is often attributed to the fact that multimedia applications consume significant CPU resources themselves — which breaks the basic assumption of the UNIX scheduler — and to the lack of specific support for real-time features.

In recent years the topic of scheduling multimedia applications has become a crowded research area, in which we can identify several common approaches. The main ones are support for soft real time applications and for proportional sharing of resources.

Multimedia Schedulers with Soft Real Time support

Multimedia is commonly associated with soft real time features. Sustained frame rate and constant audio sample rate are two major features which make multimedia applications require soft real time support. Several research projects have been done to enhance multimedia support through better soft real time support from the operating system.

Soft Timers [1] use general kernel entry points, such as system calls, as opportunities to execute software timers. This is done in an attempt to give better timer accuracy than the common solution of executing all software timers from within the general clock interrupt handler which is called at a constant (and usually coarse) rate. This solution is shown to highly enhance the operating system's soft real time properties, albeit based on a statistical principle.

The SMART scheduler [35] by Nieh et al. lets a multimedia application request the operating system for certain soft real time assurances — mainly computation periods — and receive feedback from the operating system whether these requests can be met. Originally implemented on the Solaris operating system, and later implemented for Linux [56], it presented a substantial improvement to the native support. This solution however, mandates modifications to applications by forcing them to explicitly request resources from the operating system.

The applications' modifications problem was addressed by the BEST scheduler [3]. This scheduler identifies applications with soft real time qualities — application which have regular computation periods. These periods are recorded and the operating system attempts to anticipate future periods and schedule the applications accordingly. While this solution does not require any participation on behalf of the application itself, it cannot handle overloaded machines in which such computation periods overlap.

On a different note, Zhang et al. [60] approached a different version of the same problem. Their goal was to schedule real time jobs along best-effort ones in a manner which will maintain the real time deadlines. Their design was aimed at serving parallel real time multimedia applications that are too demanding to be serialized. An example for such applications is detecting motion on a stream of images taken by an airplane by pipelining the frames through a series of threads running on different processors. Their proposed solution is dividing the CPU time among the two classes according to a user supplied "fairness" ratio, and letting each class schedule its processes in a hierarchical model. The real time class uses the earliest-deadline-first (EDF) scheme.

Proportional and Fair Share Schedulers

Another common solution is to enable the user/programmer to request some resource guarantees from the operating system: a certain percentage of the CPU time, disk bandwidth etc.

One of the well known schemes in this field is *Lottery Scheduling* [57]. The basic idea is to assign each process a number of lottery tickets that is proportional to its requested percentage of the CPU time. The scheduling decision is than made by drawing a uniformly distributed ticket thus giving each process chance to "win" the CPU that is distributed according to the user/programmer's requests.

This work was later developed into *Stride Scheduling* [58] which was aimed at turning the probabilistic factor at the base of *Lottery Scheduling* into a deterministic one in order to minimize the mean error in the average throughput and latency that accompanies the probabilistic *Lottery Scheduling*.

Another well known work in this field is the *Borrowed-Virtual-Time* scheduler [11]. This scheduler assigns a virtual time to each running thread, and allocates the CPU according to a user-defined weight policy. A time sensitive thread can "borrow" time from its future allocations when its schedule is tight. This can minimize the thread's dispatch latency in time-critical sections.

Stoica et al. [51] proposed another proportional share scheduling mechanism — processes are continuously allocated CPU shares according to a user defined rate.

Mercer et al. [31] devised an operating system abstraction called *reserve* that allows a server process to bill the client process for its CPU time thus allowing a more accurate billing according to user-defined weights. Their work was designed for microkernel operating systems in general (and implemented on the Mach kernel [52]) since such operating systems rely heavily on user level processes to provide elementary services.

An adaptive approach was taken by Rau et al. [39] when designing a scheduler that monitors the CPU demands of both Multimedia and Best-Effort applications and trying to accommodate those needs by adapting itself to the momentary workload. The user must specify two limits: a highest tolerable percentage of missed deadlines in multimedia applications and a highest tolerable slowdown for best-effort applications. The scheduler than tries to change the CPU portions allocated for each class using 5% chunks in order to maintain both limits over the possibly changing workload. In case of a heavy load under which the system cannot maintain both limits, an advantage is given to the multimedia class whose limit is the one to be maintained.

The Eclipse operating system [6] went a step further in providing proportional share of the machine to the various running processes. Its scheduling algorithm, called *Move-To-Rear* [7], enables the user/programmer to guarantee portions of other resources — such as memory blocks, disk bandwidth, network bandwidth etc. — as well as CPU time to the running processes. In particular, the algorithm is designed so that latencies incurred waiting for different resources do not accumulate over time.

Hierarchical and Modular Schedulers

Some work investigated the behavior of so called "meta-schedulers" — schedulers that actually allocate CPU time between other, class specific schedulers.

Goyal et al. [18] designed a hierarchical scheduler fashioned like a tree. Each leaf is a class specific scheduler and each internal node symbolizes a meta-scheduler with a specific proportional CPU time division between its children.

In his PhD thesis, Guo [20] extended this hierarchical scheduler model. He designed a priority based algorithm, unique in that it identifies the server processes a client is serviced by. It then increases the server processes' priorities to that of the client (if necessary) to prevent the common cases of priority inversion.

The *Vassal* project [8] from *Microsoft* described a hierarchical scheduler with strict ordering: when a dispatch decision is to be made the dispatcher queries the various class-specific schedulers for processes, in a strict, predefined order. The focus of this work is to enable the dynamic loading and unloading of schedulers at runtime, according to the applications' demands. This principle is somewhat similar to the mechanism we described as the *Stacked Scheduler* (section 8.3.1).

Summary of the Common Approaches

As we have seen, all these schedulers' attempt to find ways to allocate CPU time to the running applications in a manner that will keep the user happy, and force the computer the behave according to the user's expectations.

Although some of these schemes are very elaborate, there is one common downside to all of them: they still require the user/programmer to manually specify the needs of the various application — either in terms of deadlines or of relative weights — and use this hand tuned process identification as a guide by which to distribute resources.

The only exception to this approach is the *BEST* scheduler [3] that tries to automate the identification of soft real time processes and their requirements, rather then letting the user manually supply this information. But even this provides an automation that is limited to soft real time processes, cannot distinguish between interactive and non-interactive processes, and cannot handle overloaded machines.

Our goal is to fully automate this process, and make the computer anticipate the

user, his desires and interests. Resource allocations will then be based on this understanding of the user's interests.

1.3 Our Solution

In this thesis we introduce a novel approach to process scheduling based on identifying interactive processes by monitoring their interactions with the user.

The novelty in our research is that contrary to other solutions — aimed at providing a *mechanism* enabling the application programmer or the user to request certain resources from the operating system — our solution is focused on providing a *policy* that will enable the scheduler to choose the process the user is interested in, without any need for special input from either the application programmer or the user. The *policy* is rather based on understanding the properties of interaction with the human user.

This identification of interactive processes is dynamic and requires no cooperation from either the user or the programmer. It is thus fully independent, user oriented and is compatible with the new *human centered computing* paradigm.

Another benefit in this approach is that there is no need for specialized interfaces, that may reduce the portability of applications, and require a larger learning and coding effort.

On a secondary note we also try to investigate the effects of clock resolution on the scheduling of interactive and real time processes, and show that a little change in the operating system's clock can lead to great benefits.

1.4 Layout of this Document

An overview of this thesis is as follows: we first discuss the properties of an interactive process (chapter 2), then turn to a detailed review of common operating systems' schedulers (chapter 3). After describing our testbed (chapter 4), we give an analysis of the effects of the operating system's clock on interactive and real time applications (chapter 5) and analyze why common schedulers fail to handle multimedia applications (chapter 6). We then introduce the *information flow tracking methodology* (chapter 7) on which we base our proposed scheduler (presented in chapter 8) and continue to describe our experimental results (chapter 9). Finally, we ponder on future applications of our research (chapter 10).

Chapter 2

What is an Interactive Process?

When investigating how an operating system's scheduler should handle interactive processes, the first step should be formalizing a definition for an *interactive process*.

The *Free On-line Dictionary of Computing* [22] defines an interactive process as:

> A term describing a program whose input and output are interleaved, like a conversation, allowing the user's input to depend on earlier output from the same run.
>
> The interaction with the user is usually conducted through either a text-based interface or a graphical user interface. Other kinds of interface, e.g. using speech recognition and/or speech synthesis, are also possible.
>
> This is in contrast to *batch* processing where all the input is prepared before the program runs and so cannot depend on the program's output.

This definition however, is a little simplistic for two reasons: First, the interaction between the user and an application might be indirect via a third process, and second, the rate of user interactiveness is spread over a wide spectrum.

In most UNIX systems the user keyboard and mouse inputs are not delivered directly to the application, but rather to the X server [59] and the window manager. These are responsible to deliver the input event to the application, so the user interaction might not even be a direct one. Other, more extreme examples include the UNIX X terminal [59] which emulates a rudimentary computer terminal inside the X Windows System for the purpose of running console based applications. In this case the user I/O is proxied to and from the application by both the X server *and* the X terminal.

Such examples might even include cases of *remote interactiveness* — an X application running on one machine and connected to an X server on another machine, making it interactive on one computer, but relative to a remote user. A more extreme example is a web server that interacts with an internet browser. In this case the server interacts indirectly with a human user (actually, because of the X windows system it interacts directly with an application that interacts indirectly with the user - complicated, hah?). The web browser may quite possibly be running on a remote computer, making it another example of remote interactiveness.

There is a wide variety of interactive applications. The user feels a delay in an application's response only relatively to its expected runtime (or query time): he frequently interacts with applications whose response is expected to be quick, and infrequently interacts with slower applications (by "interact" we also mean that the user checks if

an application finished its work and has produced some output). By identifying this difference in frequency, or granularity of the interactions, we have placed the different applications on a vast spectrum of different frequencies. On one hand we have the role playing games, to which the user reacts almost immediately after the game draws something on the screen (and in the opposite direction, the game redraws the screen almost immediately after the user sends some input event — key press, mouse move etc.) and movie players: both types need to maintain a constant frame rate, which requires CPU resources as well as limited dispatch latency — almost real time assurances. On the more frequently-interactive side of the spectrum we also have text editors to which the user sends bursts of inputs. Going toward the other end of the spectrum we'll see applications such as web browsers and web servers (on the more frequent side), and even compilers (less frequent)

This spectrum indicates that a frequency threshold has to be set, so that every application whose interaction frequency is higher than the threshold will be considered interactive, and we will try to improve its response time. Applications over the threshold include both internet browsers which have a big response time, largely due to network latency and bandwidth limitations, and multimedia and graphical applications which require fixed CPU usage and dispatch latency. This threshold is agreed upon among human computer interaction specialists **to be in the area of a few seconds**, with the exact threshold depending on the user and the hardware (on slower hardware the user will probably be aware that applications may be slower, thus a longer threshold).

2.1 Is the determination that a process is interactive persistent?

After we have determined what makes a process interactive, we face the next question: can the interactive status change during a process' lifetime, or simply — can a process be considered interactive one minute an non-interactive the next?

The answer to this question is yes. In current working environments, a user may have more applications openned than he can monitor simultaneously (in other words, a user open more windows than he can watch). This results in some applications being dormant while waiting for input from the user. An example of this is an Emacs editor that is open and showing some file being edited while the user is surfing the net — the editor is in general an interactive application but this instance of the application is not interactive at the moment.

But this is the simple case. Some applications might actually be dormant but appear to be active: An application can be waiting for input from the user to continue its work, even though it still requires CPU resources (so it is not dormant from the operating system's point of view). Just imagine an instance of netscape that is displaying a web page with some graphics (animated gif, for example) that needs redrawing: it requires CPU resources to redraw the graphics, but effectively does nothing but wait for the user to click the mouse on some link.

From this observation we can deduce that a process is only interactive relating to its frequency of communication with the user during a period of time. An interactive process's lifespan might also have times when the user is working on some other application and checks the output from this process less frequently. If this frequency crosses the interactive frequency threshold, the process might be demoted to be non-interactive until the user regains interest in it and the interaction frequency crosses the threshold

back to interactive level.

2.2 Conclusions

In this chapter we have discussed the characteristics of an interactive application, and found that there is a plethora of such, so we must limit our quest.

For the remainder of this document we will not handle cases of *remote interactiveness*, and limit our discussion to local interactive application. We do however address the issues of both direct and indirect interactiveness, and offer a novel approach to dynamically identify such processes.

Chapter 3

The Classics: A survey of Modern Operating Systems' Schedulers

This review of scheduling policies in contemporary operating systems demonstrates how CPU usage is factored into scheduling decisions, therefore making accurate billing important. It also reviews the various operating systems' clock resolution (note that billing is always done in operating system clock tick units). It also lists typical clock interrupt rates and subsequent scheduling time quanta.

3.1 Traditional UNIX Scheduler

In **Traditional UNIX** [2] 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 representing relative importance, and a *usage* value. Lower numerical values represent higher priorities. The usage is incremented on each clock tick for the currently running process, so priority is reduced linearly when a process is running. However, this is at tick resolution, so running for less than a tick is unaccountable. The usage is reduced each second for all processes according to the following formula, where *load_avg* is the average length of the runqueue in the last second:

$$usage = \frac{2\,load_avg}{2\,load_avg + 1} \times usage$$

Thus when the load is high, and the process gets to run less often, the aging is also slower.

3.2 BSD 4.4

BSD Unix[30, 16], which is the basis for FreeBSD[1]and Mac OS-X, uses a similar formula.

[1]FreeBSD has a new proposed development scheduler that is based on a proportional share algorithm, as opposed to the older priority feedback based algorithm. Since the change is not relevant to interactive processes specifically, and is not yet part of the stable distribution, we discuss the stable algorithm.

One difference is that CPU consumption by the current process is only tabulated once every four ticks. This makes the resolution problem worse than in traditional Unix. Another is that the *nice* value is also added into the aging of the *usage*, so processes with high priority (negative *nice*) get some of their CPU usage for free, whereas processes with low priority (large *nice*) look as if they used more CPU than they actually did. The time quantum in BSD is fixed at 100 ms.

3.3 Linux

In **Linux**[2] [4, 5, 54] the priority dictates both which process is chosen to run, and how long it may run.

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 reduced on each tick. 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 interactive processes that were blocked for I/O get a higher total allocation, and hence a higher priority. Allocations in an epoch are in the range of 6–11 ticks (in Linux 2.4), and then a process is preempted. Special cases exist, though: when a process forks, its allocation is split between the parent and child processes, and when a process terminates, its remaining allocation is added to its parent.

3.4 Solaris

Solaris [29, 32] is somewhat more sophisticated. 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).

A scheduler module registers itself with the kernel, specifying the range of priorities it uses. The Solaris scheduler has 170 global priority levels, and each scheduler module specifies a range of priorities is uses. Module priorities can overlap.

User threads are usually handled by the TS and IA classes, which are very similar. Priorities and quanta are set according to a scheduling-class-specific table, which sets the quantum length for each priority, and the priority the thread will have if it finishes its quantum (lower) or if it blocks on I/O (higher). The quanta are in 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 clock is set to a frequency of 100 Hz, both on the UltraSPARC and on the i386 architectures. The clock tick handler calls the scheduling-class-specific tick handler for thread runtime accounting. In both the TS and IA classes this decrements the *cpuleft*

[2] Actually, the new development version of the Linux kernel also has the scheduler completely rewritten [33]. However, since the changes relate to the scheduler's data structures and not the algorithm proper, we discuss the stable version's scheduler.

counter in the process structure. The quanta in these classes is in the range of 2–20 ticks.

Solaris' default timer interrupt frequency is 100Hz, but it can be changed to 1000Hz using a parameter in the */etc/system* configuration file. This is however highly discouraged by the kernel commentators [29].

3.5 Windows NT4.0/2000

The priority of treads in **Windows NT4.0/2000** [48, 49] also has static and dynamic components. The static component depends on the process and thread. 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 (this rule only applies to the NT Workstation version).
- Threads that seem to be starved get a double quantum at the top possible priority, and then revert to their previous state.
- Threads that wait for user input from a GUI get a double quantum at a priority level that is one less than the maximum, 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.

The net effect is a large boost for threads that are explicitly interactive, at the expense of others. CPU usage enters into the equation in its effect on terminating the boost, and in the special handling of starving processes.

The basic quantum in NT Workstation is 6 units, and in NT Server it is 36 units. On each clock tick the scheduler deducts 3 units from the running thread's quantum, so even though the quantum unit is less than a tick, the scheduler's resolution is not improved. The length of a tick is 10–15ms, depending on the processor (from 486 to Pentium4) and the number of processors present (multiprocessor or uniprocessor). This gives a resolution of 66–100 Hz.

3.6 Analysis: No Reference to Interactive Processes

It is quite obvious that none of the reviewed schedulers acknowledge the fact that interactive processes requires special care, and all treat them as simple I/O bound processes.

The only special handling available is by the *Windows* scheduler (section 3.5) which gives a longer quantum to the thread associated with the focus window. Although unique, this is a very simplistic approach and not necessarily a good one.

In the following chapters, we will discuss the adequacy of the 30 year old assumption — that interactive processes are merely I/O ones in disguise — and we will show that it is no longer true for modern workstations' common workload.

Chapter 4

The Testbed

In this chapter we will review all the hardware and software systems used to evaluate our experiments.

4.1 System

Our hardware consists of a 664MHz Pentium III (Coppermine) machine equipped with 256 MB RAM, with a 3DFX Voodoo3 graphics accelerator with 16 MB RAM that supports openGL in hardware.

Linux is becoming more and more common as a desktop operating system. Because of its open source nature and the availability of many kernel information resources it can be modified and monitored to measure performance.

For that reason we used the 2.4.8 version of the Linux kernel [54] as the operating system running with the RedHat 7.0 Linux distribution [40]. The kernel's hardware clock was changed from the default 100Hz frequency to 1000Hz based on our investigation of clock effects (see chapter 5).

For the window system we used the open sourced XFree86 X Windows implementation, version 4.1.0 [53].

4.2 Kernel LOGGER (KLogger)

The measurements were conducted using **klogger**, a kernel logger we developed that supports fine-grain events. In order to reduce interference and overhead, logged events are stored in a largish buffer in memory (we typically use 4MB), and only exported at large intervals (by a daemon that wakes up every few seconds; the interval is reduced for higher clock rates to ensure that events are not lost; in latency measurements, the intervals during which klogger data was offloaded were explicitly excluded). The implementation is based on inlined code to access the CPU's cycle counter and store the logged data. Each event has a 20-byte header including a serial number and timestamp with cycle resolution, followed by event-specific data. The overhead of each event is only a few hundred cycles (we estimate that at 100Hz the overhead for logging is 0.63%, and at 1000Hz it is 0.95%). In our use, we log all scheduling-related events: context switching, recalculation of priorities, forks, execs, changing the state of processes, and monitoring of activity on Unix-domain sockets (to track potential in-

teractions with the X server). While the code is integrated into the kernel, its activation at runtime is controlled by applying a special *sysctl* call using the /proc file system.

4.3 Applications Tested

Our measurements were based on several applications, trying to represent the variety of interactive applications. We also used two applications as background load, which were modeled to represent common background load applications.

The classical interactive applications were represented by the *GNU Emacs text editor [14].* To simulate reasonable user I/O, Emacs was measured with a type rate of ~ 8 characters per second, over a period of 60 seconds.

Modern interactive applications are more diverse, hence were represented by several examples:

- *XFree86 X Windows Server [53].* In UNIX systems, the X server is responsible for all interactions with the user as it acts as a virtual console. As such, this server can be thought of as a user proxy from the workstation's point of view.

  This version of the server includes two important extensions to the X protocol: the first is the *MIT Shared Memory (MIT-SHM) extension*, which is used to transfer images via shared memory rather than using multiple buffer copies with UNIX domain sockets, thus reducing the load on the CPU. The other extension is the *Direct Rendering Infrastructure (DRI) [37]* which allows a graphical application direct access to the display adapter's GPU instead of proxying all these request through the X server.

- *OpenOffice [24].* This is a modern, full fledged office productivity suit, with graphical capabilities. We used its text editor as an example of a modern editor. The measurements were similar to those used with Emacs, i.e. roughly 8 characters per second, over 60 seconds.

- *Xine movie player [13].* This is an example of a multi-threaded application. Xine uses the MIT-SHM X protocol extension. *Xine* was measured while playing a 40 seconds MPEG movie - the movie we used was encoded using the MPEG-1 standard [19], at 25 fps and size of $352x288$ pixels. All measurements of *Xine* were zooming the movie at a 2:1 ratio - twice its size.

- *MPlayer movie player [38].* This is a single threaded player. MPlayer also uses the MIT-SHM X protocol extension. *MPlayer* was measured playing the same 40 seconds MPEG movie as *Xine* did, but at normal size.

- *Quake III Arena [47].* This is an example of a modern role playing game, with heavy graphical requirements. *Quake* uses the OpenGL [23] library to render graphics. *Quake* can run in two different modes: it can either run in normal mode in which a human player is managing the game, or in demo mode in which the computer plays a game by itself. We measured both modes, to see the effect of input from the X server on the process' CPU usage. Both measurement lasted 50 seconds.

For comparison, we measured the CPU usage of two types of common non-interactive applications:

- Compilation : A compilation the linux kernel. Since there are several processes involved in the compilation the data shown is the summation of all processes involved. Such a compilation takes about 400 seconds.
- CPU Intensive : To represent a CPU intensive application, a simple program was used. The program is an infinite loop incrementing an integer, and is referred to as a *stresser*. The *stessers* were measured over a 300 seconds period.

Chapter 5

Operating Systems' Clock Resolution and Its effects on Real-Time and Interactive Processes

It is generally agreed that scheduling mechanisms in general purpose operating systems do not provide adequate support for modern interactive applications, notably multimedia applications. The common solution to this problem is to devise specialized scheduling mechanisms that take the specific needs of such applications into account. A much simpler alternative is to better tune existing systems. In particular, we show that conventional scheduling algorithms typically only have little and possibly misleading information regarding the CPU usage of processes, because increasing CPU rates have caused the common 100Hz clock interrupt rate to be coarser than most application time quanta. Significant increases in clock interrupt rates are possible with acceptable overheads, and lead to much better information. In addition, they provide a measure of support for soft real-time requirements. However, under loaded conditions, the system may still be unable to distinguish between CPU-intensive multimedia applications and background CPU-intensive tasks.

5.1 The Resolution of Clock Interrupts

Computer systems have two clocks: a hardware clock that governs the instruction cycle, and an operating system clock that governs system activity. Unlike the hardware clock, the frequency of the system clock is not predefined: rather, it is set by the operating system on startup. Thus the system can decide for itself what frequency it wants to use. It is this tunability that is the focus of the present paper.

The importance of the system clock (also called the timer interrupt rate) lies in the fact that systems measure time using this clock, including CPU usage and when timers should go off. The most common frequency used today is 100Hz, and hasn't changed much in the last 30 years. For example, back in 1976 Unix version 6 running on a PDP11 used a clock interrupt rate of 60Hz [27]. At the same time the hardware clock rate increased by about 3 orders of magnitude [41]. As a consequence, the size of an

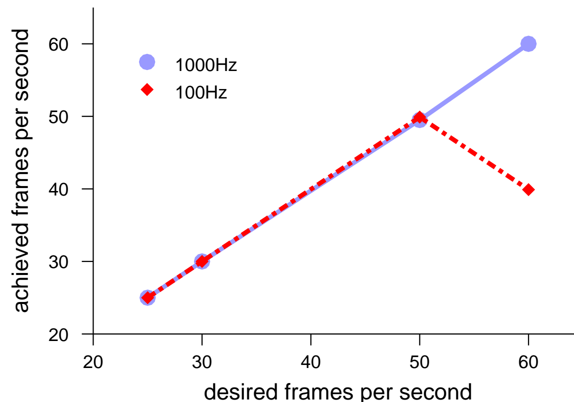


Figure 5.1: *Desired and achieved frame rate for the Xine MPEG viewer, on systems with 100Hz and 1000Hz clock interrupt rates.*

operating system tick has increased a lot, and is now on the order of 10 million cycles or instructions. Simple interactive applications such as text editors don't require that many cycles per quantum, making the tick rate obsolete — it is too coarse for measuring the running time of an interactive process. For example, the operating system cannot distinguish between processes that run for a thousand cycles and those that run for a million cycles, because using 100Hz ticks on a 1 GHz processor both look like 0 time.

Another problem is providing support for real-time applications such as games with realistic video rendering, that require accurate timing down to several milliseconds. These applications require significant CPU resources, but in a fragmented manner, and are barely served by a 100Hz tick rate. In some cases, the limited clock interrupt rate may actually prevent the operating system from providing required services. An example is given in Figure 5.1. This shows the desired and achieved frame rates of the Xine MPEG viewer showing 500 frames of a short clip that is already loaded into memory, when running on a Linux system with clock interrupt rates of 100Hz and 1000Hz. For this benchmark the disk and CPU power are not bottlenecks, and the desired frame rates can all be achieved. However, when using a 100Hz system, the viewer repeatedly discards frames because the system does not wake it up in time to display them if the desired frame rate is 60 frames per second.

Increasing the clock interrupt rate may be expected to reduce and maybe even overcome these problems, but this comes at the expense of additional overhead. In this chapter we focus on a single simple tuning knob — the clock interrupt resolution, and investigate the benefits and the costs of turning it to much higher values than commonly done, without changing the scheduling algorithm.

5.2 Preview of Results

Our initial goal is to show that increasing the clock interrupt rate is both possible and desirable. Measurements of the overheads involved in interrupt handling and context switching indicate that current CPUs can tolerate much higher clock interrupt rates than those common today (Section 5.3). We then go on to demonstrate the following:

- Using a higher tick rate allows the system to perform much more accurate billing, thus giving a better discrimination of interactive processes (Section 5.4). This is a real issue with typical interactive workloads on today's machines.

Table 5.1: *Timer interrupt overhead (average±standard deviation).*

Clock	Load	Interrupt processing		Context switch	
		Cycles	μs	Cycles	μs
100 Hz	unloaded	4157±211	6.25	702±909	1.06
	with Xine	7571±1967	11.39	1743±2102	2.62
1000 Hz	unloaded	4176±131	6.28	734±899	1.10
	with Xine	4731±822	7.12	1604±1802	2.41

- Using a higher tick rate also allows the system to provide a certain "best effort" style of real-time processing, in which applications can obtain high-resolution timing measurements and alarms (as exemplified in Figure 5.1, and expanded in Section 5.5). For applications that use time scales that are related to Human perception, a modest increase in tick rate may suffice. Applications that operate at smaller time scales, e.g. to monitor certain sensors, may require much higher rates and shortening of scheduling quantum lengths (Section 5.7).

We feel that improved clock resolution — and the shorter quanta that it makes possible — have to be a part of any solution to the scheduling of interactive applications, and should be taken into account explicitly.

5.3 Clock Resolution and Overheads

A major concern regarding the increase of the clock interrupt rate is the resulting increase in overheads: with more clock interrupts more time will be wasted on processing them, and there may also be more context switches, which in turn lead to reduced cache and TLB efficiency. This is the reason why today only the Alpha version of Linux employs a rate of 1024Hz (according to the Linux Kernel mailing list this is because the Alpha is "strong enough to handle it"). This is compounded by the concern that operating systems in general become less efficient on machines with higher hardware clock rates [36]. We will show that these concerns are unfounded, and a clock interrupt rate of 1000Hz or more is perfectly possible.

In Linux, clock interrupts are handled by the `timer_interrupt` function, which is called from `do_IRQ`, the main interrupt dispatch function. Using the klogger infrastructure we measured the execution time of the handler function for kernels running at both 100Hz and 1000Hz interrupt rates. To test the effect of load we ran the tests both on an unloaded machine, and on a machine running Xine.

The results are shown in Table 5.1, and indicate that the handler overhead is relatively small and largely independent of the clock interrupt rate. Handling a clock interrupt takes less than 4200 cycles; on our 664 MHz machine, this causes an overhead of only 0.07% if called at 100Hz, and a higher but still negligible 0.7% if called at 1000Hz. Higher rates would also be tolerable: system overheads measuring 10–30% were the norm a decade ago [10]. A context switch takes even less time, although it grows with load.

An interesting phenomenon is that on a loaded system the average handling time actually *drops* when the clock interrupt rate is increased. We are not sure about why this happens, and suspect cache effects. The possibility that it is due to the accumulation of timer events that need to be handled was checked and refuted.

Table 5.2: *Overheads on different processor generations.*

Processor	Interrupt processing		Context switch		Cache BW	Trap	
	Cycles	μs	Cycles	μs	MB/s	Cycles	μs
P-90	939±379	10.41	2056±723	22.80	28.01±0.82	187±201	2.08
PP-200	1648±376	8.28	1576±468	7.92	438.49±13.91	379± 85	1.91
PII-350	2372±237	6.79	1451±409	4.16	828.23±17.37	344±102	0.98
PIII-664	4098±695	6.17	1375±468	2.07	2512.06±32.76	346± 23	0.52
PIII-1.133	6475±566	5.73	1356±517	1.20	2683.13±36.80	364±266	0.32
A1.6	11246±662	7.03	2004±502	1.25	4086.78±60.81	291± 79	0.18
PIV-2.2	14130±573	6.44	3978±1191	1.81	3572.57±61.68	1717± 69	0.78

Table 5.3: *Time between successive timer interrupts (average±standard deviation).*

Clock	Load	Interval in cycles	Clock rate
100 Hz	unloaded	6645205.97±3390.41	99.9969 Hz
	with Xine	6645206.34±5891.94	99.9969 Hz
1000 Hz	unloaded	664409.23±1463.48	1000.1366 Hz
	with Xine	664409.22±6358.13	1000.1366 Hz

How are these results expected to change on future machine generations? Ousterhout has claimed that operating systems do not become faster as fast as hardware [36]. We have repeated some of his measurements on a range of Pentium processors with clock rates from 90MHz to 2.2GHZ, and on an Athlon at 1.6GHz with DDR-SDRAM memory. Our results, listed in Table 5.2, show the following. First, we find that the overhead of processing a clock interrupt is dropping at a much slower rate than expected according to the CPU clock rate. This is due to an optimization of the gettimeofday() accuracy by accessing the 8253 timer chip on each clock interrupt, and is therefore not related to the CPU clock rate. But even with this optimization, the overhead is still short enough to allow many more interrupts than are used today, up to an order of 10,000Hz. Second, we find that the overhead for context switching takes roughly the same number of cycles, regardless of CPU clock speed (except on the Pentium4, which is using SDRAM memory at 133MHz and not the newer RDRAM). We also found that the trap overhead and cache bandwidth behave similarly. This is more optimistic than Ousterhout's results. The difference may be due to the fact that Ousterhout compared RISC vs. CISC architectures, and there is also a difference in methodology: we measure time and cycles directly, whereas Ousterhout based his results on performance relative to a MicrovaxII and on estimated MIPS ratings.

A potential problem with increasing the clock interrupt resolution stems from the fact that the Linux kernel is monolithic and non-preemptable. It therefore contains many pieces of code in which interrupts are blocked. Having more clock interrupts runs the risk of conflicting with these code sections, leading to timing inaccuracies. Results of measuring the times between the handling of successive clock interrupts are shown in Table 5.3. As we can see, the interrupt-blocked code sections in the kernel do not cause a major loss of timer interrupts, or mishandling them, even under loads. The measured timer frequency is very similar to the programmed one, and the low standard deviation suggests a relatively constant interrupt rate.

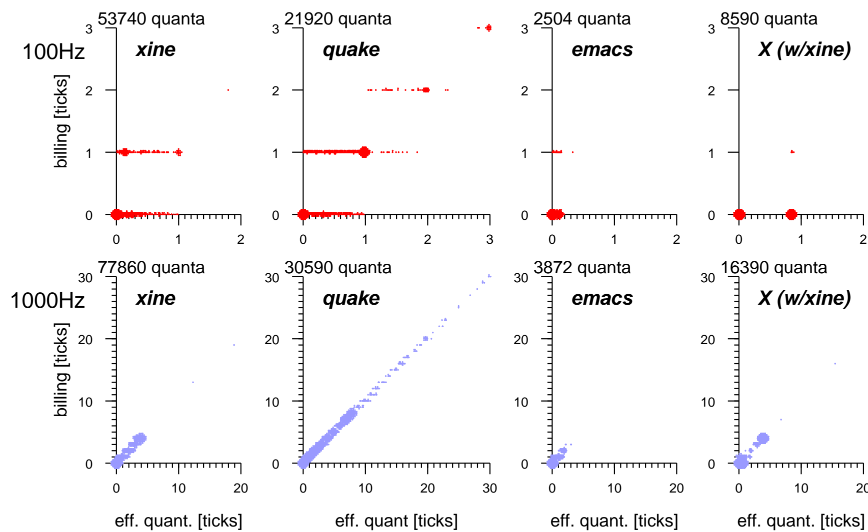


Figure 5.2: *The relationship between quanta durations and how much the process is billed, for different applications, using a kernel running at 100Hz and at 1000Hz. Concentrations of data points are rendered as larger disks; otherwise the graphs would have a clean steps shape, because the billing (Y axis) is in whole ticks. Note also that the optimal would be a diagonal line with slope 1.*

Table 5.4: *Scheduler billing success rate.*

	Billing ratio		Missed quanta	
Application	@100Hz	@1000Hz	@100Hz	@1000Hz
Emacs	1.0746	0.9468	95.96%	73.42%
Xine	1.2750	1.0249	89.46%	74.81%
Quake	1.0310	1.0337	54.17%	23.23%
X Server (w/Xine)	0.0202	0.9319	99.43%	64.05%
CPU-bound	1.0071	1.0043	7.86%	7.83%
CPU-bound (w/Quake)	1.0333	1.0390	26.71%	2.36%

5.4 Clock Resolution and Billing

Practically all operating systems use priority-based schedulers, and factor CPU usage into their priority calculations as discussed in chapeter 3. CPU usage is measured in ticks, and is based on sampling: the process running when a clock interrupt occurs is billed for this tick. But the coarse granularity of ticks implies that billing may be inaccurate, leading to inaccurate information used by the scheduler.

The relationship between actual CPU consumption and billing is shown in Figure 5.2. The X axis in these graphs is the effective quantum length: the exact time from when the process is scheduled to run until when it is preempted or blocked. While the effective quantum tends to be widely distributed, billing is done in an integral numbers of ticks. In particular, for Emacs and X the typical quantum is very short, and they are practically never billed!

Using klogger, we can tabulate all the times each application is scheduled, for how

much time, and whether or not this was billed. The data is summarized in Table 5.4. The billing ratio is the time for which an application was billed by the scheduler, divided by the total time actually consumed by it during the test. The miss percentage is the percentage of the application's quanta that were totally missed by the scheduler and not billed for at all.

The table shows that even though very many quanta are totally missed by the scheduler, especially for interactive applications, most applications are actually billed with reasonable accuracy in the long run. This is a well-known probabilistic phenomenon. Since most of the quanta are shorter than one clock tick, and the scheduler can only count in complete tick units, many of the quanta are not billed at all. But when a short quantum does happen to include a clock interrupt, it is over billed and charged a full tick. On average, these two effects tend to cancel out, because the probability that a quantum includes a tick is proportional to its duration. The same averaging happens also for quanta that are longer than a tick: some are rounded up to the next whole tick, while others are rounded down.

A notable exception is the X server when running with Xine (we used Xine because Xine intensively uses the X server, as opposed to Quake which uses DRI). According to Figure 5.4, when running at 100Hz this application has quanta that are either extremely short (around 68% of the quanta), or around 0.8–0.9 of a tick (the remaining 32%). Given the distribution of quanta, we should expect about 30% of them to include a tick and be counted. But the scheduler misses over 99% of them, and only bills about 2% of the consumed time! This turns out to be the result of synchronization with the operating system ticks. Specifically, the long quanta always occur after a very short quantum of a Xine process that was activated by a timer alarm. This is the process that checks whether to display the next frame. When it decides that the time is right, it passes the frame to X. X then awakes and takes a relatively long time to actually display the frame, but just less than a full tick. As the timer alarm is carried out on a tick, these long quanta always start very soon after one tick, and complete just before the next tick. Thus, despite being nearly a tick long, they are hardly ever counted.

When running the kernel at 1000 Hz we can see that the situation improves dramatically — the effective quantum length, even for interactive applications, is typically several ticks long, so the scheduler bills the process an amount that reflects the actual consumed time much more accurately. We can also see the dramatic improvement in the X server: on a 1000 Hz system it is billed for over 93% of the time it consumed, with the missed quanta percentage dropping to 64% — the fraction of quanta that are indeed very short.

5.5 Clock Resolution and Timing

Increasing the kernel's clock resolution also yields a major benefit in terms of the system's ability to provide accurate timing services. Specifically, with a high-resolution clock it is possible to deliver high-resolution timer interrupts. This is especially significant for real-time applications such as multimedia players, which rely on timer events to keep correct time.

A striking example was given in the introduction, where it was shown that the Xine MPEG player was sometimes unable to display a movie at a rate of 60 frames per second. This is somewhat surprising, because the underlying system clock rate is 100Hz — higher than the desired rate.

The problem stems from the relative timing of the clock interrupts and the times

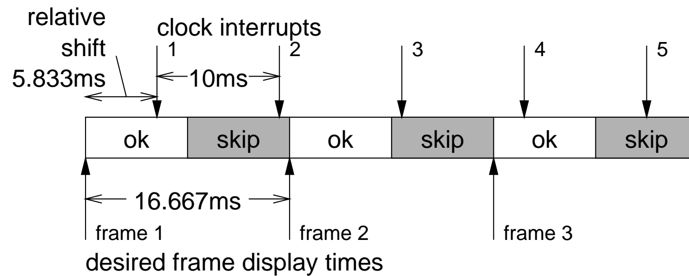


Figure 5.3: *Relationship of clock interrupts to frame display times that causes frames to be skipped. In this example the relative shift is 5.833ms, and frame 2 is skipped.*

at which frames are to be displayed. Xine operates according to two rules: it does not display a frame ahead of its time, and it skips frames that are late by more than half a frame duration. A frame will therefore be displayed only if the clock interrupt that causes Xine's timer signal to be delivered occurs in the first half of a frame's scheduled display time. In the case of 60 frames per second on a 100Hz system, the smallest common multiple of the frame duration and clock interval is 50ms. Such an interval is shown in Figure 5.3. In this example frame 2 will be skipped, because interrupt 2 is a bit too early, whereas interrupt 3 is already too late. In general, the question of whether this will indeed happen depends on the relative shift between the scheduled frame times and the clock interrupts. A simple inspection of the figure indicates that frame 1 will be skipped if the shift (between the first clock interrupt and the first frame) is in the range of $8\frac{1}{3}$–10ms, frame 2 will be skipped for shifts in the range 5–$6\frac{2}{3}$ms, and frame 3 will be skipped for shifts in the range $1\frac{2}{3}$–$3\frac{1}{3}$ms (for a total of 5ms). Assuming the initial shift is random, there is therefore a 50% chance of entering a pattern in which a third of the frames are skipped, leading to the observed frame rate of about 40 frames per second (in reality, though, this happens much less than 50% of the time, because the initial program startup tends to be synchronized with a clock tick).

To check this analysis we also tried a much more extreme case: running a movie at 50 frames per second on a 50Hz system. In this case, either all clock interrupts fall in the first half of their respective frames, and all frames are shown, or else all interrupts fall in the second half of their frames, and all are skipped. And indeed, we observed runs in which all frames were skipped and the screen remained black throughout the movie.

5.6 Clock Resolution and the Interleaving of Applications

Recall that we define the effective quantum length to be the interval from when a process is scheduled until it is descheduled for some reason. On our Linux system, the allocation for a quantum is six ticks. However, as we can see from Figures 5.2 and 5.4, our applications never even approach this limit. They are always preempted or blocked much sooner, often quite soon in their first tick. In other words, the effective quantum length is very short. This enables the system to support more than 100 quanta per second, even if the clock interrupt rate is only 100Hz, as shown in Table 5.5.

The distributions of the effective quantum length for the different applications are

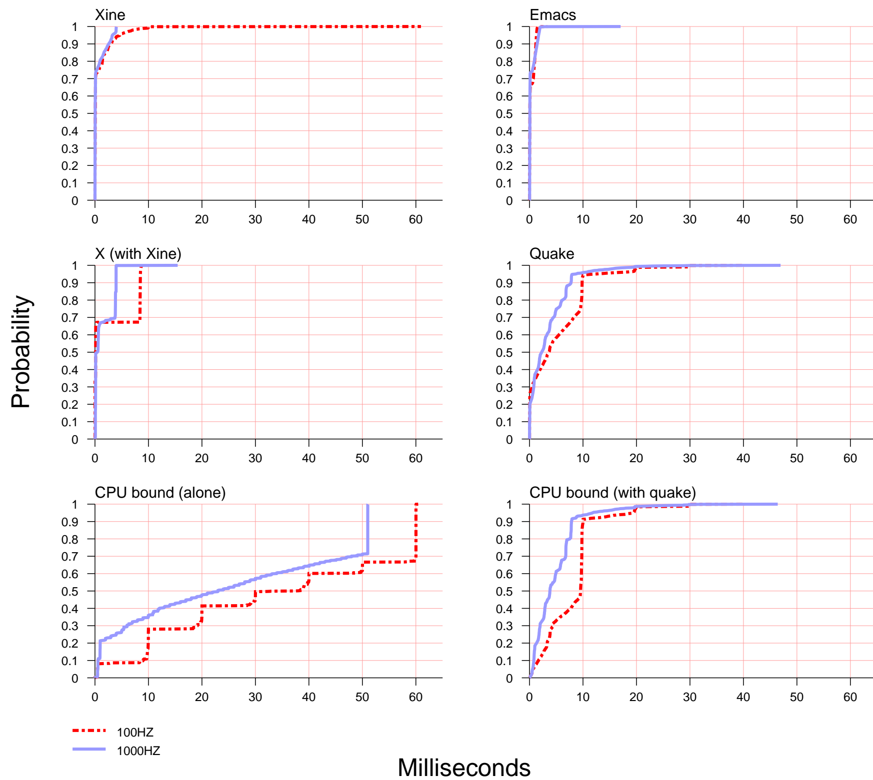


Figure 5.4: *Cumulative distribution plots of the effective quantum durations of the different applications.*

Table 5.5: *Average quanta per second achieved by each application when running in isolation.*

Application	Quanta/sec @100Hz	Quanta/sec @1000Hz
Emacs	22.36	34.60
Xine*	470.67	695.94
Quake	187.88	273.85
X Server°	71.35	148.21
CPU-bound	28.81	38.97

* Sum of all processes

° When running Xine

Table 5.6: *CPU usage distribution when running Xine.*

Application	CPU usage @100Hz	CPU usage @1000Hz
Xine	39.42%	40.42%
X Server	20.10%	20.79%
idle loop	31.46%	31.58%
other	9.02%	7.21%

shown in Figure 5.4, for 100Hz and 1000Hz systems. An interesting observation is that when running the kernel at 1000Hz the effective quanta become even shorter. This happens because the system has more opportunities to intervene and preempt a process, either because it woke up another process that has higher priority, or due to a timer alarm that has expired. However, the total CPU usage does not change significantly (Table 5.6). Thus increasing the clock rate did not change the amount of computation performed, but the way in which it is partitioned into quanta, and the granularity at which the processes are interleaved with each other.

A specific example is provided by Xine. One of the Xine processes sets a 4ms alarm, that is used to synchronize the video stream. In a 100Hz system, the alarm signal is only delivered every 10ms, because this is the size of a tick. But when using a 1000Hz clock the system can actually deliver the signals on time. As a result the maximal effective quantums of X and the other Xine processes are reduced to 4ms, because they get interrupted by the Xine process with the 4ms timer.

Likewise, the service received by CPU-bound applications is not independent of the interactive processes that accompany them. To investigate this effect, these processes were measured alone and with Quake running. When running alone, their quanta are typically indeed an integral number of ticks long — Often the allocated 50ms plus one tick (which is an additional 10ms at 100Hz, but only 1ms at 1000Hz). But when Quake is added, the quanta of the CPU-bound processes are shortened to the same range as those of Quake, and moreover, they become less predictable. This also leads to an increase in the number of quanta that are missed for billing (Table 5.4), unless the higher clock rate of 1000Hz is used.

5.7 Towards Best-Effort Support for Real-Time

To see how close a general purpose system can come to supporting real-time processes, we measured the timing delays of processes that requested timer alarms when competing with each other and with CPU-bound processes. The experiments involved processes that repeatedly request an alarm signal 500 times; each alarm is set for a certain number of milliseconds between 1 and 1000 (a second). All the processes are assigned to the (POSIX) RR scheduling class. The performance metric was the latency till these signals are delivered. There were three types of processes: The first only set alarms and did not perform any computing (denoted BLK in the table of results). The second computed for a certain fraction of the time till the next alarm; specifically, we checked computation of 1, 2, 4, and 8% of the interval till the next alarm (denoted by the percentage). The third computed continuously (denoted CONT). In addition, we used CPU-bound processes that did not set any alarms as a background load (denoted by +xCPU, where x is the number of such processes). We checked combinations of 1, 2, 4, and 8 processes of each kind. Note that for the combination of 8 processes computing for 8% of the time, this leads to an average load of 64% of the CPU capacity.

The base system used in the experiments is the original Linux, with 100Hz clock interrupt rate and scheduling quanta of 6 ticks (60ms). To see what can be achieved with current technology, we compared this with a rather aggressive alternative: a clock interrupt resolution of 20,000HZ and a quantum of 3 ticks (i.e. 150μs).

Measurements show that even at this rate, the overhead for clock interrupt processing is only 11.7%, and that for context switching 2.1%. A sample of the results is shown in Figure 5.5. As we are interested in the worst case latencies, the tails of the distributions for selected experiments are summarized in Table 5.7.

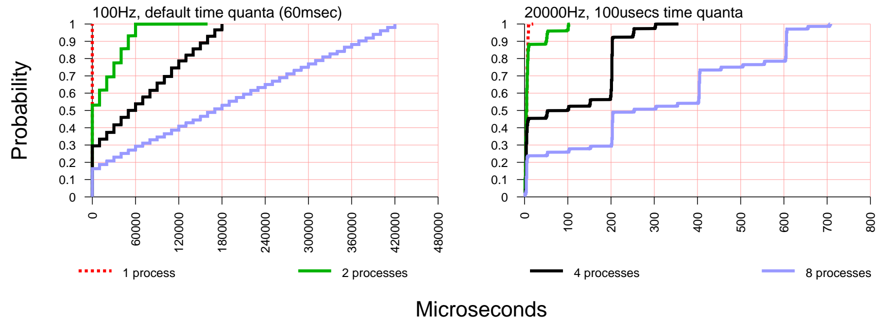


Figure 5.5: *Distributions of latencies till a timer signal is delivered, for processes that compute continuously and also set timers for random intervals of up to one second.*

Table 5.7: *Tails of distributions of latencies to deliver timer signals in different experimental settings. Table values are latencies in microseconds, for various percentiles of the distribution.*

Processes		@100Hz				@20,000Hz			
Type	Number	0.9	0.95	0.99	max	0.9	0.95	0.99	max
BLK	2	5	8	11	40	13	14	21	23
BLK	8	5	12	22	420	7	9	13	25
CONT	2	50003	60003	60004	160006	50	53	102	105
CONT	8	370014	400014	420015	740025	606	606	706	708
2%	2	6	9	9193	19153	13	15	23	837
2%	8	2910	8419	17940	32944	12	52	53	1809
8%	2	9	12431	39512	60003	14	19	53	3797
8%	8	40003	60005	130006	294291	53	53	54	37328
4%	1+2CPU	50003	50003	50004	50005	55	56	200	256
4%	1+8CPU	50003	50003	170014	280010	56	57	59	856

The graphs in Figure 5.5 are for the processes that compute continuously while setting alarms (this is the worst case, because the CPU is always busy, and, except for the case of a single process, there are always alternative processes waiting to run). Examining the results for the original 100Hz system (left of Figure 5.5), we see that a single process receives the signal within one tick, as may be expected. When more processes are present, there is also a certain chance that a process will nevertheless receive the signal within a tick (0.53, 0.30, and 0.16 for 2, 4, and 8 processes, respectively, slightly more than the probability that this process has the highest priority). But it may also have to wait until its relative priority becomes high enough. This leads to the step-like shape of the graphs, because the wait is typically an integral number of ticks. The maximal wait is a full quantum for each of the other processes; in the case of 8 competing processes, for example, the maximum is 60ms for each of seven others, for a total of 420ms.

The situation on the improved system is essentially the same, with two differences. One is that the distribution of waiting times is less uniform — a process typically has to wait a full quantum for an even number of other processes. However, the reason for waiting an *even* number of processes remains a conundrum, and is left for future work.

The other difference is that the time scale is much much shorter — the latency is almost always less than a millisecond. In other words, the high clock interrupt rate and rapid context switching allows the system to deliver timer signals in a timely manner, despite having to cycle through all competing processes. Table 5.7 shows that this is the case for all our experiments.

Note that using the higher clock rate also provides significantly improved latencies to the experiments where processes only compute for a fraction of the time till the timer event. With 100Hz even this scenario sometimes causes conflicts, despite the relatively low overall CPU utilization.

The very few long-latency events that remain are attributed to conflicts with system daemons that perform disk I/O, such as the pager. Similar effects have been noted in other systems [21]. These problems are expected to go away in the next Linux kernel, which is preemptive; they should not be an issue in other systems that are already preemptive such as Solaris.

Chapter 6

Why is the Classic, CPU Consumption Based, Process Classification No Good anymore?

As noted in chapter 3 processes are classically divided into two groups - CPU bound and I/O bound - and the operating system's scheduler is designed to identify those two groups and prioritize the processes accordingly.

In this chapter we will show why the classic distinction is no longer valid, and new metrics must be used to identify interactive process. As the saying goes: *"The Times They Are A Changin"*'[12].

6.1 Total CPU Consumption

Historically, computers were text oriented, and interactive applications were no exceptions (text editors and browsers, shell interpreters etc.). Because of this I/O with the user was very simple and did not require much CPU resources, and applications spent most of the time waiting for user input. This was in fact the behavior of an I/O bound process, so interactive processes were treated as such.

This behavior was appropriate since I/O bound processes usually have higher CPU priority, so when the user typed something, the interactive application responded promptly.

This all changed in recent years with the developments in multimedia technology. These modern interactive applications, which involve graphics and image rendering require much more CPU resources for user I/O. Such applications include movie players which require CPU resources both for decoding multimedia streams and for displaying high quality images. Other examples include graphical role playing games that require many CPU cycles to render complicated geometric models, map textures, etc. Although modern hardware uses a dedicated Graphical Processor Unit (GPU) based on the display adapter to assist with these tasks, they still require heavy assistance from the CPU.

We measured the CPU usage of our test applications (described in section 4.3). Each application was run on its own, with no interference. We started by simply mea-

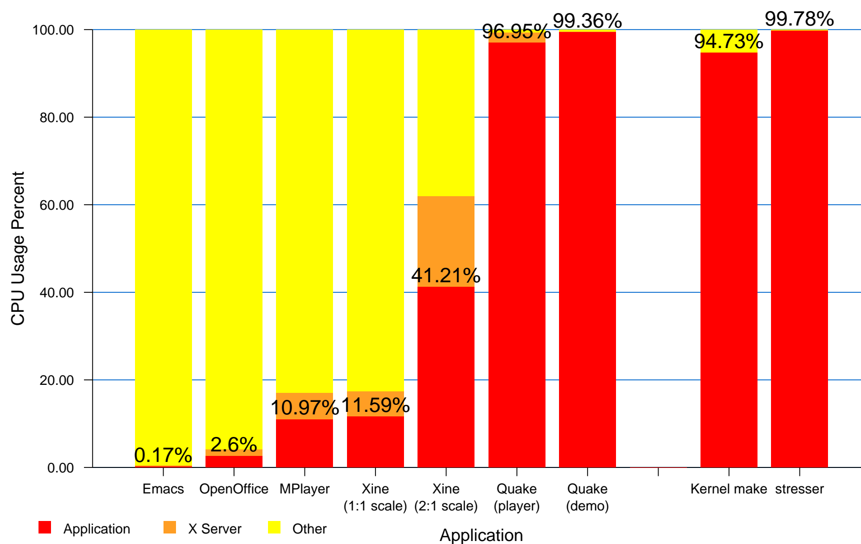


Figure 6.1: CPU Usage Percentage of Various Applications

suring the percentage of CPU consumed by the various example applications, over the time of the test. The result can be seen in figure 6.1. It is clear that modern applications consume much more CPU.

When we compare text editors, we can see that the classic *Emacs* required the CPU for 0.17% of the time measured, while the modern *OpenOffice* required 2.6%. While still very low compared to others, we can see a 15 fold increase in the CPU consumption of an inherently non-multimedia application.

With the measured movie players we see that multimedia applications require a much more substantial portion of the CPU's time. Although suspicious at first, the found that the difference in CPU portions consumed by *Xine* and *MPlayer* is a result of *Xine* playing it at double size, having to zoom each frame. Also, having the X server require over 20% of the CPU time while serving *xine* looks exaggerated, but figure 6.1 clearly shows that the ratio of 2:1 between the CPU time consumed by the movie player and the X server is kept both when playing the movie with *MPlayer* and when playing it at normal size with *Xine*. This means that playing a relatively small movie (about $\frac{1}{7.75}th$ of the screen size) requires 15% of the CPU resources, and displaying it at 2:1 zoom requires more than 60% of the CPU's time (note that 2:1 zoom means 4 times the previous size). Note that following this consumption playing a movie in full screen mode at a mere 1024x768 resolution will require more than 100% of the CPU time!

This shows us that total CPU consumption is not a good metric for differentiating between a common interactive application such as a movie player, and between a clear batch job such as kernel compilation, especially if we remember that even the zoomed movie is using only a little more than half the screen ($\frac{1}{1.94}th$ of it, to be exact).

It is not surprising that the batch applications consume almost all the CPU time when allowed to do so, but we can see *Quake* behaves quite the same - when allowed to run in demo mode, it consumption resembles that of an infinite loop - our *stresser*.

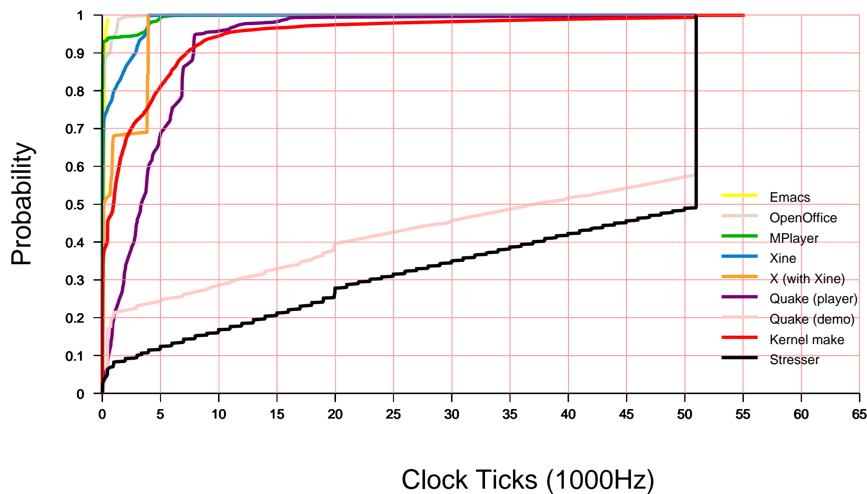


Figure 6.2: Cumulative Distribution Function (CDF) of Effective Quanta Lengths

When it accepts user input, *Quake* requires a little less of the CPU time, but this gap is actually filled with the X server serving the input to quake. This is actually a result of the adaptive nature of *Quake*. The rendering engine is designed to use whatever CPU time it can get, and adapting the achieved frame rate accordingly: the more CPU it gets, the better the frame rate it produces.

In conclusion of figure 6.1, it is clear that the distinction between modern interactive applications and well known batch jobs, based on CPU consumption is getting fuzzy at best, and even non-existent in some cases.

6.2 CPU Consumption Distribution

Schedulers do not calculate the CPU usage statistics based on total CPU consumption, but rather on the momentary consumption, so another important aspect of CPU consumption is the distribution of the length of effective quanta.

An effective quanta is the actual time consumed by a process, from the time it was given the CPU, until the time it relinquished the CPU, either voluntarily or because of preemption. We would expect the CPU-bound applications are less likely to relinquish the CPU voluntarily, thus have generally longer quanta than interactive applications which block on user I/O frequently (all this assuming the system is unloaded, as is the case in this text).

Figure 6.2 show the cumulative distribution function (CDF) of the effective quanta length for the various applications we used.

As expected we can see that the effective quanta length of the interactive applications is relatively short. Almost all the quanta of *Emacs*, *OpenOffice*, *MPlayer*, *X Server* and *Xine* are shorter than 5 clock ticks (5msec), when the maximum quantum is 51 clock ticks - less than 10% of the alloted time quantum are actually used by the process!

The kernel compilation's pattern is, surprisingly, not much different from that of the aforementioned interactive tasks. Even though the kernel compilation includes many

disk oriented tasks such as *rm*, we checked and found that there is no big difference between the distribution of **all** the processes involved, and that of the *cc1* process that performs the actual compilation. This phenomenon is attributed to the increasing gap between the CPU speed and the disk speed. Even the compiler process itself is becoming more and more dependent on disk speed for reading the input file, rather than on processor speed for processing the data.

On the other hand, we can see a striking resemblance between the *Quake* running a game demo and the *stresser* process. Because of *Quake*'s grab-every-cycle-possible policy in order to achieve an adaptive frame rate, and the fact that for the demo game it does not communicate with the X server at all (no input, and the output is done using DRI) its CPU usage pattern is that of the *stresser*'s.

When running *Quake* with a human player controlling it, its typical effective quantum length is immensely reduced because the X server keeps waking up to handle mouse input. That is the reason for the big difference in *Quake*'s two running modes in the CDF graph.

In summation, we see the difference between interactive and batch applications is getting fuzzy with regards to CPU usage pattern, just as it is diminished with regards to total CPU consumption (section 6.1).

6.3 CPU Consumption is Misguiding the Scheduler

After seeing how the difference in both CPU consumption quantity and pattern are getting smaller and smaller, let's see how this diminishing gap actually affects the scheduler distinction of interactive processes.

6.3.1 Interactive Processes Aren't Getting the CPU When They Need It

One aspect of a misguided scheduler is that it does not prioritize the processes according to their real importance. If the scheduler is scheduling the processes according to their real importance we would expect the more important processes to get hold of the CPU as soon as they need it, preempting less important processes.

We tried to evaluate the amount of time an interactive process has to wait for the CPU on average — its *dispatch latency* — and how much of the time it is runnable at all is spent waiting for the CPU.

The results can be seen in figures 6.3 and 6.4 respectively. For the multi-threaded applications — *Xine* and *OpenOffice* — the results are shown for the thread that consumed the most CPU during the test, hence was the thread most affected by the scheduler.

The average dispatch latency was calculated as the time between a process insertion into the run queue and the first time it is scheduled to run, or as the time between two consecutive events when it is scheduled to run. Figure 6.3 clearly shows that the average dispatch latency increases dramatically for interactive processes that heavily consume CPU cycles, such as *Quake* and *Xine*. Interactive applications that consume less CPU are also affected, when running such a number or *stresser* processes so their portion of the CPU time become less than the interactive one. An example of that is the *MPlayer* which normally claims about 14% of the total CPU time (figure 6.1), but when more than 6 *stresser* processes are running even it's equal (and non-preferential) share of the CPU time becomes less that what it normally needs.

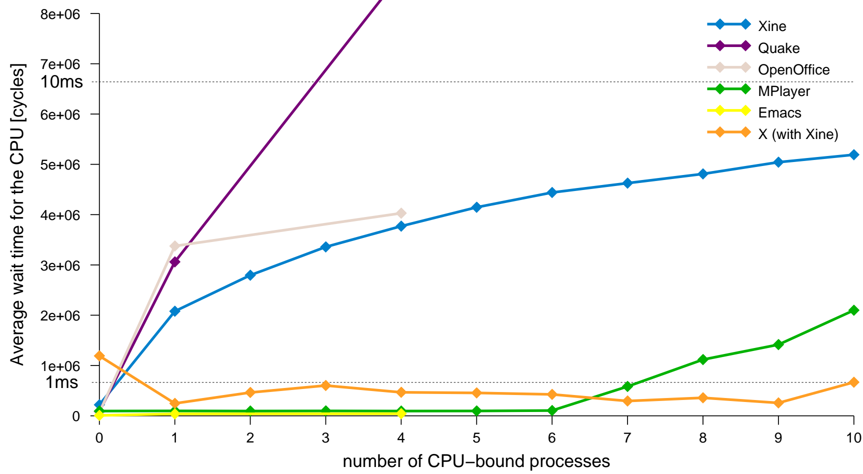


Figure 6.3: Average dispatch latency of various interactive applications using the original Linux scheduler (reference lines showing 1ms and 10ms times relative to cycles)

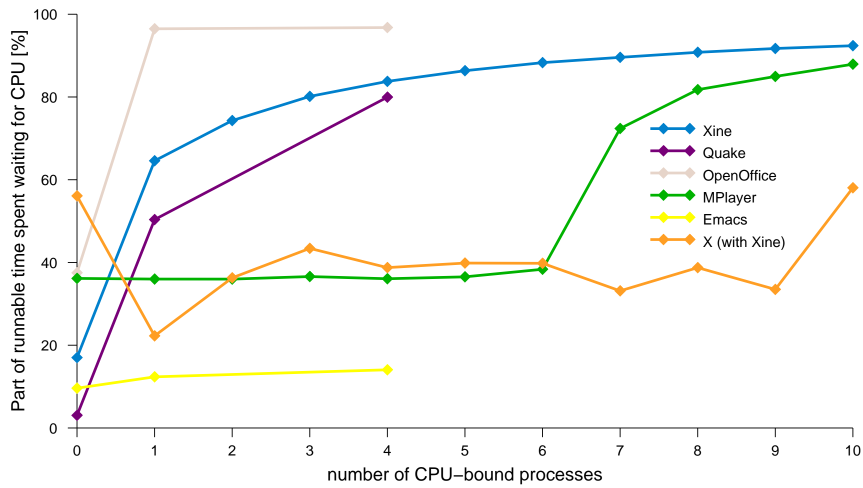


Figure 6.4: Percent of the interactive processes' runnable time spent waiting for the CPU using the original Linux scheduler

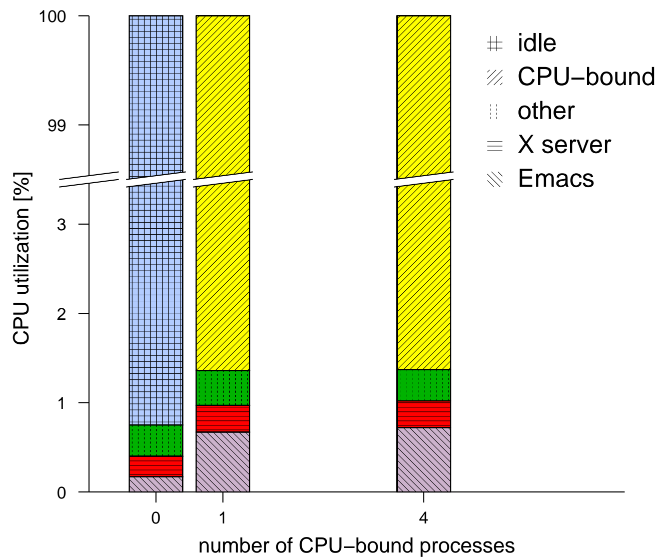


Figure 6.5: Effects of using Emacs with background batch jobs using the original Linux scheduler

An interesting effect though is the X server running alongside the *Xine* movie player. Since it normally requires about 20% of the CPU time in such workload (figure 6.1), we would expect it to be affected even with 4 *stresser* processes running, which is obviously not the case here. The reason for that is that the X server serves the *Xine* application, which is affected by the load much earlier. Since the *Xine* is so affected, it cannot produce the same amount of requests to the X server, so the CPU consumption decreases along that of the *Xine* movie player. Because of that the *Xine* is actually the CPU bottleneck here, and the X server does not suffer from the same problem.

Figure 6.4 shows another analysis of the same data. In this figure we see the percent of time a process is spent waiting out of the entire time it is runnable (as oppose to the time it is blocked). Again, we can see the CPU bound interactive application are heavily affected by the increasing load in the system, where the interactive applications that require less CPU are still favored by the CPU consumption metric and need not wait more for the CPU. Again, we see the affect on the *MPlayer* when running 6 *stresser* processes alongside it, and the same phenomenon that the X server is not affected because the *Xine* acts as the CPU bottleneck.

6.3.2 Interactive Processes Aren't Getting Enough CPU

Another aspect of a misguided scheduler is that it does not give enough CPU time to important applications that require it. In this experiment, we wanted to see how an interactive task's CPU allocation is reduced in the presence of a batch job. In three separate tests we ran *Emacs Xine* and *Quake* while adding background *stresser* processes, and recorded the CPU consumption of the three applications. When playing a movie with *Xine* we also recorded how many frames did not show because *Xine* missed their display deadlines.

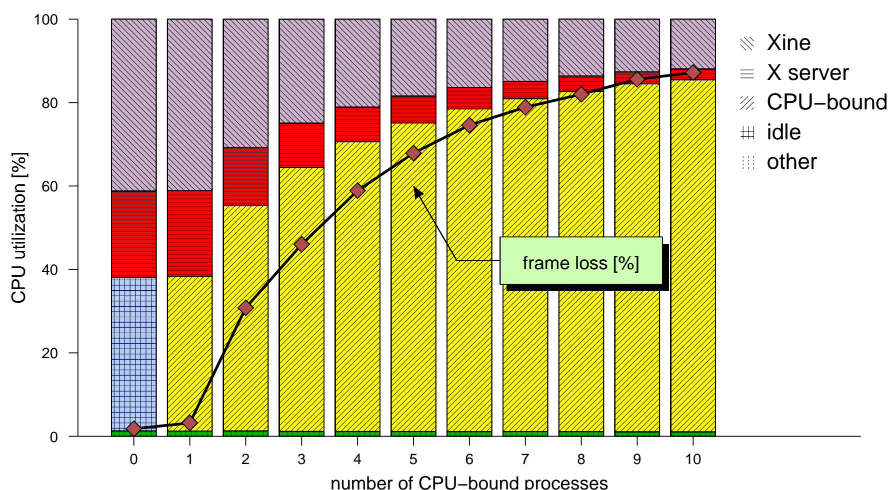


Figure 6.6: Effects of playing a movie with background batch jobs using the original Linux scheduler

Figure 6.5 shows the result for *Emacs* (note that unlike other graphs of its kind it uses a logarithmic scale because of *Emacs*'s low CPU consumption). In short, it is very clear the *Emacs* is not affected by the background load, because its CPU consumption pattern is that of a classic interactive application and very different than that of the *stresser* processes, as described in section 6.2. That is the reason the original Linux scheduler can distinguish it from the background load and favor it over the *stresser*s.

The next benchmark was the *Xine* movie player. Let us first elaborate on how *Xine* works: as mentioned earlier, *Xine* is a multi threaded application. One of its threads is in charge of decoding the MPEG frames read from the input stream (the *decoder* thread) and another thread displays the decoded frames on the screen using the X windows system (the *display* thread).

The main consumer of CPU cycles among *Xine*'s threads is the *decoder* thread. This thread keeps decoding data from the input stream, putting the decoded frames on a shared queue, along with their timing information. The *display* thread pops each decoded frame from the shared queue, and checks its scheduled display time. If the frame is less than $\frac{1}{2}$ of a frame time ($\frac{1}{50}th$ of a seconds, as the movie is encoded at 25 fps) early, it will be displayed. If it's earlier, the *display* thread will try it again in 4 msecs. If the frame missed its scheduled display time it will be discarded. This means that if the *decoder* thread is not getting enough CPU time it will not keep up with the frame rate, and frames will be lost.

The decision to use the *stresser* as the background process, rather than a kernel compilation was based on the fact that this is the most extreme example of a batch job — one that consumes as many CPU cycles it can get.

The results of this experiment are shown in figure 6.6.

When running *Xine* with no *stresser*s at all, the result is similar to that of figure 6.1 - *Xine* and the X server together require about 60% of the CPU time, leaving the rest mainly to the idle process. When running only 1 background *stresser* it takes the place of the idle process. Since *Xine* requires much less CPU than the *stresser* does, it has

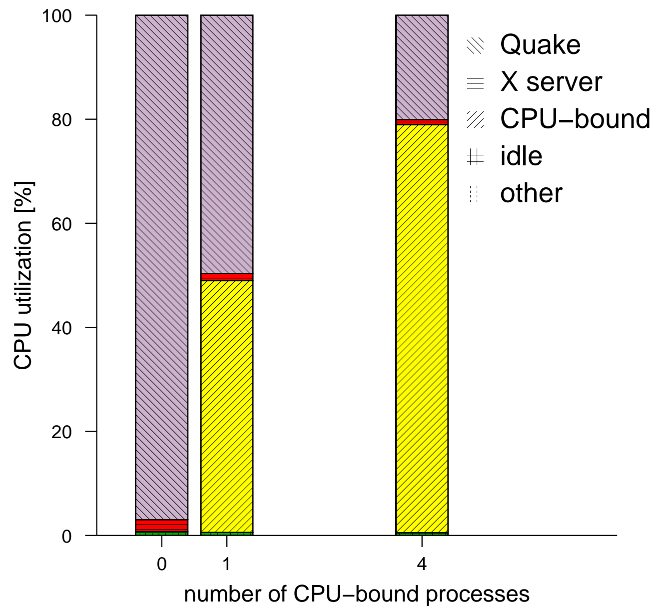


Figure 6.7: Effects of playing Quake with background batch jobs using the original Linux scheduler

a higher priority, thus is given a little more CPU. Note that there is a slight increase in the frame loss rate even when only running 1 background *stresser*. This is because even though the *stresser* does not require more CPU than the idle process, unlike the latter it has a valid priority and does not relinquish the CPU automatically when *Xine* requires it. In very rare cases the *stresser* might have a higher priority and get the CPU.

As soon as more background *stresser*s are added to the system, it is clear that the scheduler cannot clearly tell the interactive application from the batch ones, and *Xine* is getting less CPU time than needed. The obvious result is the correlation between the increase in frame loss rate and the increase in the CPU time alloted to the *stresser* tasks.

The resulting problem is that when running a modern interactive job such as an MPEG movie player, we cannot make effective use of the multi-programming feature of modern operating systems. If we want to achieve good frame rate we must not let any other application compete with the movie player over the CPU resources, thus dismissing the multi programming quality of modern operating systems.

The last benchmark was the *Quake* game. Figure 6.7 show the grim results: even under light load *Quake* needs to compete with the *stresser* over the CPU.

Quake's CPU consumption pattern was discussed in section 6.2. It was established that it is very similar to that of a CPU bound process such as the *stresser*. This means that even when running a single *stresser* the Linux scheduler regards both *Quake* and the *stresser* as two processes of the same kind — CPU bound — and divides the CPU time in similar portions between them. As a result *Quake*, which requires almost 100% of the CPU time when run alone, has to make do with approximately half of that when running a single *stresser*, and much less when running more.

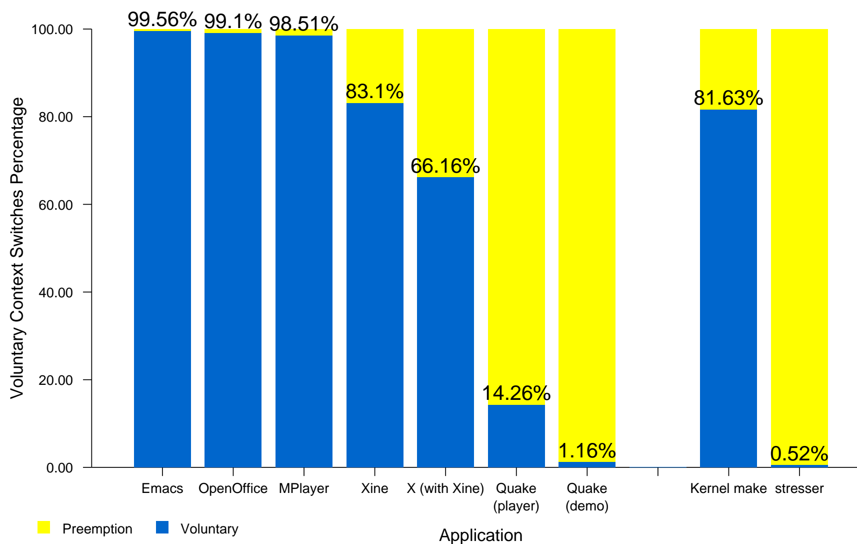


Figure 6.8: Context switch reason for various applications

These benchmarks are a clear sign that CPU consumption is no longer a viable metric to differentiate interactive tasks from batch ones, and other, more effective metrics are needed.

6.4 Another Possible CPU Consumption Based Metric (and why it doesn't work)

In the effort to find a metric that will distinguish interactive from batch jobs, we tried another unexplored metric: the effective quantum termination reason, or simply put how many of an application's quanta ended voluntarily, and how many were preempted.

The rationale behind this metric is based on another interactive workload assumption that such processes do not fully utilize their alloted quantum.

This assumption is based on the idea that the goal of an interactive process is to reply to queries from the user. Since the user expects a quick reply usually, we assume that interactive processes will not fully utilize their alloted time quantum — which is at a scale of $\frac{1}{20}$th of a second — since they do not need a long processing time. Even if an interactive process does need more than a full quantum, it is again very likely that it will relinquish the CPU after very few quanta where a CPU bound process will take as much CPU time as the operating system is willing to give.

We defined 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. The resulting visualization can be seen in figure 6.8.

Even with this metric we see that there is no clear distinction between interactive and batch jobs. Here too *Quake* behaves very similar to our *stresser*, consuming every possible cycles. When not in demo mode and having to receive input from the X server

we see an increase in the rate of voluntary context switches *Quake* experiences, but not a drastic one. The difference between the two modes of *Quake* is much less visible here than in figure 6.2. The similarity between *Xine* and the kernel compilation is ever so apparent, and this metric gives no data to differentiate between the two.

All in all, we see that this metric gives us no better understanding as to the interactive nature of the observed applications.

6.5 Indeed, the Times They Are A'Changin...

In this chapter we investigated the quality of the classic metrics used by most schedulers to differentiate between interactive and batch jobs — metrics which are all related to the CPU consumption of a task: the rate of consumption and its pattern.

We conclude that even though this metric was a sufficient one when it was introduced a few decades ago [55, 46], changes in the nature of interactive applications over the recent years have made it obsolete. Hence, it can no longer be considered a reliable one, and we must find new metrics to characterize modern interactive applications.

Chapter 7

An Alternative Metric to Identify Interactive Tasks — Information Flow Tracking

After establishing in chapter 6 that the classic CPU consumption based metric is obsolete, let us now explore an alternative metric based on a direct quantification of human computer interactions.

7.1 Following the Flow of Information

The word *interaction* implies information flow. Interactions between the user and the computer naturally implies information is flowing from the user to the computer, and vice versa.

The relationship between the user and the computer can be viewed as an alternating consumer-producer relationship, when the computer is the consumer of input information, and the producer of output information. Since the abstract computer is actually several processes, we can rank the processes based on the volume of input the consume and output they produce.

Following flow of information requires a two phase analysis: the first phase is tracking the information exchanged between the user and the computer, such as input keystrokes or output displayed on screen. This phase is insufficient because of information flow between the different running process in the computer, so a second phase is required to track the flow of information between the processes inside the computer itself. A full discussion about the need for two phases is found in section 7.3.

The two phases divide the information flow analysis between two mechanisms: the first is the *interactive device* tracking described in section 7.2, and the second mechanism is the *interprocess communication graph* described in section 7.4.

7.2 Interactive Devices

All of the information received by the computer, or produced by it, is passed through various devices: keyboard, mouse, disk, network controller, display controller, and many others. Of these, some may be classified as *interactive devices*.

An *interactive device* is a device that bridges between the user and the computer. A device the mediates information from the user to the computer is regarded as an *interactive input device*, and one the mediates information in the other direction is obviously regarded as an *interactive output device*. Examples of interactive input devices are the keyboard, mouse, joystick, graphic tablet, microphone etc. Interactive output devices are the display controller, the monitor, the sound controller and others. Some devices mediates both input and output between the user and the computer — an example of which is a force feedback joystick.

These devices hold the key to the amount and type of information exchanged between the user and the computer. By maintaining usage statistics of the interactive devices we can evaluate the level of interactiveness of the different interactive processes.

The scheduler can be made aware of this data simply by giving both kernel level and user level (system call) interfaces by which device drivers can deliver these statistics.

7.3 Is the Interactive Devices' Information Complete?

Is the information acquired from the interactive devices sufficient? Is it enough to trace the patterns that processes use interactive devices to classify them as interactive processes?

The straight answer is (unfortunately) no. A process does not have to directly interact with a device to be interactive — interactivity has a transitive quality, so a more accurate definition interactive processes could be:

> A process is interactive if it either exchanges information with an interactive device or with another interactive process

A simple example where the interactive device information will not suffice is the X windows system, present on most UNIX platforms. With this system, the X server is the application responsible for controlling the user input and user output devices, and the various processes, called X clients, draw on the screen by sending X request messages to the server and receiving input as X event messages from the server.

In such cases the interactive device information will only identify the X server as an interactive process, since it is the only process they come in contact with. The other processes which generate user output and consume user input will not be identified as interactive processes, because they do not communicate directly with an interactive device. This is a case in which identification based on *interactive devices* alone results in a false-negative. Such processes should be considered interactive since they interact with the user, albeit in an indirect manner.

To overcome this problem we must use a second mechanism which will monitor the information exchanged between the different processes, in order to find the pathways of information between the user and all relevant processes, and to identify all the processes involved in such interactions to some extent.

7.4 Interprocess Communication Graph

Monitoring exchanges of information between processes can be done by maintaining an *Interprocess Communication Graph*. This graph is composed of nodes, representing the different running processes in the system, and edges which represent information

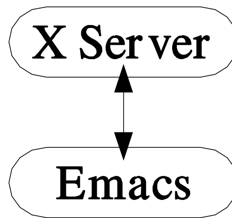


Figure 7.1: Connected component of the X server when using only the *Emacs* text editor

exchanges. A more formal definition is *let* $G = < V, E >$, *where*
$V = \{v_p |\ p\ is\ a\ running\ process'\ pid\}$ *and*
$E = \{e_{i,j} |\ information\ was\ exchanged\ between\ processes\ i\ and\ j\}$

Actually, some consideration should be taken whether the graph should be directed or undirected. Since information exchanged between processes is not only user related — getting information from various system servers, querying the *X server* — it seems simpler to make the graph undirected since the aggregate volume of the communication on a link might not represent the portion of it the is directly related to the user. However, if since interactive applications react to the user all the information they produce or consume is targeted at serving the user. The result is that the flow direction — specifically whether a process is a consumer that is dependent on a producer, or whether it is a producer itself — is crucial to the understanding of the interprocess communications graph, which then must be directed.

This graph gives us crucial information — by using simple graph algorithms to find the connected component of a node representing a process which is directly using an interactive device, we can extract the lists all the processes involved in some computation that results in user output, or all the processes involved in some computation depending on user input.

An example of a simple scenario is a user that is using the *Emacs* text editor. In this case, the X server is identified as interactive by using an interactive device. By running a simple algorithm such as Breadth First Search (BFS [9]), we can find the connected group of the X server in the interprocess communication graph. Since *Emacs* is an X windows application, and is the only application in the system, the connected group consists of only two nodes, as can be seen in figure 7.1.

By using both techniques — the interprocess communication graph and the interactive devices — we can even identify interactive processes in complex scenarios. Figure 7.2 describes a scenario where a user uses the *VI* text editor from within an X terminal emulator (xterm) [53] window on an X windows system. The flow of information in this case is as follows. When the user presses a keyboard key, the X server reads the typed character from the keyboard device, sends it as an X event message to the *xterm* process, which in turn sends it to the *VI* process. The *VI* process does the necessary processing, and to update the user it sends the output information to the terminal emulator (*xterm*), which forward it as an X request to the *X server* that sends the necessary information to the display device for drawing. The connected component in the interprocess communication graph resulting from this scenario can be seen in figure 7.3.

From the user's perspective only the *VI* process is interactive, and the user does not

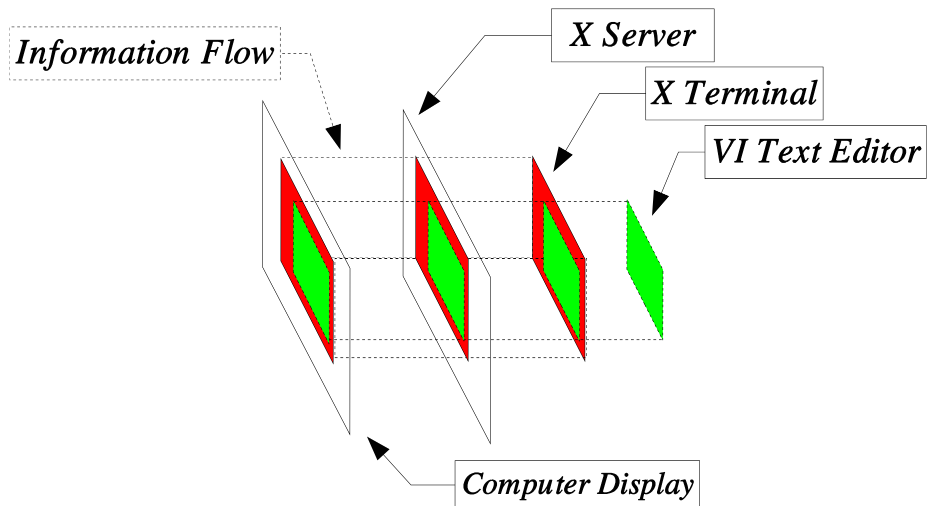


Figure 7.2: Logical layout of using the VI text editor from within an X terminal emulator

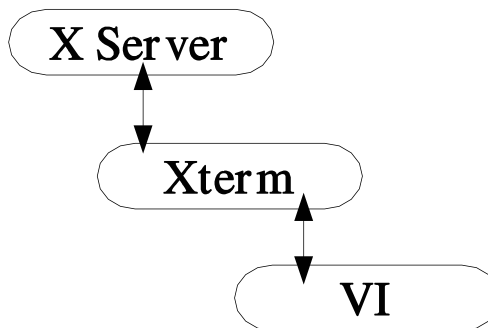


Figure 7.3: Interprocess communication graph's connected component when using the VI text editor from within an X terminal emulator

even have to be aware of the other processes involved in the action.

By following the flow of information using both the interactive devices and the interprocess communication graph the scheduler can identify all the involved processes as interactive ones: the X server is identified as interactive since it controls an interactive device, and the *xterm* and *VI* processes are identified as interactive because they compose the X server's connected component in the interprocess communication graph.

7.5 Quantifying Interactions With The User

The remaining problem with this approach is how to quantify the amount of information flowing in each direction.

Quantifying user input is relatively simple: we can simply count the number of input events delivered to each process, and rank the processes accordingly. Since the human user's attention span is measured in seconds [45], when we look at a time frame

of one second this quantification is almost binary — a human user can deliver events simultaneously to very few processes in one second.

Output to the user is a little harder to quantify: simply counting events will not work in this case since an event can be as small as printing a character to the screen, or as large as changing the background image, and it is very hard to estimate the importance of an event to the user — which event the user is really interested in, and which is a by-product.

Although it is very hard to read the user's mind, we can exploit a feature in human perception, that is a remnant of our predatorial days: human vision is more sensitive to movement [45]. Thus by quantifying the rate of changes produced by each process we have good guess which process grabbed the user attention more often. If we are assuming the user does not like to be distracted and will eliminate any source of interference — he will close a window that displays an irrelevant movie for example — anything that grabs the user's attention is important.

7.6 Scheduling with Positive Feedback — Preventing Starvation

An immediate problem that arises is that of starvation. The starvation might occur when an interactive process is favored over others thus getting more CPU time, have a better change to communicate with the user, thus getting even higher priority, eventually starving other processes waiting for the CPU. In essence, this is a positive feedback loop.

This is not a problem if the other processes are non-interactive and as such considered less important, but what happens if the starved processes are interactive? the user might want to change focus to another application but the previous focused application's priority is so high that it dominates the CPU and does not let the newly focused application gain momentum (and priority).

Such effects can be dealt with by using two mechanisms: one is CPU allocations that will guarantee that the less privileged processes will get hold of the CPU when they need it, thus giving them a change to gain momentum communicating with the user. The other mechanism is a gradual priority decay that will decrease the priority of a process not communicating with the user.

Combining these two mechanisms will guarantee that when the user changes focus between applications the newly focused application can gain momentum, while the previously focused will lose the priority it gained earlier.

The information flow based scheduler we propose, which is described in chapter 8, incorporates these two mechanisms thus preventing the starvation caused by positive feedback.

Chapter 8

A New, User Oriented Process Scheduler For Linux

In this chapter we will review all of the implementation aspects of a scheduler for interactive tasks based on the information flow tracking concept described in chapter 7.

These aspects include the implementation of *interactive device tracking* (section 8.1), of the *interprocess communication graph* (section 8.2) and of the actual scheduler and the quantification of a process' interactiveness (section 8.3).

8.1 Maintaining the Interactive Device Statistics

As discussed in section 7.2 there is a plethora of possible interactive devices, ranging from keyboards to sound cards to full sensor suits. To examine the feasibility of basing scheduling decisions on the information flow tracking methodology, we have decided to implement it only for the most mainstream devices — the keyboard, mouse and monitor. We decided to leave other common devices, such as the sound device for further research, because it is not required for the current purpose of feasibility testing.

The data is maintained in a per-process data structure, so that each process has its own interactive devices ratings in the form of "changes per second":

- The input rating is simply an estimated average of input events the process receives per second, and there is no distinction between keyboard input and mouse input.
- Output rating is also based on the rate of changes, but is normalized to the screen size, so it is an estimate of the fraction of screen area the process changes every second.

The input and output rates cannot simply be the actual changes during the current second (as this would require an oracle...), but rather an estimate based on the last few seconds. This is achieved by separating the data structure into two parts: *current changes* and *average*.

Whenever an interactive device associates a change (either an input event or a screen change) with a process, the amount of change is accumulated in the *current changes* section of the data structure. Once a second the scheduler adds the new data

to the previous calculated average using an exponential decay mechanism:

$$new\ average = \frac{old\ average + current\ changes}{2} \tag{8.1}$$

Equation 8.1 assures us that a process interactive device rating is equally based on the last second and previous seconds, so momentary changes of user focus will not affect the process, but longer changes will decrease the process rating — a process will maintain some of its peek rating for up to $\log peek$ seconds.

Choosing a second granularity is an attempt to achieve an equilibrium between the rate of human perception [45] and the amount of processing power needed to compute a recent average for every process.

Processes' statistics are updated by the various devices using a well defined interface, presented in appendix A. Part of this interface is also exported to user level by adding non-standard parameters to the standard POSIX *sched_setparam* system call [17, 42].

8.1.1 Acquiring The Interactive Device Statistics — Using the X Server as a Meta-Device

Acquiring information from the interactive devices is simply a matter of altering the appropriate subsystems to report the association of events to processes.

In UNIX however, these subsystems do not actually reside in the kernel. The UNIX kernel does contain the keyboard and mouse drivers, but UNIX uses the X Windows System [59, 53] to multiplex input and output between the user and the various applications. The processes are referred to as clients, and are connected to the *X Server*, which is the only[1] application that receives user input from the kernel and writes to the display using kernel mechanism. In this sense, the kernel part of the keyboard and mouse subsystems is degenerate and the actual association of user I/O's with X clients (processes).

This design enables us to log processes' user I/O statistics by simply hacking the *X Server* itself. The *X Server* can simply log the necessary statistics for every client connected to it and communicate the statistics to the kernel once a second (by adding another timer to the X server's own timer mechanism), using the system call interface mentioned above.

The changes to the *X Server* are described in the following subsections.

8.1.2 Modifying the X Server Data Structures

X Clients connect to the X server using either TCP for remote connection or UNIX domain sockets for local connections. The server maintains some data for each connection, which is regarded as a *client record*. We added our own data to this record, data which includes the number of input events sent to the client and the fraction of the screen changed by this client since the last time the X server updated the kernel. Also, the peer process pid was saved for local connections.

X normally does not normally know (nor care about) the pid of a client, or even if the client is local or not. One of the design goals of the X Windows System was to

[1] Actually, most UNIX flavors provide one or more virtual terminals which offer the possibility of user I/O without using the X Windows System. This form of I/O however, is rarely used for routine interactive workload hence not relevant to our discussion.

separate the interface implementation from the interface management [59], to give an abstraction of a user terminal, either local or remote. This design has some exceptions though, such as the Direct Rendering Infrastructure [37] which allows local clients to access the graphics hardware.

By modifying the communication layer in the X server we could associate a client pid with a client connection for any client connected using UNIX domain sockets [50]. This was done by a non-standard socket option implemented in Linux [26].

8.1.3 Monitoring the User Input Using the X Server

The X server reads input events from the input device files, and dispatches them to the waiting clients. We had to hook into this mechanism so we can log all the input events send to clients.

The X protocol permits (almost) any client to request input events occurring in any windows (the only exception is when a window's creator disables this option on window creation). This means that an input event can be dispatched to multiple clients.

Implementing the input logging mechanism was quite forward, since the X server already has a mechanism for hooking various events that is used by X server modules. It simply maintains lists of callbacks, one of which is called whenever and input event is sent to a client.

Our modification simply included adding a callback to the input event callback list that will log the input event sent to the client in the client record data structure.

The current implementation does not distinct between keyboard and mouse events, and regards all input events as equal. A legitimate question is whether this approach is valid, or maybe the various input devices, and even different events emanating from a single device should be prioritized differently. This issue is left for further research (see chapter 10).

8.1.4 Monitoring the User Output Using the X Server

The X server accepts graphic requests from the various connected clients and executes them. Our goal was to estimate the fraction of the screen affected by each request, and accumulate it in the per-client record (for local clients only).

Since the X protocol defines a reduced set of graphical operations available for clients, estimating the fraction of the screen affected by every available operation is a very feasible task. The list of client request that might affect the screen is displayed in Table 8.1.

We added functions to calculate the fraction of the screen affected by each of the listed requests. Some of the calculations were very straightforward, for example, calculating the size of a rectangle. However, some of the graphical requests require complex calculations to get the exact fraction of the screen affected by them. Since we did not want to add a substantial overhead to the X server, we only estimated the fraction of the screen changed by those complex requests.

Examples of such estimates are the calculations for text drawing requests, arc drawing requests and even when drawing a simple diagonal line:

- 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

Table 8.1: X protocol graphical requests [44]

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 the given 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 types)
PutImage	Draw a bitmap
PolyText8	Draw a 1-byte character string
PolyText16	Draw a 2-byte character string
ImageText8	Draw a 1-byte character string after filling the character's background
ImageText16	Draw a 2-byte character string after filling the character's background

figure 8.1. Although in this case the inaccuracy is in the range of 1-2 pixels only, it might be bigger for other lines.

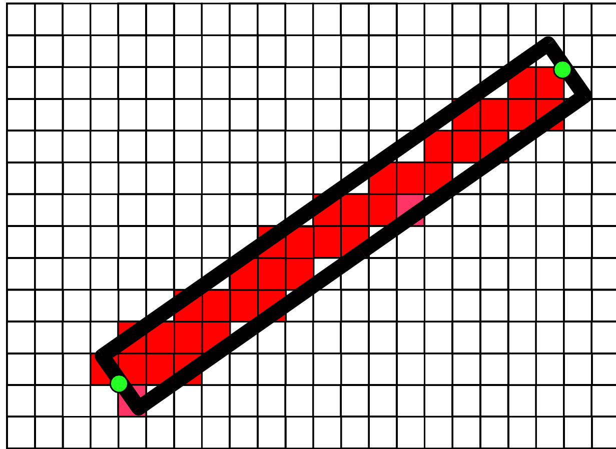
- Figure 8.2 shows the calculation of the area used by a character. When drawing only the character (without the background, using a PolyText8/16 X request) the estimate of the area drawn is the sum of the bounding boxes of all the characters used, whereas the actual area used is smaller.

- An arc is complex primitive, so the estimate is even less accurate, as seen in figure 8.3. The bounding box of the complete circle/ellipse is multiplied by the fraction of a complete circle the arc's angle uses.

Since we are only interested in the fraction of the screen affected, and not in the nature of the change, it is quite obvious that some of the request use the same estimation code. For example, clearing a rectangular area on the screen and drawing a rectangular area is essentially the same operation, using different colors, so we can use the same calculation.

After estimating the area of the screen used by the drawing, the area must be clipped to the viewable region of the screen. An application might be drawing on a hidden of partially hidden window, so the estimated region must be clipped to the viewable region. The clipping action is very important, since the information on what is actually viewable is the best hint on what really interests the user — what does the user think is important enough to allocate a portion of the screen to.

Again, the clipping cannot always be accurate. In cases where the estimated drawn region is an axis aligned rectangle, the X server's own clipping mechanism can be used, and the resulting clipped region is the estimate we eventually accumulate in the drawing client's record.



The inaccuracy when drawing a diagonal line between points (x_1, y_1) and (x_2, y_2) is apparent: requested line's boundaries are shown in black, projected over the pixel grid, with the actual pixels used in red.

Figure 8.1: Estimation of a diagonal line area

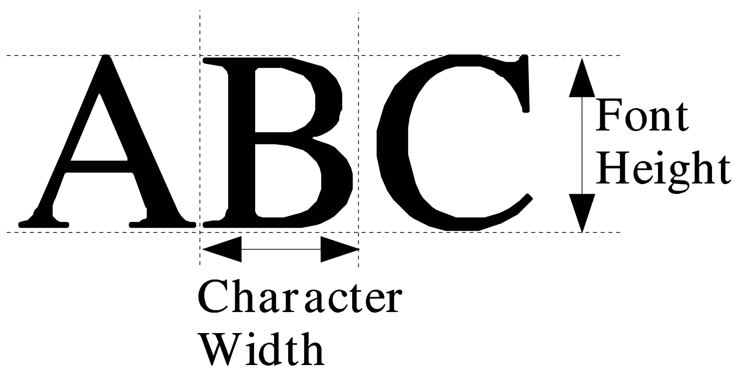
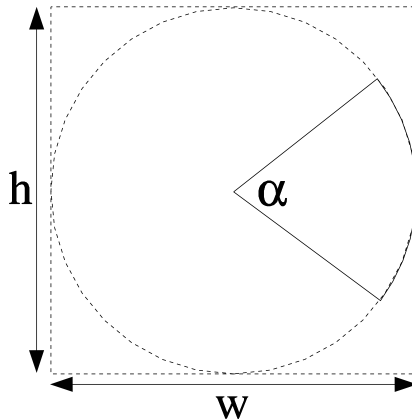


Figure 8.2: Estimation of text area



$$estimated\ arc\ area = w \cdot h \cdot \frac{\alpha}{2\pi}$$

Figure 8.3: Estimation of arc area (PieSlice mode)

When the estimated region is not an axis aligned rectangle, as is the case with arcs (figure 8.3), we calculate what fraction of the drawing's bounding box is viewable, and multiply the earlier estimated region by this fraction. Statistically, the clipping estimate should converge to depict the accurate portion of the area estimate actually drawn.

An exception to this mechanism is the Direct Rendering Infrastructure (DRI [37]), which interacts directly with the graphics controller, circumventing this entire mechanism thus not being tacked by it. DRI is used by the OpenGL library [23], commonly used by graphical software such as the *Quake III Arena* role playing game.

For this reason measurements of *Quake* do not produce any output priority, but we will later see (section 8.3) that since its input events are still proxied through the X server it is still identified as an interactive process.

By modifying the *OpenGL* library to notify the kernel of the area drawn by a process it is possible to produce output statistics for applications using DRI, but we did not implement it since it is not crucial for this feasibility test. For that reason all measurements of *Quake* when using our proposed scheduler do not include the demo mode, since in this mode there is no input and *Quake* will not be identified as interactive.

8.2 Maintaining the Interprocess Communication Graph Information

The IPC graph is a directed graph and is maintained inside the processes — every process is a node in the graph, and the node data is kept inside a per process data structure.

This data structure mainly consists on the incoming and outgoing edges. The edges are kept in two arrays. To avoid dynamic memory allocations inside the kernel — which causes memory fragmentation and cannot be done in Linux when holding a lock — the edges' arrays are fixed sized, containing up to 12 edges. If a new edge needs to

be allocated when the array is full, the least weighted edge is recycled.

The interprocess communication graph is actually an interprocess dependency graph. Whenever a dependency event is identified (just what is a dependency event is described in section 8.2.1) the weight of the corresponding edge is incremented. If the corresponding edge does not exist, one is allocated.

Once a second all the edges are decayed — all the weights are divided by 2, and the edges whose weight becomes 0 are freed. This decay mechanism assures us that graph information will represent the interprocess activity of the recent few seconds, and not of the entire machine's uptime history.

8.2.1 Identifying Interprocess Dependency

At any time the Linux kernel is running either in process context, executing code on behalf of a process (system call or even the process itself running in user-mode) or in interrupt mode executing interrupt or tasklet code.

Whenever a process is sending a message or even releasing a shared resource, another process, either a receiving process or a process waiting for the just released shared resources is tested for being in the run queue. This scenario is considered to represent a dependency between the two processes.

The test if a process is on the run queue is always done using the kernel function *try_to_wake_up* [54], which adds the process to the run queue if not already on it.

Every time the *try_to_wake_up* function is called when the kernel is running in process context we increment the weight of the edge from the tested process to the running process, since it is considered that the running process has done something the tested process waited for, even if the tested process was runnable.

This is indeed a liberal approach to interprocess dependency, that extends the formal definition of interprocess communications. However, it is in place since we are trying to encompass even implicit dependencies between processes, and not only the explicit ones.

Nonetheless, this approach is not perfect. Several other common interprocess communication primitives are not identified by this mechanism. Primitives such as explicit shared memory or shared files require different identification mechanisms, but we felt such mechanisms are not necessary for our exploratory system, and left their implementation for future work (chapter 10).

8.2.2 Handling of Multi-Threading/Shared Memory

We did address one aspect of shared memory, which is implicit shared memory, or simply put — threads.

The multi-threaded programming paradigm is becoming very common, and new processors even support it in hardware level [28]. As such, we could not ignore multi-threading when designing our scheduler, and even used a multi-threaded application as a benchmark (*Xine Movie Player*, section 4.3).

Multi-threaded applications are handled by representing them in the interprocess dependency graph as hyper-nodes. The kernel keeps track of all threads of a specific application, they are all represented in the graph as a single node and the communication with the user is accumulated for the entire thread group collectively, with no regards which of the threads actually participated in it.

Any dependency between two threads of the same application is ignored since all the edges are represented as one node in the graph. By nature, most communication

between threads is done using shared memory so they are all considered equal. Other more complex mechanisms can be designed to rate the different threads among themselves, but our design highlight is its simplicity.

This approach assures that all threads will have the same priority, and the same quantum length (section 8.3).

8.3 Putting it all Together

After implementing the interprocess dependency graph, and hacking the X server to deliver interactive information, it is now time to modify the scheduler to use all the interactive information gathered.

The proposed scheduler is designed to address the common timings of human perception. The scheduling decisions at any point in time are based on statistics gathered up until no more than a second before, with a larger consideration in the recent few seconds to maintain temporal locality. Using a time resolution of seconds is appropriate to human perception rate, as discussed in [45].

In this section we will describe the design of the new scheduler, and its handling of interactive processes.

8.3.1 The Stacked Scheduler

The original Linux scheduler is not modular. It is hard coded to handle the three POSIX scheduling classes — FIFO, Round Robin (RR) and Other (implementation dependent) [43] — so adding a special handling for interactive processes proved to be too tricky. Also, when prioritizing the interactive processes we must pay attention not to starve the kernel threads, otherwise the system might become unstable.

For this reason we had to rewrite the scheduler and modify its data structures. Our design was inspired by that of the Solaris 8 scheduler [29], and described as a *Stacked Scheduler*.

As mentioned earlier, POSIX divides processes between three scheduling classes. The *Stacked Scheduler* maintains and enhances the notion of scheduling classes, but enables programmers to design and incorporate new scheduling classes into the kernel.

Whenever the scheduler needs to choose a process to run, it traverses the stack of scheduling classes, in the order of the classes' importance, "asking" each class to pick a process. Since the classes are traversed in the order of their importance, whenever a class is found which has a runnable process, that process is chosen.

A process can migrate between the different classes, using a modified version of the *sched_setparam* system call [42]. When a process is created, it inherits its scheduling class from its parent (an exception to this are kernel threads, which have no parent, and are specifically born into the KTHREAD class which is discussed later).

The scheduler comes with 6 default scheduling classes, listed in table 8.2. These default classes consist of the three POSIX classes — FIFO, RR and OTHER — and three new ones — KTHREAD, INTERACTIVE and IDLE.

The FIFO and RR classes prioritize their processes according to the FIFO and RR models accordingly. Note however that this breaks the POSIX scheduling class model a little, since in our *Stacked Scheduler* any FIFO process *always* has a higher priority than any RR process. The POSIX model only states that the FIFO and RR classes are always chosen before the OTHER class, but does not define a strict order among themselves.

Table 8.2: Default scheduling classes, in their stacking order

Class Name	Description
FIFO	POSIX First-In-First-Out
RR	POSIX Round-Robin
KTHREAD	Kernel Threads
INTERACTIVE	Interactive Processes
OTHER	Linux Original Scheduler — All regular processes
IDLE	Idle Processes

The KTHREAD, OTHER and IDLE classes prioritize processes using the original Linux scheduling algorithm (section 3.3) — each class with its own processes.

Another class — the INTERACTIVE one — is designed to support the scheduling of interactive processes whose identification is described in section 8.3.2. A full description of the scheduling algorithm the class implements is found in section 8.3.3.

In conclusion, this design has several major advantages over the original Linux scheduler:

- *Modular* - this design lets programmers design their own scheduling classes and add them to the stack, with each class prioritizing its process with no external intervention.

- *Efficient* - the scheduling decision does not need to traverse a list of all the running processes, as is the case in the original Linux scheduler, but only until it finds a runnable process. The nature of this traversal is adaptive — if a high priority process is present it will be found faster since its class will be traversed earlier, thus giving adaptive efficiency based on the process priority (the new development Linux kernel has a new and more efficient scheduler [33], but not as modular as ours).

- *Dynamic* - new scheduling classes can be introduced at runtime, according to the user's needs (although this feature is not fully implemented yet).

- *Multiple Idle Processes* - the *Stacked Scheduler* support long term, low priority computations be expanding the notion of an idle process into a scheduling class. This way the user can migrate a long term process to the IDLE class, without it competing with more important processes over the CPU.

8.3.2 Identifying the Interactive Processes and Prioritizing Them

As discussed in section 7.4, the connected component of the X server in the interprocess dependency graph is the group of processes that communicate with the X server, both directly and indirectly. Also, a single stream of user I/O might depend on a group of processes rather than only one process. We'll refer to such a group as an interactive application, with each member of that group being an interactive process.

A process' priority is calculated *once a second* according to the following equation:

$$process\ priority = distance + shortcut \tag{8.2}$$

with *distance* being the topological distance between the process and the X server in the interprocess dependency graph, and *shortcut* being per-application prioritizing number set according to amount of user I/O generated and received by that process. As we'll see next, the *distance* is an intra-application priority, and the *shortcut* is an inter-application priority.

For distance calculation, all the edges are given a length of 2, regardless of the amount of interprocess interactions they represent. This simplistic approach was found sufficient for our purposes (see results in section 9).

All the processes which directly communicate with the X server and have a interactive priority set by the server (processes regarded as *explicitly interactive*) are sorted according to both their input and output rating. These processes then accumulate their *shortcut*s according to the following rules:

1. All the processes that send output to the X server (and thus to the user) receive a path shortcut of half an edge.

2. All the processes that receive input from the X server (and thus from the user) receive a path shortcut of one edge.

3. The 3 processes that have the most effect on the display get half an edge shortcut.

4. The two processes that receive the most input from the user receive a shortcut worth two edges.

Since we regard user input as the best indication that the user is interested in a process, the *shortcut* value for input is higher than that of output.

After the *shortcut*s has been calculated for the explicitly interactive processes, a Breadth First Search (BFS, [9]) algorithm is run on the interprocess dependency graph to topologically sort the various processes, find the shortest path from the X server to each of them and thus find the X server's connected component. The topological distance is calculated for each process.

While traversing the graph, the BFS algorithm propagates the *shortcut* value of each explicitly interactive process to all the processes whose shortest path from the X server passes through that explicitly interactive process.

This way, aside from obtaining the X server connected component, the processes are prioritized in groups, with each group being considered an interactive application. Among themselves, the processes inside an interactive application are prioritized according to their topological distance from the X server, with the *shortcut* value being an inter-application priority.

For example, let's consider a shell prompt which receives more user input than any other process. That shell communicate with an *X terminal*, which in turn communicates directly with the X server. Since all the data between the user and that shell passes through the *X terminal*, this terminal with receive a *shortcut* value fitting the process that receives the most input — 2 for any input and 4 for best input — totaling at 6.

The shortest path to the shell passes through the *X terminal*, so the *X terminal*'s *shortcut* value is propagated to the shell, and the entire interactive application (*X terminal* + shell) receives the *shortcut* value, with the priority inside the application rated according to the topological distance.

Finally, the X server gets a special *shortcut* value that will give it the highest interactive priority, since this is the user information junction. We could have similarly treated the X server as a kernel thread (which will give it a higher priority than any user process) but decided it should be treated as part of the *INTERACTIVE* process class.

Note that this algorithm is general enough that even a crucial process like the window manager can be treated just like any other interactive process — since it is directly connected to the X server, and it monitors almost all input events, it has a very high priority.

Although this algorithm seems time consuming, our measurements shows that is requires ~ 2 milliseconds, and it is only run once a second so its overhead is a mere 0.2% of the CPU time.

8.3.3 Allocating CPU Time and Choosing the Next Interactive Process To Run

Although many changes we've made to the scheduler, the concept of computation epoch, introduced with the original Linux scheduler (section 3.3), is still maintained.

At the beginning of an epoch each process is alloted a CPU time quantum to consume. Whenever a scheduling decision is to be made, the scheduler chooses the process with the biggest remaining time quantum. Once all processes have consumed their aloted quantum, the epoch is considered over and a new allocation is made. The only difference from the original Linux scheduler is way a time quantum is allocated for each process.

The time quanta are allocated in the following manner: we define a maximum time quantum for an interactive application (which currently stands at ~ 200 clock ticks, with a clock tick every 1 millisecond). The highest priority interactive processes (according to the priority calculated using the augmented BFS algorithm described in section 8.3.2) receive the maximum time quantum. The processes with the second best priority get half that quantum. This goes on with the processes at each priority level getting half as much time as those in the higher level (unless the processes in the higher level got a quantum which less than 4 clock ticks, in which case the quantum is set to 2 clock ticks). This exponential decay continues until all the processes got a new time quantum.

Using the notion of a computation epoch gives two major benefits: it prevents starvation caused by the positive feedback mechanism[2] thus solving the problem discussed in section 7.6. Also, it does not add any overhead to the process selection mechanism, relative to the original Linux scheduler thus limits the more time consuming graph algorithm to run only once a second.

8.4 Conclusions

In this chapter we introduced a novel approach to scheduling interactive application based on interprocess and process-user information flow.

We have proposed and implemented a new scheduler design, one which captures the complexity and variety of existing workloads, and includes a new scheduling algorithm that specifically targets the interactive workload, most common in workstations and desktop.

[2]Actually, some of our measurements show a problem with positive feedback when an interactive process hogs the CPU, but we suspect this is caused by an illusive bug in the interprocess dependency graph implementation

Chapter 9

Now Let's See If It Works...

To see if the proposed scheduler works, we ran a variety of interactive applications with background CPU bound processes, as described in section 4.3. Our goal was to check if our proposed scheduler solves the problem arising from using the CPU consumption as a scheduling metric, problems which are shown in section 6.3.

The questions to be asked are whether an interactive process gets the CPU when it need it (discussed in section 9.1) and whether an interactive process get enough CPU time, even though its CPU consumption pattern is similar to a CPU bound pattern (discussed in section 9.2)

9.1 Do Interactive Processes Get the CPU When They Need It?

The main purpose of a scheduler is to allocate the CPU to the important processes when they need. As described in section 6.3.1, we measured how long does an interactive process has to wait in the run queue until it is given the CPU to run on — its dispatch latency.

Figure 9.1 shows the average time the various interactive processes wait on the run queue until they get the CPU, when using our proposed scheduler. It is clear that the background load has practically no effect on the dispatch latency of the interactive processes, which means the proposed scheduler identifies them correctly as interactive and prioritize them accordingly, as oppose to the original Linux scheduler dispatch latency described in figure 6.3.

The other aspect we discussed in section 6.3.1 is the fraction of its runnable time a process waits for the CPU, and we have seen that this percentile increases dramatically when using the original Linux scheduler and running background CPU bound applications (figure 6.4).

However, figure 9.2 clearly shows that using our proposed scheduler prevents this increase altogether. It shows that the fraction of runnable time spent waiting for the CPU hardly increases even under heavy CPU load. This means that the proposed scheduler identifies the interactive processes correctly, and maintains the correct identification over time.

Figures 9.1 and 9.2 establish that the proposed scheduler's priority scheme is correct — it identifies the interactive processes, and the scheduler epoch model works. This observation leads us to the next question: we now know the proposed scheduler

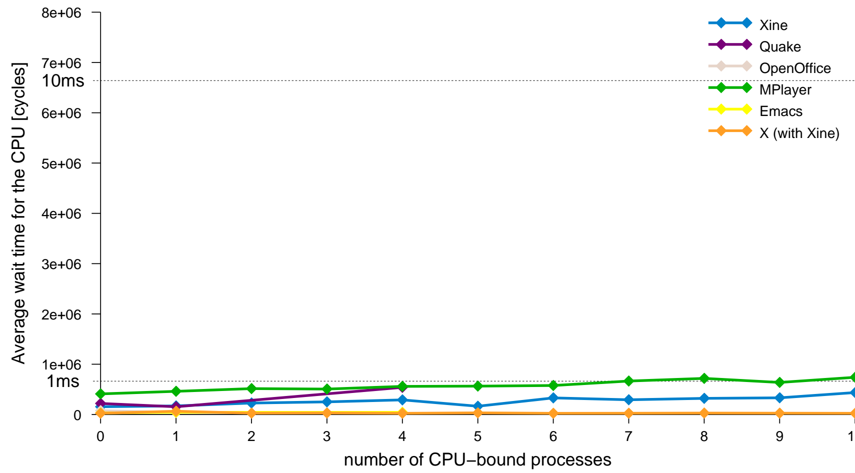


Figure 9.1: Average dispatch latency of various interactive applications using our new information flow based scheduler (reference lines showing 1ms and 10ms times relative to cycles). Compare with figure 6.3.

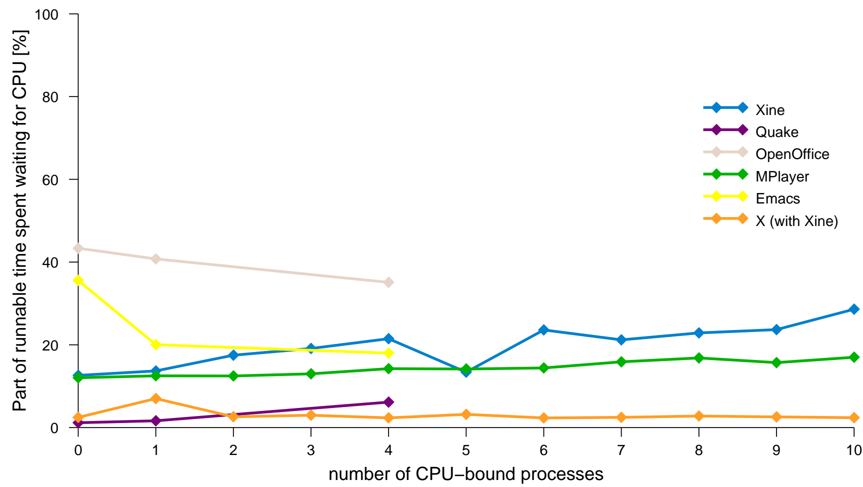


Figure 9.2: Percent of the interactive processes' runnable time spent waiting for the CPU using our new information flow based scheduler. Compare with figure 6.4.

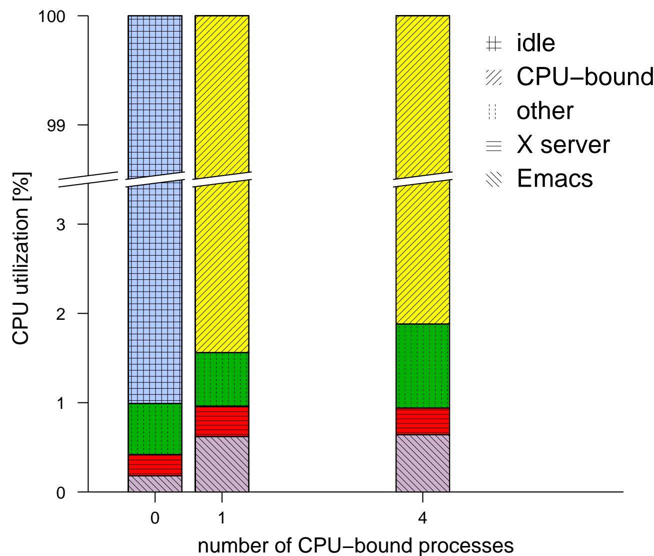


Figure 9.3: Effects of typing text with background batch jobs using our new information flow based scheduler. Compare with figure 6.5.

gives the CPU to the interactive processes when they need it, but does it give them *enough* CPU time?

9.2 Do Interactive Processes Get As Much CPU Time As They Need?

First, let us verify that we did not mess things up, and things that worked correctly with the original Linux scheduler, still do with the proposed scheduler. Chapter 6, and especially section 6.3.2 and figure 6.5 show that CPU consumption based schedulers, such as the original Linux scheduler, still favor classic interactive applications such as the *Emacs* text editor, and prioritize them over non-interactive applications. We wanted to verify that *Emacs* is getting enough CPU time when using information flow as the scheduling metric. We ran *Emacs* with a variable number of *stresser* processes as described in section 4.3 and measured the fraction of the CPU time *Emacs* received. The result is shown in figure 9.3 (again, note that unlike other graphs of its kind it uses a logarithmic scale because of *Emacs*'s low CPU consumption). We can clearly see that *Emacs* is getting approximately the same percentage of CPU time no matter how many *stresser* processes are running in the background.

The next step is to check for improvements. Section 6.3.2 shows that the *Xine* movie player is heavily affected by background CPU load, and this effect is graphically depicted in figure 6.6.

We ran the same benchmark using the information flow based scheduler, and the results can be seen in figure 9.4.

It is clear that our proposed scheduler identifies the *Xine* movie player as interactive and prioritized all its threads accordingly, so it is not affected by background CPU

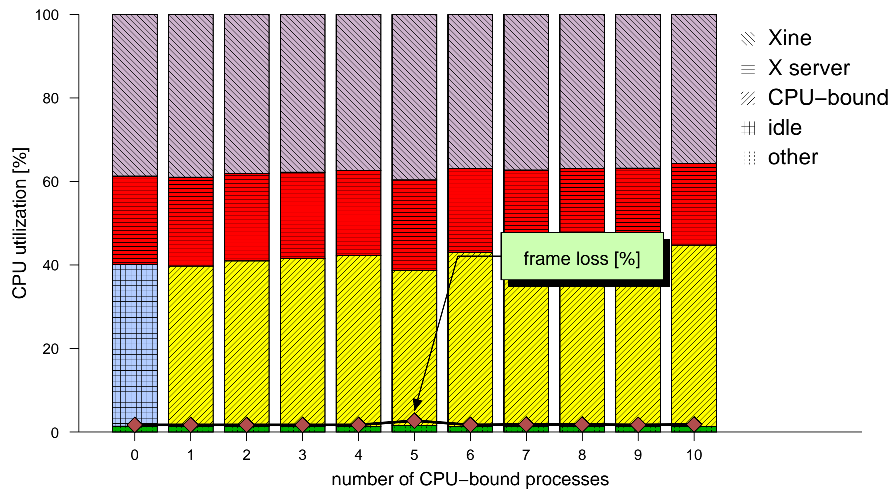


Figure 9.4: Effects of playing a movie with background batch jobs using our new information flow based scheduler. Compare with figure 6.6.

load, and the percentage of lost frames remains approximately constant as *stresser*s are added.

The final step was to test the *Quake* role playing game. Section 6.2 shows that *Quake* CPU consumption pattern is very similar to that of a CPU *stresser*, a similarity that is further established when observing how *Quake* has to compete with background load (figure 6.7).

However, figure 9.5 which sums the last benchmarks clearly shows that the proposed, information flow based scheduler easily characterizes *Quake* as an interactive application and favors it over the background load simulated by the *stresser* processes.

9.3 Conclusions

Using a variety of benchmarks involving sample interactive applications that represent most of the common type of interactive applications, we have given an alternate scheduling theory. We have shown that the information flow tracking methodology discussed in chapter 7 is a viable and feasible alternative to the 30 year old general purpose process scheduling theory governing most general purpose operating systems.

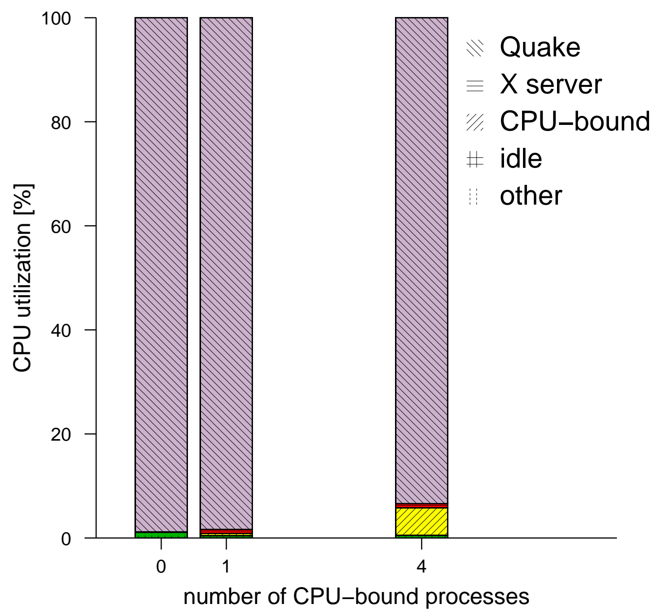


Figure 9.5: Effects of playing Quake with background batch jobs using our new information flow based scheduler. Compare with figure 6.7.

Chapter 10

Further Research

10.1 Extending the Research Presented

The most immediate extension to this research is to explore other interprocess communication mechanism and monitor them for interprocess dependencies. Such mechanisms include shared memory, shared files and many others.

Other extensions include further analysis of the different types of user I/O and their workloads. For example, we assumed all input events are equal and treated them as such, but this may not be true. For example, a keyboard event might be more indicative of the user's interests than a mouse event. Another example is differentiating events according to their rate — a slow rate of text output might indicate that the user is reading it, while a fast rate can be beyond user's perception rate thus indicate that the user is not evaluating it at real time.

Exploring these paths involves cognitive research and can give much understanding of human computer interactions.

Another very interesting path is extending the interactive event monitoring to include *remote interactiveness*, as described in chapter 2. This path might add new possibilities to the fields of grid computing and distributed operating systems. This path is also interesting since it involves several security challenges of saving global interprocess dependency kernel information in non-secure networks.

10.2 Various Application of Information Flow Tracking

Application of tracking *information flow* in other areas of system research might be very interesting. Such applications include:

- *Global Scheduling* — Using information flow we evaluated the importance of a process to the user, and used this priority to manage CPU time. However, CPU time is not the only resource managed by modern operating systems. Other resources such as network bandwidth, disk bandwidth and memory are also managed and using our evaluation of the user's interests to allocate other resources to the running processes can make the computer more user centered, thus friendlier from the user's perspective.

Implicit Gang Scheduling — Using interprocess dependencies can help identify gangs of processes, thus making gang scheduling implicit rather than explicit. A similar idea has been already presented in [15].

Appendix A

API for the Interprocess Statistics Scheduler

- The interprocess communications graph must be protected by a global lock so changes to the data structures will not cause inconsistencies.

  $static\ inline\ void\ sstats_global_lock(void);$
  $static\ inline\ void\ sstats_global_unlock(void);$

- Initialize the IPC scheduler. Called during the boot process.

  $int\ sstats_init(void);$

- Graph management: process p is added to the run queue.

  $static\ inline\ int\ sstats_runqueue_add(structtask_struct\ *p);$

- Graph management: process p called exit, and we have to remove it from the graph.

  THIS FUNCTION MUST BE CALLED WITH THE tasklist_lock LOCKED (because it calls find_task_by_pid). (since the only place we call it from is unhash_process which write-locks this lock it's ok).

  $void\ sstats_process_release(struct\ task_struct\ *p);$

- Graph management: $child$ was just forked from $parent$, using $clone_flags$ flags.

  $void\ sstats_process_fork(struct\ task_struct\ *parent,$
  $struct\ task_struct\ *child,$
  $unsigned\ long\ clone_flags);$

- Graph management: process p just called exec.

  $void\ sstats_process_exec(struct\ task_struct\ *p);$

- This function is used to update the X server perceived user priorities for process p. It is called from the modified $sched_setparam$ system call during the periodical update from the X server.

  $static\ inline\ int\ sstats_interactive_update(struct\ task_struct\ *p,$
  $int\ input_priority,$
  $int\ output_priority,$
  $int\ part_of_output);$

- The basic goal: a process priority. These functions return a process's interactive priority (and whether it is considered interactive at all). The interactive priority is an integral value in the range $0\ldots255$.

  $static\ inline\ int\ sstats_process_is_interactive(struct\ task_struct\ *p);$
  $static\ inline\ long\ sstats_process_interactive_priority(struct\ task_struct\ *p);$

  The following function does the same but does not lock the graph itself, so it MUST BE CALLED WITH THE SSTATS LOCKED!!!

  $static\ inline\ int\ __sstats_process_is_interactive(struct\ task_struct\ *p);$

Bibliography

[1] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[3] Scott A. Banachowski and Scott A. Brandt. The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes. In *Multimedia Computing and Networking (MMCN)*, January 2002.

[4] Michael Beck, Harald Bohme, Mirkok Dziadzka, Ulich Kunitz a nd Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, 2nd edition, 1998.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001. ISBN: 0596000022.

[6] John Bruno, Eran Gabber, Banu Özden, and Abraham Silberschatz. The eclipse operating system: Providing quality of service via reservation domains. In *USENIX Technical Conference*, pages 235–246, 1998.

[7] John L. Bruno, Eran Gabber, Banu Özden, and Abraham Silberschatz. Move-to-Rear List Scheduling: A New Scheduling Algorithm for Providing QoS Guarantees. In *ACM Multimedia*, pages 63–73, 1997.

[8] George Candea and Michael B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Second USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, August 1998. USENIX.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, second edition, September 2001. SBN: 0262032937.

[10] R. T. Dimpsey and R. K. Iyer. Modeling and measuring multiprogramming and system overheads on a shared memory multiprocessor: Case study. *Journal of Parallel and Distributed Computing*, 12(4):402–414, Aug 1991.

[11] Kenneth J. Duda and David R. Cheriton. Borrowed virtual time (BVT) scheduling: supporting latency sensitive threads in a general purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 261–276, May 1999.

[12] Bob Dylan. The Times They Are A Changin'. Columbia Records, Febuary 1964.

[13] Günter Bartsch et al. Xine Movie Player. http://xine.sourceforge.net/. version 0.5.1.

[14] Free Software Foundation. GNU Emacs. http://www.gnu.org/software/emacs/emacs.html. version 20.7.1.

[15] Eitan Frachtenberg. Flexible coscheduling. Master's thesis, School of Computer Science and Engineering, Hebrew University of Jerusalem, December 2001.

[16] FreeBSD.org. The freebsd 4.4 kernel. http://www.freebsd.org.

[17] Bill O. Gallmeister. *Posix. 4: Programming for the Real World*. O'Reilly & Associates, January 1995. ISBN: 1565920740.

[18] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[19] Moving Pictures Experts Group. *Short MPEG-1 description*. International Organisation for Standardisation, June 1996. http://mpeg.telecomitalialab.com/standards/mpeg-1/mpeg-1.htm.

[20] Xingang Guo. *Predictable CPU Bandwidth Management Framework for Next-generation Operating Systems*. PhD thesis, The University of Texas at Austin, 2000.

[21] Joe Gwinn. Some measurements of timeline gaps in VAX/VMS. *Operating Systems Review*, 28(2):92–96, Apr 1994.

[22] Denis Howe. The Free On-line Dictionary of Computing. www.foldoc.org.

[23] Silicon Graphics Inc. OpenGL. http://www.opengl.org/.

[24] Sun Microsystems Inc. OpenOffice. http://www.openoffice.org. version 1.0.1.

[25] Lawrence J. Kenah and Simon F. Bate. *VAX/VMS Internals and Data Structures*. Digital Press, 1984.

[26] Andi Kleen. *Linux Manual Page: UNIX Domain Sockets*.

[27] John Lions. *Lions' Commentary on UNIX 6th Edition*. Annabooks, 1996.

[28] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, 6(1), February 2002.

[29] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture*. Sun Microsystems Press/A Prentice Hall Title, 2001. ISBN: 0130224960.

[30] Marshal Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1997. ISBN: 0201549794.

[31] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[32] Sun Microsystems. Solaris 8 kernel sources. http://www.sun.com.

[33] Ingo Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler. www.kernel.org.

[34] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the Forth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1993.

[35] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.

[36] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Conf.*, Jun 1990.

[37] Brian Paul. Introduction to the Direct Rendering Infrastructure. http://dri.sourceforge.net/doc/DRIintro.html, August 2000.

[38] Various Programmers. MPlayer. http://www.mplayerhq.hu/. version 0.90pre1.1.

[39] Melissa A. Rau and Evgenia Smirni. Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads. In *MASCOTS Conference, Maryland, USA*, pages 252–261, October 1999.

[40] RedHat, Inc. RedHat Linux Distribution — Version 7.0. www.redhat.com.

[41] Ronny Ronen, Avi Mendelson, Konrad Lai, Shih-Lien Lu, Fred Pollack, and John P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, Mar 2001.

[42] *Linux Manual Page: sched_setparam System Call.*

[43] *Linux Manual Page: sched_setscheduler System Call.*

[44] Robert W. Scheifler. *X Protocol Reference Manual*, volume 0. O'Reilly & Associates, July 1989.

[45] Ben Shneiderman. *Designing the User Interface*. Addison-Wesley, third edition, 1998. ISBN: 0201694972.

[46] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison Wesley, fifth edition, 1998.

[47] ID Software. Quake III Arena. http://www.idsoftware.com.

[48] David A. Solomon. *Inside Windows NT*. Microsoft Press, second edition, 1998.

[49] David A. Solomon and Mark E. Russinovich. *Inside Windows 2000*. Microsoft Press, third edition, 2000.

[50] W. Richard Stevens. *UNIX Network Programming*, volume 1. Networking APIs: Sockets and XTI. Prentice Hall PTR, second edition, 1998. ISBN: 013490012X.

[51] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE Real-Time Systems Symposium*, December 1996.

[52] Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the unix kernel: The battle for control. In *Proc. Summer USENIX Technical Conference*, pages 185–197, June 1987.

[53] The XFree86 Project Inc. XFree86 , an open-source implementation of the X Window System. http://xfree86.org/. version 4.1.0.

[54] Linux Trovalds, Alan Cox, and many others. The Linux Kernel Sources, Version 2.4.8. http://www.kernel.org.

[55] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice-Hall, first edition, 1995. ISBN: 0131019082.

[56] Jorge E. Vieira and Dilma M. Silva. The SMART Scheduling for Linux.

[57] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Symposium on Operating System Design and Implementation (OSDI)*, November 1994.

[58] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[59] X Consortium. X Windows System. www.X.org.

[60] Yanyong Zhang and Anand Sivasubramaniam. Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters. In *ACM Symposium on Parallel Algorithms and Architectures(SPAA)*, pages 209–218, July 2001.