Integrated Quality of Service Management

David Ingram

Jesus College
University of Cambridge

A dissertation submitted for the degree of Doctor of Philosophy, January 2000

Revised and updated, August 2000

Abstract

General purpose operating systems need integrated real time scheduling.

It has recently become standard practice to run soft real time applications, such as multimedia and games programs, on general purpose desktop systems. The operating systems in use on these platforms employ scheduling algorithms derived from the traditional multi-user timesharing world, which are unsuitable for real time purposes. The scheduler is viewed as a “black box” whose behaviour depends in an unpredictable way on the entire system load. Many hard real time systems use static priorities instead, but these are not suitable for the dynamic task sets encountered in a general purpose computing environment.

A large number of prototype systems with improved real time schedulers have been created in the past. Unfortunately, designers of these systems placed constraints on the operating system structure which are incompatible with ubiquitous monolithic kernels, client-server architectures, existing standards and applications. This has prevented their adoption in a production desktop system. Furthermore, little regard has been given to making real time capabilities convenient to use. An integrated user interface and automated quality of service management are necessary in a desktop environment.

This dissertation makes three main contributions which combine to overcome the difficulties just described.

(a) Scheduling

We present a conventionally structured, general purpose platform which provides effective soft real time scheduling. Binary compatibility with a large application software base has been preserved by extending an existing operating system; the modified platform is called *Linux-SRT.*

A basic design premise is that scheduling is separated from functionality, which allows quality of service to be associated with unmodified Linux applications and permits centralised control. We have developed a *named reserve* abstraction to share quality of service between threads and take advantage of application-specific knowledge where appropriate. Reserves and processes are handled by the same kernel scheduler, without a separate real time mode or hierarchy of schedulers.

(b) Servers and IPC

Techniques for scheduling real time servers accurately are discussed, and a solution presented. This allows server processes to utilise their clients' quality of service without restructuring. Multi-threaded servers are handled by allocating a single reserve to a set of threads. Single-threaded servers, including the X window system, are addressed with a retrospective accounting mechanism.

The implementation makes use of a novel IPC mechanism. This distributes kernel events to servers, so they can synchronise with other activities, and is also used to integrate with window management and desktop control functions. In addition we have improved the normal socket abstraction by adding authentication and resource propagation, so that priority inheritance can take place between real time processes.

(c) Quality of Service Management

Linux-SRT applies quality of service parameters automatically to real time applications. These are described using a new kind of *dual policy* specification. Tools and user interface components which allow ordinary users to interact with the quality of service management system are demonstrated. These are tightly integrated with window management functions, avoiding cumbersome control programs.

We also evaluate methods for determining scheduling parameter values without user intervention. Processor time slices can be determined by empirical adaptation. Where statistical multiplexing is used, overrun probabilities are considered explicitly. Simple forms of mode-change support such as automatic idle state detection are also possible. Access control to real time service classes has been defined in a flexible capability-based manner, so programs do not need administrator rights to use them. Limits prevent overcommitting of resources, starvation of lower priority processes and denial of service.

The resulting system has the following properties which have not been achieved before: soft real time scheduling on a desktop operating system, binary application compatibility, real time support for single-threaded servers, and a simple user interface for quality of service management.

Preface

This dissertation is the result of my own work and is not the outcome of work done in collaboration, except where stated otherwise in the text.

I hereby declare that no part of this dissertation has been or is currently being submitted for any other degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words.

Acknowledgements

I am grateful to my supervisor, Jean Bacon, and to Ken Moody for encouragement, advice and valuable feedback.

The Nemesis OS group at Cambridge and Mike Jones at Microsoft Research both assisted greatly by providing access to the internals of their own real time systems.

Many people helped me bounce ideas back and forth, including Neil Stratford, Dickon Reed and Andrew McNeil. Dominic Camus helped refine the IPC design. Daniel Andor deserves special thanks for many enthusiastic discussions and for testing Linux-SRT. Martin Keegan started it all with his deep understanding of UNIX, and also corrected many of my errors.

Finally I should like to thank the developers of the Linux kernel, the XFree86 project, and the community at large for creating a stable, interoperable system—and releasing the source code for every component so that I could reuse and adapt it.

This work was supported by the UK EPSRC.

Contents

List of Figures

List of Tables

Glossary

ACL	–	Access Control List
API	–	Application Programming Interface
BE	–	Best Effort (not real time)
CLI	–	Command Line Interface
CORBA	–	Common Object Request Broker Architecture
CSCW	–	Computer Supported Cooperative Working
DE	–	Desktop Environment
EDF	–	Earliest Deadline First
FCFS	–	First Come First Served
FD	–	File Descriptor
FIFO	–	First In First Out (also a POSIX static priority scheduling class)
FPS	–	Frames Per Second
GID	–	Group Identifier
GUI	–	Graphical User Interface
HRT	–	Hard Real Time
IDE	–	Integrated Development Environment
IDLE	–	Scheduling class for background processing
IPC	–	Inter Process Communication
MMU	–	Memory Management Unit
MPEG	–	Moving Picture Expert Group (compressed video format)
MP3	–	MPEG part 3 (compressed audio format)
OTHER	–	POSIX scheduling class for BE applications
PID	–	Process Identifier
PIP	–	Priority Inheritance Protocol
POSIX	–	Portable Operating System Interface for Computer Environments (IEEE standard)
POSIX.1	–	Main OS API standard
POSIX.1b	–	Real time extensions
POSIX.1c	–	Threads standard
POSIX.1e	–	Security enhancements
PSS	–	Proportional Share Scheduling
QOS	–	Quality Of Service
RID	–	Reserve Identifier

RM	–	Rate Monotonic
RPC	–	Remote Procedure Call
RR	–	Round Robin (also a POSIX static priority scheduling class)
RT	–	Real Time
RTOS	–	RT Operating System
SFQ	–	Start-time Fair Queueing
SMP	–	Symmetric Multi Processing
SRT	–	Soft Real Time
TID	–	Thread Identifier
UI	–	User Interface
UID	–	User Identifier
VM	–	Virtual Memory
VOD	–	Video On Demand
WM	–	Window Manager
X11	–	X Window System, version 11 (The Open Group)

Chapter 1

Introduction

1.1 Real Time Systems

This dissertation considers the application of Real Time (RT) scheduling to a desktop computing environment. We shall start by defining some useful terminology.

A RT program is one which must meet temporal as well as logical correctness criteria if it is to fulfill its intended function. By contrast, best-effort (BE) applications are those which produce useful results even when run slowly—although of course speed is usually still *desirable*.

The purpose of a RT system is to extend the virtual machine presented by the OS to each application, so that it provides consistent *performance* as well as functionality.

The most common metrics for performance are throughput and response time. A real time system must offer *predictable* performance, so we are more concerned about the statistical distribution of response times. A system which is slow can be optimised or upgraded, but an unpredictable one cannot host RT applications at all.

For example, a text editor which always takes 0.1 seconds to respond to keystrokes is preferable to one which takes 0.01 seconds 90% of the time, but 0.5 seconds otherwise. The latter is almost twice as fast on average, but will behave unevenly in use.

1.1.1 Hard and Soft Real Time

A further distinction is drawn between Hard and Soft RT systems. A hard RT application is one which will fail completely if any timing guarantee is not met. A soft RT application experiences only temporary failure if a short-term scheduling deadline is missed; it is not necessary to abort the application.

For example, a continuous speech recognition program is SRT, an air traffic control system is HRT, and a wordprocessor is BE. We shall be concerned specifically with SRT systems.

A more detailed analysis requires the statement of a "threat model" (by analogy with this notion in security), in which the situations that may cause the system to renege on guarantees are clearly identified (since any real system has some chance of failure).

Threats to accurate scheduling which can be allowed for include uncooperative applications, scheduler overheads, and client-server interactions. However we will not consider certain other threats such as excessive numbers of interrupts, or critical sections. This is justified because these are either very unlikely in practice, or intrinsic limits due to a particular application's design.

1.1.2 Scenario

In the past, low end platforms did not require a RTOS. Future consumer operating systems will need much better RT support, because users want to run RT applications now and most of them have access only to mass market personal computers. This trend has been driven by the increasing power of small machines and by the demand for multimedia applications in particular.

It isn't practical or cost effective to use a dedicated machine, with a special purpose operating system to run increasingly common real time applications. Applications such as video-conferencing are in fact likely to be *more* useful on standard desktops than on dedicated RT systems.

The personal computer environment is significantly different from a multi-user system or the high-end workstations which have been used for real time research previously. Here are some of the contrasting characteristics:

- Shrink-wrapped software
- No specialised hardware
- Real time and conventional applications on the same machine
- Relatively few busy applications running at once
- Primarily single-user
- Non-technical end-users and no system administrator

These factors have a number of consequences. The use of shrink-wrapped software means that programs cannot easily be customized by each site, so real time support must not require changes to the application. We also cannot rely on special hardware, such as "multimedia" disk drives, framebuffer ownership tags or user-safe devices [Barham96].

We require predictable behaviour whilst handling a mixture of simultaneous RT and BE processes. For example, during a video-conference it may be necessary to make concurrent use of desktop accessories such as note-taking, spreadsheets or document exchange. Although there may be many light background processes there will usually be a small number of very busy tasks, which means that statistical multiplexing techniques are not likely to work well.

Although we cannot guarantee there will be a single user, this will often be the case, which makes the problem of access control to real time resources somewhat easier. However, the lack of expert assistance means that configuration must be automatic for the most part.

1.1.3 Objectives

The primary design criteria we wish to satisfy are as follows:

- Focus on the desktop environment
- Predictability
- Run real applications, not benchmarks
- Provide backwards compatibility and integration with existing systems
- The system must not leave the choice of parameters to applications or users
- High level tools are required
- Effective user interface design
- Simplicity

In addition, it should be possible to provide fundamental support for network installations, multiple users, and SMP architectures. These are essential features of any production system, though they are not considered in detail by this dissertation.

The predictability objective includes a desire to minimize the amount of surprise experienced by end-users under conditions of resource contention, adaptation, and multiple tasks.

Note that we are considering a *complete* system, not simply the low level scheduling aspects or QOS management alone. The user interface is an important aspect in its own right; it also turned out to highlight the need for new APIs and IPC primitives, so that RT information could be shared between system components.

The need for appropriate tools is also more pressing than it would at first appear. For example, Microsoft Windows NT has a powerful ACL model for file permissions, but suffers from a lack of convenient tools for viewing this information for collections of files and directories. As a result it is harder to manage file permissions correctly than with the simpler UNIX model. Another example is the POSIX support for RT scheduling under Linux. This is potentially useful but is defined as a system call interface, with access restricted to root. It is not often used because there is no GUI, nor even a command line tool to activate it.

1.1.4 Application domains

Many desktop applications can benefit from RT scheduling, as shown by the list below. In Chapter 5 we identify the particular requirements of each application, and show how to set scheduling parameters in order to meet these demands.

- Video and animation playback (DVD for example)
- Voice and video mail (recording and playback)
- Internet phones and video-conferencing
- Streaming internet radio, video on demand (news clips, movies)
- TV tuner display
- Sound synthesis and MIDI
- Audio servers (mixing and processing)
- Real Time Games and VR (consistent speed provides a realistic and reproducible environment, though adaptive algorithms help achieve this)
- Java applets (a temporal sandbox)
- Emulators (allows the use of legacy RT software)
- User interfaces (guaranteed responsiveness for mouse movement, window management, voice control and abort functions)
- Debugging (pausing applications, testing at reduced speeds, benchmarking).

1.2 Scheduling

In this section we review some of the history of real time and conventional scheduling. The advantages and limitations of current methods are highlighted.

Note: detailed discussion of the Linux scheduler is deferred until Chapter 2.

1.2.1 UNIX scheduling

The conventional UNIX scheduler is based on multilevel feedback queues [Leffler89]. Dynamic priorities depend on exponentially weighted processor usage. Threads running in kernel mode have priority over all those in user mode.

It achieves a combination of good interactive response times for I/O bound tasks, and high throughput. This works well for best effort tasks and multiuser systems.

The scheduler functions as a “black box” with only one adjustable parameter—the *nice* value, which is added to a task’s dynamic priority when it is recalculated. This is rarely satisfactory for prioritizing RT applications because its effect depends in a complex way upon the entire system load.

Solaris [Khanna92] is an example of a more modern UNIX variant, which also has some RT features. Interestingly the kernel, although monolithic, is fully preemptive. This attractive structure is more often used to achieve high degrees of SMP scalability than for RT purposes, however. The scheduler provides static priorities for RT tasks as well as the usual dynamic ones. A System V `priocntl` interface is used to control this.

1.2.2 Windows NT

The Windows NT [Solomon98] scheduler supports 16 static priority levels for real time tasks, and 16 dynamic levels. Scheduling within each level is round-robin, with a time quantum adjustable for each thread.

As with Solaris, user mode threads may preempt those in the kernel (threads maintain their ordinary priority level in kernel mode).

There is no facility to lower the priority of threads based on CPU usage. However, the dynamic priority is increased for a task when it unblocks (the level depends on what it was waiting for). Such elevated tasks are moved back down a level every time quantum until they block or reach the base level.

The scheduler also obeys hints from the GUI in an attempt to deliver better apparent performance. The thread which owns the "foreground" window has its time quantum tripled, and any thread with a dialog box open has a greatly increased priority.

Finally, the scheduler provides a fail-safe mechanism in case of priority inversion. Every second it scans for processes which have been ready to run for more than 3 seconds, and gives them the maximum dynamic priority for 2 time quanta. They then immediately revert to their normal priority. This method is obviously too coarse to provide acceptable response times for a desktop system.

1.2.3 Weighted scheduling

Several different scheduling algorithms are designed to assign fair proportions of the available processor time to each process or user. These are easily generalised to deal with arbitrary weighting factors.

The Fair share scheduler [Kay98] is an extension of dynamic priorities in UNIX to respect group weightings. The formula for calculating priorities is summarised in Figure 1.1.

Lottery scheduling [Waldspurger94] is a probabilistic method which operates on individual process weights. Processes are assigned a number of 'lottery tickets' in proportion to their weights. Random numbers are used to select which process wins the lottery and will be scheduled next.

Stride scheduling [Waldspurger95] is similar to the lottery algorithm but makes the selection process deterministic. This has better statistical properties and thus guarantees fair allocation of CPU at shorter time scales.

$$p = b + C + (G/w) \qquad \begin{aligned} C &= (c + C')/2 \\ G &= (g + G')/2 \end{aligned}$$

$$\begin{aligned} p &= \text{process dynamic priority} \\ b &= \text{process base priority} \\ w &= \text{group weight} \\ \{c, g\} &= \text{\{process, group\} CPU usage this time interval} \\ \{C, G\} &= \text{exponentially weighted \{process, group\} CPU usage} \\ \{C', G'\} &= \text{previous values of \{C, G\}} \end{aligned}$$

Figure 1.1: Fair share scheduling

Proportional Share Scheduling (PSS) achieves similar results to Stride scheduling, although it is not derived from a probabilistic method. PSS has been implemented for FreeBSD [Jeffay96]. It is viewed as an approximation to pure processor sharing (in which all processes would receive their fair share during any time interval, however small).

A *virtual time* concept is used to account for the discrepancies between the progress experienced by each task and ideal, continuous time. The scheduling algorithm is Earliest Virtual Deadline. A bound can be derived for the deviations caused by the discrete time quanta, so the algorithm can be used to guarantee response times for RT applications.

The weights used in PSS have been applied to the scheduling of operating system activities as well [Jeffay98]. This solves the livelock problem caused by incoming network packets, without the need for dedicated kernel threads.

Start-time Fair Queueing [Goyal96] is another variant of PSS. It has been implemented for Solaris and is also used by QLinux [Goyal96].

A disadvantage of any method based on pure weights is that different quantities of time are assigned to a process depending on the other tasks in the system. This is not usually the required behaviour for RT applications.

1.2.4 Commercial and embedded real time systems

Most industrial real time systems make use of static priority schedulers. These offer very predictable behaviour. In some cases dynamic priorities are supported as well; in this case the static priorities are used for real time tasks and are higher than all the dynamic ones.

These operating systems are tuned for fast response times by reducing context-switch delays and interrupt latencies. They almost all have microkernel architectures to ensure a high degree of preemptivity, and priority inheritance is used to improve client-server interactions. However there is no support for quality of service guarantees.

In cases where there is more than one RT task, it is difficult to achieve the desired quality of service for all of them with static priorities. Moreover, there are now so many RT applications in common use that it is not sufficient to restrict to a single RT process. Another problem with static priorities is that they cannot impose upper limits on uncooperative RT tasks.

In order to correctly schedule multiple tasks with static priorities there must be a fixed task set with priority values carefully chosen in advance (this can be extended to a number of different task sets for modal systems such as aircraft flight control). It is then possible to check that scheduling constraints will be met using offline analysis, or long-term empirical tests.

There are two similar APIs for setting static priorities in common use; the System V `priocntl` interface, and POSIX.1b [Gallmeister95].

Examples

There are many examples of this type of OS, which we will describe briefly.

BeOS [Be99] is positioned as a "media OS" for desktop multimedia applications, and has a fairly conventional architecture. The main improvement over UNIX is a finer timing granularity, with microsecond timers and a 3ms quantum for best effort tasks.

Chorus [Sun99a] offers some distributed as well as RT features. It supports POSIX.1a, 1b and 1c (the threads standard). Scheduling classes include FIFO, best effort and a "user-defined" policy. It is also used as the basis for JavaOS for Consumers [Sun99b].

LynxOS [Lynx99] is aimed at embedded applications and can run from ROM. Although it lacks dynamic priorities it supports an optional MMU for address space protection between tasks.

Real time features include a preemptible kernel and priority inheritance on semaphores. Operating system activities which would normally take place in interrupt service routines, such as I/O, run in kernel threads with priority matched to the user-level process concerned. This is designed to provide a predictable response time to external events. It also supports POSIX.1a, 1b and 1c.

QNX [QNX99] emphasizes the microkernel structure and includes higher level components such as a window system. Priority inheritance is performed for servers by delivering messages in priority order and running the server at a priority equal to that of the highest priority waiting client. Scheduling disciplines include RR, FIFO and dynamic priority.

VxWorks [WindRiver99] is again targetted at embedded applications and provides only static priorities, with priority inheritance on semaphores. It runs on many different architectures (PPC, 68K, ColdFire, MCORE, x86, i960, ARM, MIPS, SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST20 and TriCore).

1.3 Guaranteed real time scheduling

1.3.1 Defining quality of service

An accepted basis for scheduling RT tasks is to replace priorities with a more general Quality of Service (QOS) measure.

QOS can be described by a pair of parameters such as (*CPU share*, *time period*). This amounts to a guarantee on both throughput and timeliness. For example, a contract of (15%, 20ms) specifies that a task must be allocated at least 3ms of time in each successive 20ms interval, assuming that it is ready to run. These intervals start from the time at which the contract is established and occur periodically from then on.

This parameter specification is called a *Contract* or *Reserve*. It is more suitable for RT applications because the scheduling properties are explicit (traditionally they were implicit and usually unknown).

1.3.2 QOS Algorithms

Once QOS contracts have been established for RT tasks, the scheduler must choose which task to run or preempt in such a way that each task receives its proper guaranteed reserve. This can be ensured using the Rate Monotonic (RM) or Earliest Deadline First (EDF) algorithms [Liu73].

Liu and Layland proved these methods guarantee all contracts under certain restrictions, and provided the sum of the CPU shares allocated to RT tasks does not exceed a threshold value. This is called the *schedulability* of the system. RT systems typically perform an *admission control* check before granting new QOS contracts, to ensure that the CPU is not committed to RT tasks beyond this amount.

Rate Monotonic

RM is a static priority algorithm, in which the priorities are set in inverse relation to each task's time period. Tasks with shorter time periods are therefore scheduled first, until their QOS reserve runs out or they block. It is approximately 69% schedulable. On average it performs better; this is the lower bound which holds for any task mix.

Earliest Deadline First

EDF is a form of dynamic priority algorithm. Each task's current deadline is defined to be the end of its current period. The task with the earliest deadline at any particular time is declared to be the highest priority task, and will be scheduled first.

EDF is 100% schedulable but behaves more unpredictably than RM under transient overload (which may occur due to critical sections or synchronisation, for example).

Assumptions

The restrictions under which the schedulability bounds for either algorithm become valid are as follows:

1. All tasks are periodic
2. Deadlines are equal to periods
3. Tasks are independent—they do not synchronise or communicate with other tasks
4. Computation time is constant each period for a given task
5. Tasks are fully preemptive with zero context switch time

These are unlikely to hold in most cases, but can be relaxed in order to adequately cover real systems. The first two restrictions are not too severe because most RT applications behave this way.

Assumption 3 is hard to dispense with, since in reality processes do communicate. Priority inheritance protocols can be used to reduce the effect. Fortunately, processes which do not need to synchronise are unaffected.

Assumption 4 can be relaxed by performing adaptation. The fifth requirement can be allowed for by using a preemptive kernel and allocating slightly less than the maximum available CPU time, to cover overheads.

Time periods

A QOS system implementor must choose which time periods to support. There are three main possibilities:

1. Jubilees
2. Arbitrary periods
3. Harmonic periods

Jubilees are fixed periods which apply to the whole system. This has the advantage of simplicity but leads to a compromise: no task can have a faster guaranteed response than the jubilee time, however very frequent jubilees lead to unnecessary overhead.

The second option is to allow an arbitrary period for each task, according to its requirements.

An interesting alternative is to restrict to harmonic periods. These may differ between tasks, provided that they are a power-of-two multiple of the smallest period. Arbitrary periods can be handled by rounding down to the next smallest power-of-two, and scaling the slice appropriately.

It is very easy to process end of period events in this case, by interrupting at the highest frequency in use and maintaining a queue of tasks for each time period. This also results in fewer timer interrupts than an equivalent system with arbitrary (out of phase) periods. A further advantage is that in this special case the Rate Monotonic schedulability bound is 100% rather than 69%.

1.4 Requirements for a new platform, or "Oh no, not again!"

Developing a new platform is undesirable given the large number of existing operating systems, and is of course a major undertaking. Our particular interests (QOS management, policies and user interfaces) are higher level concerns which don't depend on the particular RTOS used.

The requirements for a platform to develop and test QOS management are broadly as follows:

1. It should support QOS
2. It must run full scale, conventional desktop applications
3. Source code must be available for modification and integration
4. It should be sufficiently well documented and supported

Unfortunately no existing OS (commercial or research) meets both of the first two requirements. This motivated the construction of a new platform, based on an existing one to minimize duplication of effort.

We choose to start with Linux [Beck98], since it meets requirements 2, 3 and 4 very well (but lacks QOS).

The availability of "real" applications is particularly important in order to test policies and interfaces under normal loads and usage patterns. Considerable work has already been done using micro-benchmarks, but such an approach brings no guarantee of realism.

It is easy to develop for Linux, because of the clean, relatively small source code base. It is also stable, well documented and offers the advantage of a familiar environment.

1.5 Dissertation Outline

In the following chapter we discuss the standard Linux scheduler in detail. In Chapter 3 we present a real time platform which supports QOS scheduling and runs standard desktop applications. The next two chapters go beyond the kernel and develop further infrastructure required for a complete system. Chapter 4 describes an improved IPC facility and Chapter 5 develops mechanisms for scheduling real time server processes. Chapters 6 and 7 briefly explore the space of resource control policies which can be built on the new platform. Finally, Chapter 8 illustrates the highest level of our prototype system; the user interface.

Chapter 2

Linux scheduling

2.1 Process terminology

Throughout this dissertation the terms *process* and *task* will be used interchangably, with no special meaning attached to either. A process comprises one or more *threads*, and the word thread is used when we wish to emphasise that several execution contexts may run within one address space.

Linux actually unifies all three terms (task, process and thread), since the same data structure `task_struct` is used to represent any of them. Whenever a new process is created, various properties of the parent may be passed on or shared with the child. In addition to familiar aspects such as open files, and (under Linux-SRT) scheduling parameters, this includes the address space. A new thread is thus created by handing the `do_fork()` routine the flag `CLONE_VM`, which requests that the address space be shared, instead of creating a new one for the child.

In the context of Linux-SRT we define another term, Thread ID (TID), which we prefer to use rather than Process IDs (PIDs). A TID consists of the following tuple: (PID, Start time, Host IP address). Platform-dependent alternatives may replace the PID field.

Using process identifiers would have some disadvantages for representing tasks in a real time networked environment. Firstly, they can be reused, which would cause problems if an operation is applied to the results of an earlier query. Although PIDs wrap around rarely in practice, no guarantees are made that they will be valid at any time. Secondly, they do not specify the host on which the process is running. By contrast TIDs identify threads uniquely.

2.2 Scheduling algorithm

The Linux timesharing scheduler fulfills the role of `SCHED_OTHER` in the POSIX real time policy classification.

The algorithm is different (and quite a bit simpler) than the BSD UNIX one, but achieves similar results. The notions of dynamic priority and timeslice are merged into one and represented by a single *counter* value for each task.

The counter for the running task is decremented at each timer interrupt. If the counter reaches zero before the process blocks, a mandatory reschedule is performed (which will preempt the process if anything else is ready to run). Also, if a process with a higher counter value unblocks, preemption occurs immediately. Whenever the scheduler is called, it chooses the task with the highest value of counter to run. The counter is therefore a form of dynamic priority, as well as representing the remaining timeslice in ticks for CPU-bound tasks.

If there are no runnable processes with counter greater than zero when the scheduler is called, it recalculates the counters (for all tasks, whether runnable or not). This is done by dividing the current counters by 2 and adding on their base value. Processes which are rarely scheduled will therefore always have a high counter.

2.3 The *nice* parameter

The base *counter* value for each task is derived from its *nice* parameter. The latter is restricted to the range [19, -20] and corresponds to a base counter value in the range [1, 40] (note the inverted sense). By default *nice* is 0, which sets initial counters to 20. The periodic timer on Linux normally interrupts 100 times per second, which means the default timeslice for CPU-bound processes is 200ms.

In the presence of competing CPU-bound processes, the *nice* parameter therefore functions as a weighting factor. Nice 19 corresponds to a multiplicative weight of 0.05, nice 10 to a weight of 0.5, and nice -20 to a weight of 2.

Table 2.1 shows the CPU usage observed for two tasks, one of which is adjusted with *nice*. Both tasks are running an infinite loop. The results correspond with our expectations; for example a task at nice level 19 has a default timeslice of only one tick, compared to 20 ticks for a normal task.

Nice level	Base counter	Nice task	Normal task
-20 (least nice)	40	67%	33%
0 (without nice)	20	50%	50%
10 (typical nice)	10	33%	67%
19 (maximum)	1	5%	95%

Table 2.1: CPU allocation to *nice* tasks

2.4 Real time support

Linux supports POSIX real time static priorities in the range $[0, 99]$. Static priority policies consist of `SCHED_FIFO` and `SCHED_RR`. Tasks in the former category are not preempted unless a higher priority task becomes ready to run; the latter schedules tasks with equal static priorities on a round robin basis.

The round robin variant has no explicit support for adjusting time quanta. Examining the code we see however that the time quantum can be set, on a per task basis, by adjusting the *nice* value for a real time task. This is still used to recalculate time slices, and therefore can be used to set a quantum between 1 and 40 ticks (10 to 400ms) as usual, for `SCHED_RR` tasks. Timing tests also verify this.

2.5 Disadvantages of Linux

The Linux architecture is based on a traditional monolithic UNIX design. This poses serious problems when it is used as a RT system, as do some of the legacy APIs and data structures.

2.5.1 Real time aspects

Linux is not at all suited to real time applications. The main problems are listed below. Note: we defer discussion of OS structures in detail until Chapter 4.

- Processes running in kernel mode cannot be preempted by more urgent ones in user space, or by others in kernel mode.
- Interrupt latency can be considerable (some critical sections in the disk device drivers disable interrupts for up to 400μs [Srinivasan98]).
- Server processes (such as the X11 graphics server) perform work for clients in a different scheduling domain; moreover requests cannot be preempted by more urgent ones within single-threaded servers.
- The low granularity of system clock ticks (100 Hz) affects the resolution achievable for timers and scheduling.

2.5.2 Other APIs

The following list describes some shortcomings of the Linux kernel which don't directly relate to RT support, but became significant during the implementation of Linux-SRT:

- The kernel is implemented in C rather than an OOPL. One consequence is that different types of list and their associated manipulation routines clutter the namespace and require code duplication.
- There is a lack of 'capabilities' for fine grained access control (some support has appeared in later kernels; see Chapter 5).
- It is not easy to perform reverse lookups from sockets, window structures and process names to the corresponding process identifier.
- Although many IPC mechanisms are offered they are not sufficiently modern or flexible (see Chapter 3).

2.6 Measurements

Before designing a RT system, we wish to understand the timing characteristics of system activities and typical applications. There is a lack of published data in this area. We therefore began with a number of timing measurements. The following observations were obtained with a 133 MHz Pentium processor, using a Linux kernel unmodified except for the tracing code.

2.6.1 Application burst lengths

The burst length for a process is the time for which it runs before blocking or being preempted by the scheduler.

We modified the scheduler to record the burst lengths for all processes during their entire runtime. This was done by defining a large number of timing intervals and maintaining frequency counts of these ranges for each task.

The frequencies with which each timing interval occurred were multiplied by the length of the interval, and the results displayed in bar chart form. By plotting frequency multiplied by time, the distribution of area under the graph illustrates the burst lengths during which the process spent most of its time (an unscaled frequency graph would emphasise a few short bursts in which the process was preempted by an OS activity over its "natural" burst length).

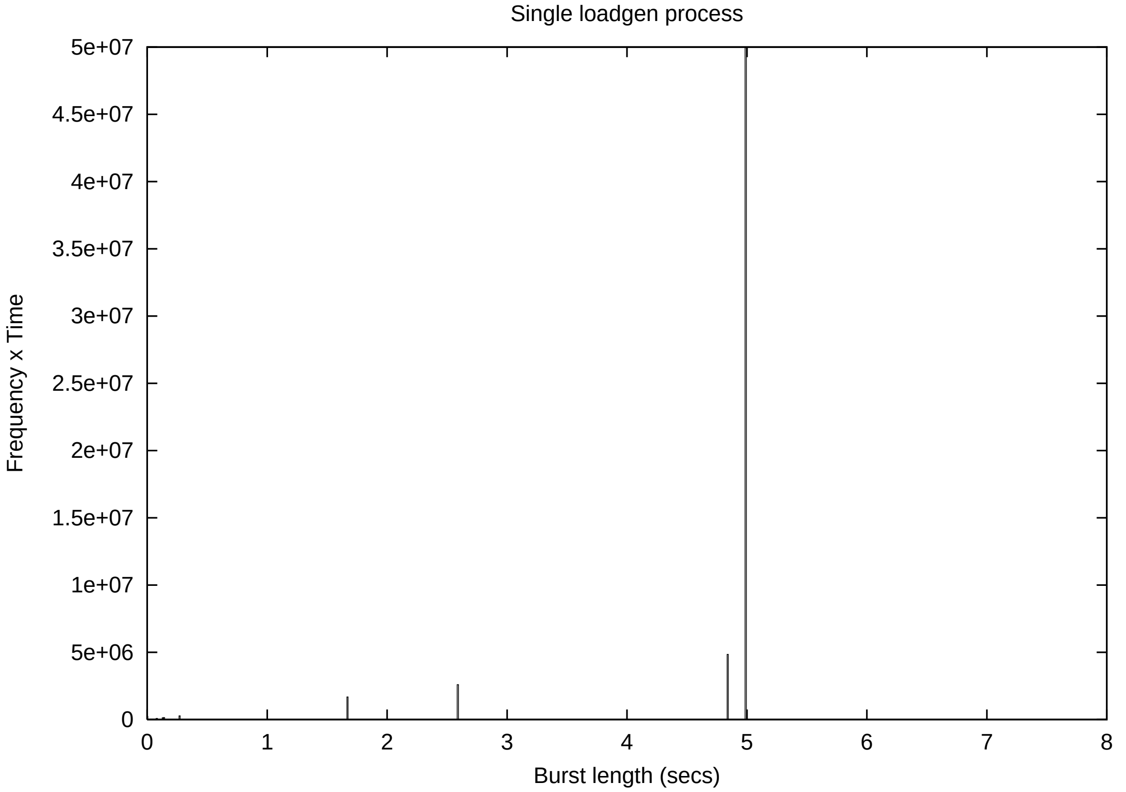
Figure 2.1 shows the trace for one or two load generator processes, consuming CPU time in a tight loop without blocking. A single such process typically runs for exactly 5 seconds before preemption (and never longer). This is likely to be the time at which a critical system task (probably the swapper) wakes up to perform general maintenance checks.

This behaviour was recorded with the system in single-user mode, running the `twm` window manager. Interestingly, most other window managers and desktop environments performed some regular processing which limited the burst lengths to far less than 5 seconds.

In the second part of Figure 2.1, two load generators are running simultaneously. Each one runs for its whole timeslice of 200ms and is preempted on the next clock tick.

Figure 2.2 repeats the experiment using raytracing processes. These occasionally render a few pixels of graphics to update the scene on-screen as the raytrace progresses, causing the X server to be scheduled. This leads to a more uneven distribution of run times, with a maximum of about half a second.

With two raytracers running simultaneously the burst lengths are reduced due to competition for the CPU, and there is a peak at 200ms which is the default timeslice. No burst length exceeds 400ms, because the maximum possible dynamic priority is twice the base value (recall that existing counters are halved before the base is added during priority recalculation).



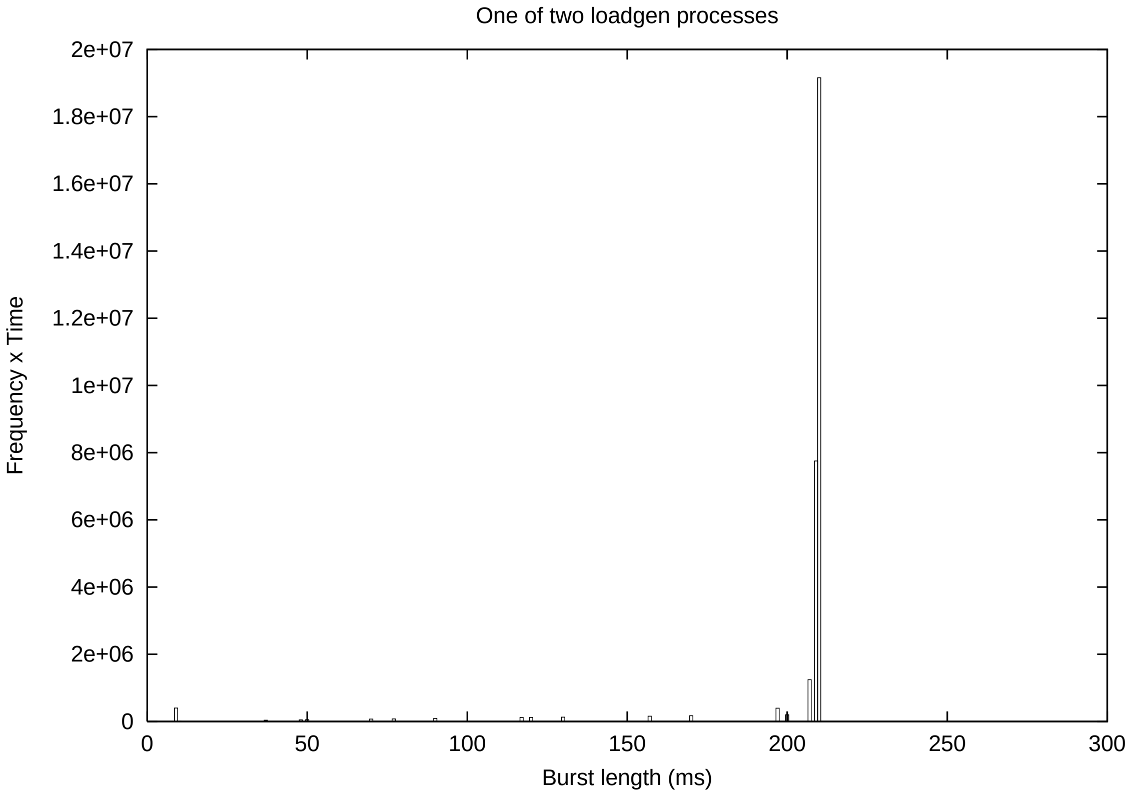


Figure 2.1: Burst lengths for load generator processes

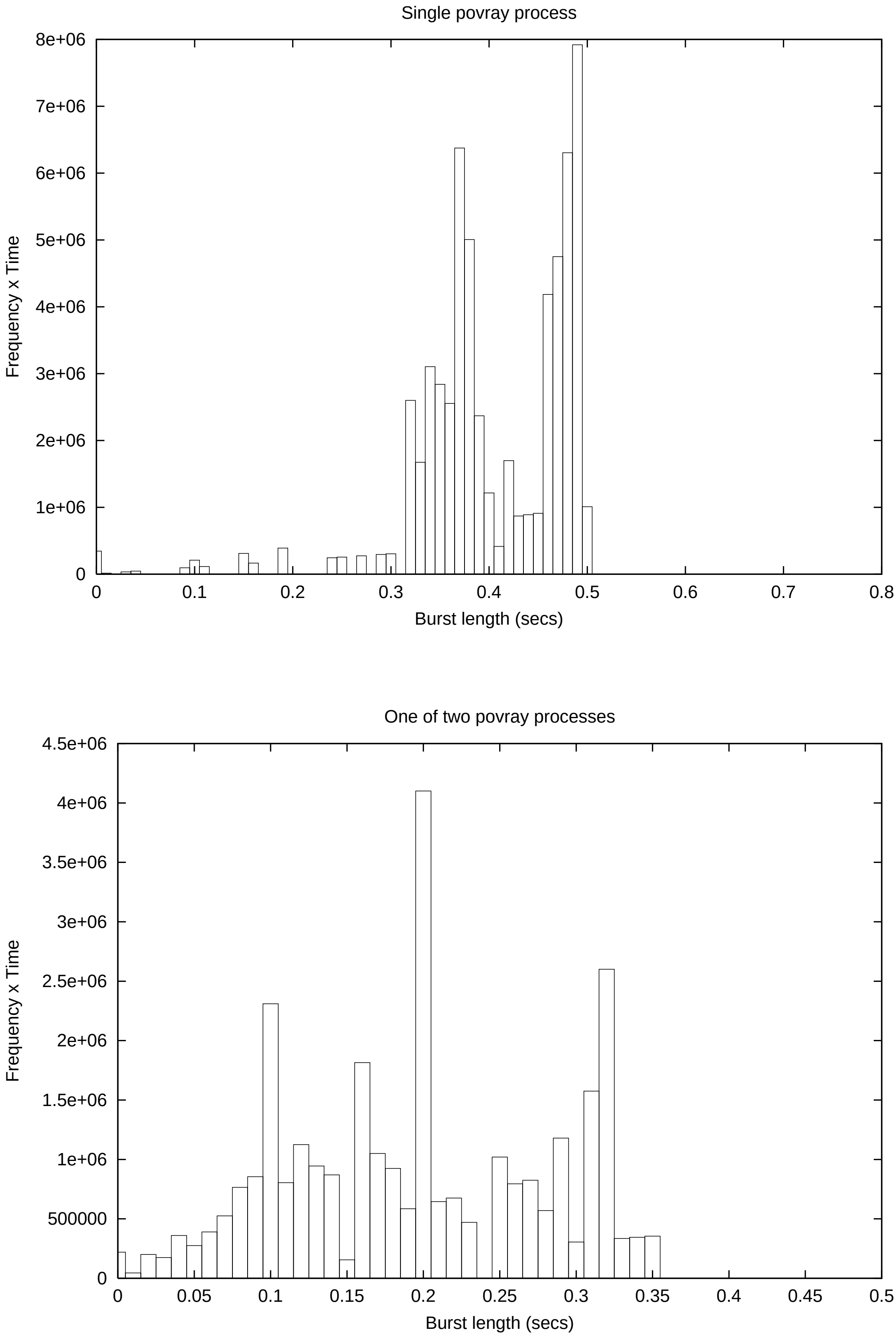


Figure 2.2: Burst lengths for raytracing processes

Figure 2.3 illustrates two more realistic, naturally bursty processes (an mp3 music decoder and a Quicktime movie player). They have natural timeslices of 6 and 10ms respectively and are very consistent because they deal with constant bit rate streams.

Finally, Figure 2.4 gives the distribution for the X server process, as observed after a period of "normal" use. This is much more uneven and has several spikes based on common patterns of X request. The peak around 10ms may be due to preemption after a timer interrupt.

2.6.2 System call duration

The time taken to complete a system call is one factor which determines the responsiveness of the system. This was measured by repeating calls large numbers of times. A "do nothing" system call was found to take 1.7μs on our test system. This is also the approximate time to make trivial calls such as `getpid()`, which merely return a value from a kernel data structure.

Another important call is `gettimeofday()`, which is used by the test programs to obtain timestamps at roughly microsecond accuracy. This was found to take 2.8μs. Allowances for this delay were made when computing other results with this call.

2.6.3 Time spent in the kernel, servers and applications

It is important to understand how much time the system spends in kernel mode (which cannot be preempted) and executing server processes. Any such time will lead to inaccuracy or jitter if it is not accounted for or made preemptive, respectively. The following proportions were observed under realistic loads in a normal working environment, and the results are shown in Table 2.2. The actual measurements were taken to microsecond accuracy using the new tracing facilities. The table shows kernel and user-space processing, time spent in the X server compared with all other tasks, and busy versus idle time.

LOAD	KERNEL	USER	X SERVER	OTHER	BUSY	IDLE
Long-term average	30%	70%	29%	71%	20%	80%
Audio playback	15%	85%	5%	95%	30%	70%
Video playback	42%	58%	37%	63%	100%	0%
Raytracing	2%	98%	1%	99%	100%	0%

Table 2.2: Kernel and server processor usage

The results show that a considerable amount of time (typically around 30%) is spent in both the kernel and X server. Raytracing is an exception as it executes in user-space with few system calls (some I/O is performed to update the status output to the console). Audio playback also uses less kernel time than average (a small amount of X server usage was caused by the GUI track name and elapsed time display).

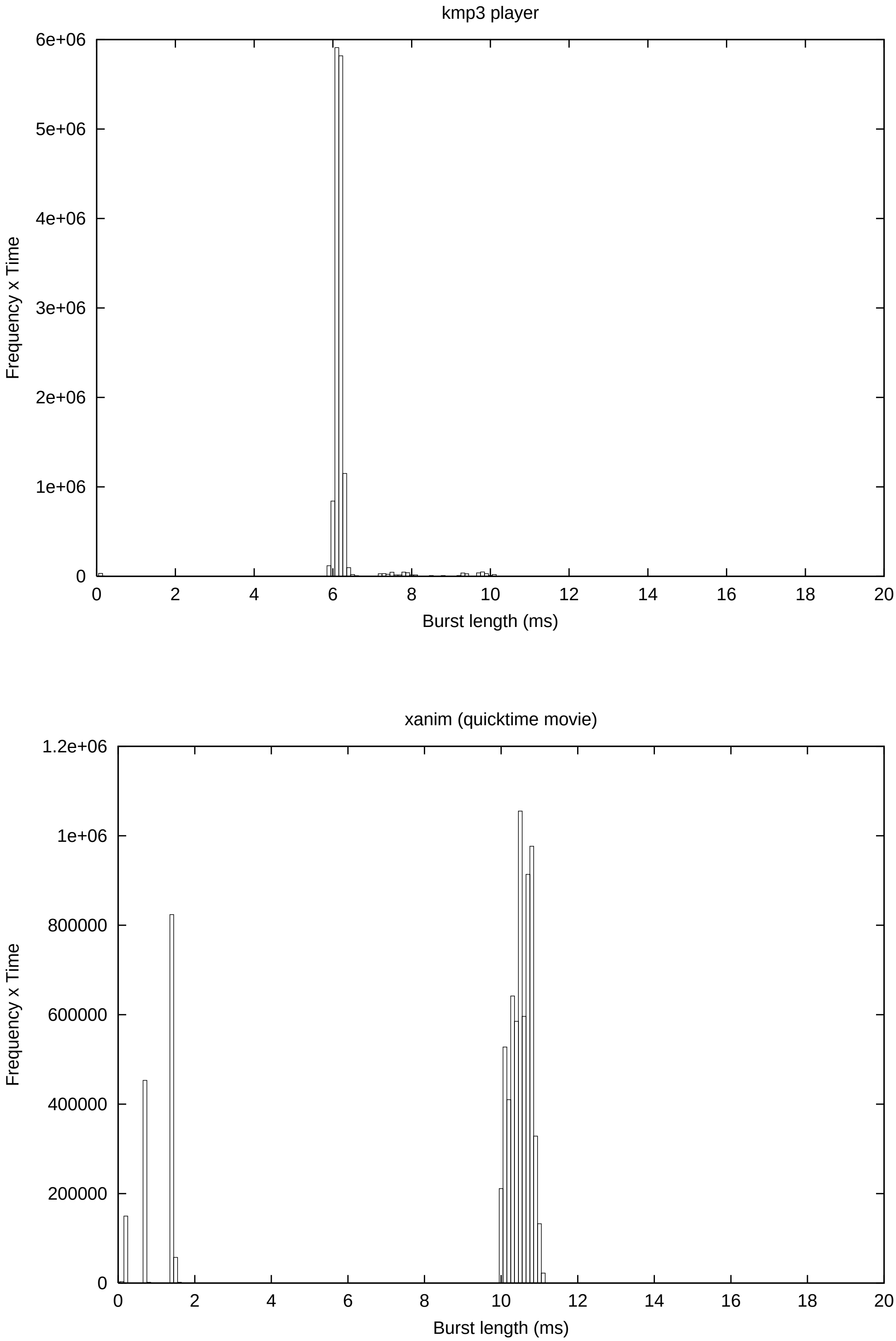


Figure 2.3: Burst lengths for multimedia processes

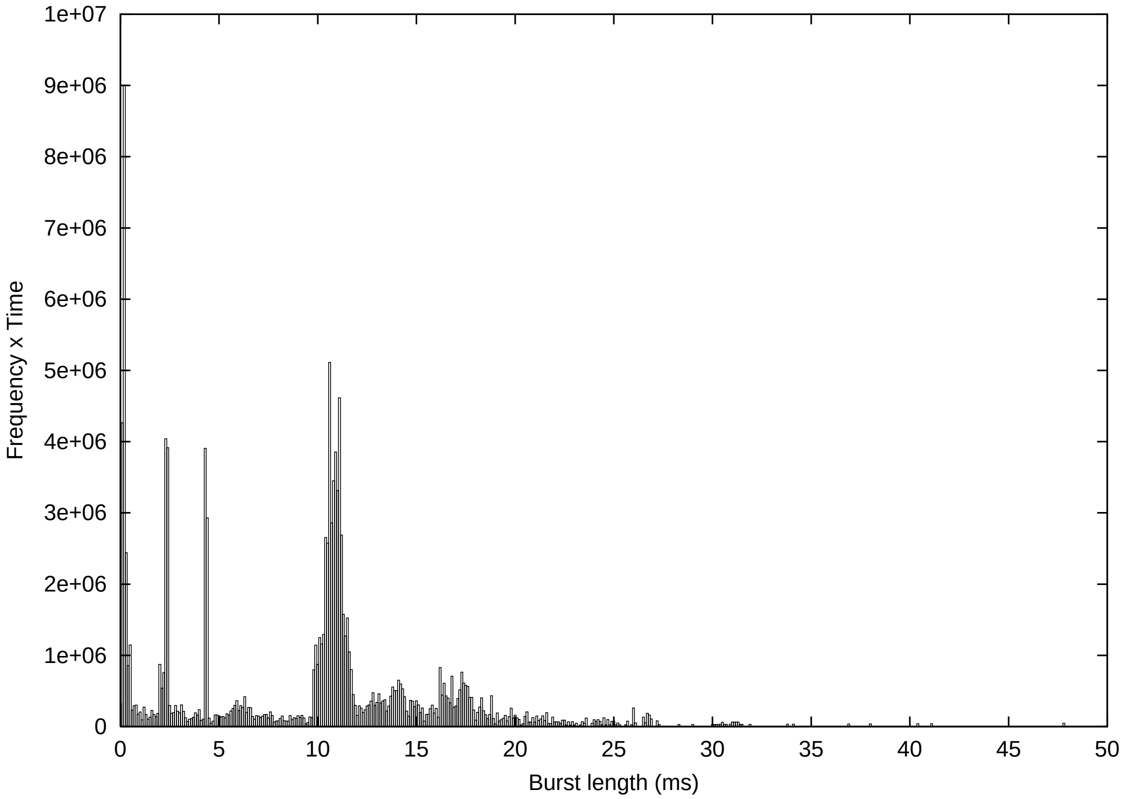


Figure 2.4: Burst lengths for X server

Note that usage related to servers other than X11 was minimal on our test machine. The long-term average figures for kernel and X server usage were surprisingly consistent ($\pm$ 2%) when measured on different occasions.

Chapter 3

Implementation

3.1 A New Real Time Platform

We have constructed a prototype operating system called *Linux-SRT* to investigate RT scheduling in a desktop environment. It is based upon a standard Linux kernel, with modifications to the kernel and X server, and user-space control programs.

Linux-SRT runs all normal Linux software and is stable enough for everyday use. It does not have to be switched into a special RT "mode", and RT processes have access to all normal services and APIs. A quality of service guarantee may be applied to any existing program without the application writer needing to have anticipated this or written any special code to handle it. We use Linux-SRT itself as our development platform.

The applications which have been used to test the system include a normal range of web browsers, office suites, movie players, other multimedia programs and a raytracer in addition to a few benchmark test-cases.

In this chapter we describe the modifications we have made to Linux in order to support QOS.

3.2 Timing granularity

The most accurate way to improve the timing granularity of the system is to program the timer in one-shot mode. Timer interrupts then occur aperiodically, at whichever times are necessary to preempt the running task. This is determined by the current task's remaining QOS allocation.

The UTIME kernel patch developed at the University of Kansas [Srinivasan98] takes this approach to overcome the kernel's coarse scheduling granularity. It also inserts pseudo-deadlines at regular 10ms intervals to simulate the normal periodic ticks, so that the rest of the system is unaffected.

Linux-SRT takes a much simpler approach by increasing the frequency of the periodic timer instead. This adds to the jitter experienced by RT tasks but is currently the most stable solution. The timer fires at a rate specified by the kernel constant `HZ`, which we have increased from 100 to 1024.

The default timeslice used by BE tasks is determined by the kernel constant `DEF_PRIORITY`. This establishes the dynamic priority which is assigned to tasks with a zero *nice* factor when the counters are recalculated. The value used is scaled depending on `HZ`; this ensures that BE timeslices are not altered by the increased periodic timer rate.

3.3 Scheduling classes

Linux-SRT adds several new scheduling classes to the standard ones supported by POSIX. The complete set of classes consists of `SCHED_PAUSE`, `SCHED_IDLE`, `SCHED_OTHER`, `SCHED_RR`, `SCHED_FIFO`, `SCHED_QOS` and `SCHED_VAR`.

The scheduler gives priority to tasks with guaranteed time (`SCHED_QOS`), followed by the static priorities (`SCHED_RR` and `SCHED_FIFO`). Next in precedence are the best effort tasks (`SCHED_OTHER`) and finally those from the idle class, `SCHED_IDLE`. Tasks in the `SCHED_PAUSE` class are ignored by the scheduler. `SCHED_VAR` is a variable priority scheduling class used for priority inheritance with RT servers (see Chapter 5).

BE tasks are therefore scheduled by the normal Linux algorithm whenever no RT task is ready to run. Limits on RT task reservations ensure that starvation of the BE tasks cannot occur. The idle class is used for processes which must not affect the normal responsiveness of the machine, such as batch jobs. The BE scheduler handles both unreserved time and unused reserved time.

In addition to the scheduling class, there are a number of scheduling parameters associated with each task. These are listed in Table 3.1. Fallback policies are described in more detail in Section 3.5, Dual policy scheduling.

Parameter	Description
Policy	Normal scheduling class.
Fallback policy	Scheduling class to use when overrunning (exceeded slice for this time period).
Inherit	Boolean flag which specifies whether child processes inherit these parameters.
Nice	In the range -20 to 19. Affects `SCHED_OTHER` service (also determines the timeslice for `SCHED_RR`).
Priority	Static priority for `FIFO` or `RR` policies in the range 1 to 99.
Period, Slice	QOS guarantee (for `SCHED_QOS`) or limit (all other policies). May be null.

Table 3.1: Scheduling parameters

3.4 Kernel Scheduler Internals

The scheduler uses the Rate Monotonic algorithm. Only a few changes to the standard POSIX RT scheduler are required to support RM, since both use static priorities. The modifications required to enable QOS are (i) *accounting*, (ii) *policing* and (iii) *admission control*. These are described below.

Priorities are chosen when contracts are established, in such a way that the tasks with shorter time periods have higher priorities, as required by the RM algorithm.

RM scheduling is more stable than EDF under transient overload; the deadlines of the longest period tasks will be missed first, rather than all tasks failing at once. Static priorities also have the advantage that priority inheritance is easier to implement.

3.4.1 Time periods

Arbitrary time periods are supported. Although harmonic periods (see Section 1.3.2) are attractive theoretically, we found that the extra efficiency is not worthwhile in practice, because the scheduler overhead is so small. Furthermore the behaviour of the system was observed to depend in a critical way on whether the time periods were set close enough to the correct value. When a smaller time period was chosen and the slice scaled down accordingly, considerable scheduling problems occurred.

The API therefore exposes an arbitrary time period model, although the underlying system could also be said to exhibit characteristics of a fine-grained Jubilee model, due to the periodic timer interrupt limitations.

3.4.2 Accounting

Accounting in UNIX consists of noting which process is active at each clock tick (every 10ms). Because this is so infrequent, the CPU usage estimates are only valid for very long time spans. We obtain much more accurate accounting information by reading the processor's Time Stamp Counter (TSC) each time a reschedule is performed. CPU usage is measured to microsecond accuracy and attributed to the correct task.

3.4.3 Policing

RT tasks must be preempted immediately if they exhaust their CPU allocation for the current time period before blocking, unless their fallback policy entitles them to carry on running. This function is called *policing*. Preempted processes only return to the conceptual QOS ready queue at the start of their next period, when they are allocated another chunk of CPU time. Any process which still has guaranteed time left is scheduled in preference to those which have overrun, or don't have a QOS contract.

3.4.4 Admission Control

The kernel performs *admission control* to grant or refuse new requests for QOS. A contract is not granted if it would invalidate any previously established contracts. If a new RT task begins but cannot be guaranteed it runs anyway, without QOS. These policies may be adjusted by the QOS manager (see Chapter 7).

3.5 Dual policy scheduling

The Linux-SRT scheduling parameters include a rate limit and a fallback policy. The limit is calculated as the slice divided by the period. When a task has exceeded its current QOS allocation, its scheduling class drops to the fallback policy until the start of the next period. In this way QOS may be used to limit the CPU consumption of a process to a fixed maximum amount, as well as providing a minimum level of service.

Some examples are given in Table 3.2.

DESCRIPTION	POLICY	LIMIT	FALLBACK
Optimistic	`QOS`	25%	`OTHER`
Hard guarantee	`QOS`	25%	`PAUSE`
Ceiling	`OTHER`	50%	`PAUSE`
Cut to idle	`OTHER`	50%	`IDLE`
Safety limit	`FIFO`	80%	`OTHER`

Table 3.2: Example scheduling parameters

Notice that limits can be applied to any task, including static priority ones. This is a very useful facility, not least to prevent a crashed static priority process from halting the entire system. In practical day to day usage several RT applications (such as RealPlayer) proved to be quite liable to crash in this way.

The limits on such processes are not of course *guarantees*, unlike those for `SCHED_QOS`; neither are they subject to admission control. The sum of the limits on static priority tasks may therefore exceed 100% at any given time.

In addition to the per-task limits, there is a global limit stored in a kernel variable called *rsvd*, which can be adjusted through a system call interface. The admission control check for `SCHED_QOS` refuses to allocate more than $100 - rsvd$ percent of the processor time. The value of *rsvd* is 31% by default, which guarantees the correctness of the RM scheduling algorithm, although we reduced it to 10% after some experimentation.

The use of `SCHED_QOS` with a `SCHED_OTHER` fallback policy corresponds to the notion of an *optimistic* task in the Nemesis OS [Leslie96]. This is a RT task which has a flag set indicating it can benefit from receiving more than its standard CPU slice if time is available.

Tasks running optimistically should not have outright priority over BE tasks, because this would lead to starvation of the BE tasks if a RT task never blocked. Conversely, the BE tasks must not have outright priority over optimistic RT ones, or a long background job would prevent any optimistic scheduling for the (presumably important) RT tasks.

The best solution is to temporarily demote RT tasks which have overrun to BE status, placing them under the remit of the standard BE scheduler. This is expressed naturally in Linux-SRT with a `SCHED_OTHER` fallback policy. If strict upper bounds are required this can be achieved by falling back to `SCHED_PAUSE`; alternatively `SCHED_IDLE` provides a low-impact overrun policy.

3.6 POSIX API

Our first implementation made the new scheduling classes and parameters available by extending the standard POSIX scheduling system calls in the recommended way. The interface is shown here:

```
struct sched_param
{
   int sched_priority;
   ...
};

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_setparam(pid_t pid, const struct sched_param *p);
int sched_getparam(pid_t pid, const sched_param *p);
```

This approach was later rejected because it implicitly assumes that scheduling parameters are associated with *processes* rather than some other abstraction such as *reserves*. This proves to be very inconvenient when reserves are shared, for example between clients and server threads. In this case there is a many to one relationship between processes and reserves. If a `sched_setparam` call is made for one such process it is not clear if the whole reserve should be altered, or just a single thread.

3.7 Linux-SRT API

The revised Linux-SRT API provides functions to manipulate *reserve* objects, and for associating reserves with threads. The reserves encapsulate descriptions of scheduling characteristics together with any associated guarantees. They exist in their own right, even if no threads have *subscribed* to them. Threads which have joined the reserve are called members of the reserve, and all share the same scheduling properties and QOS.

3.7.1 Reserve names

All reserves have a name, in the form of a text string. These appear in `/proc/reserves`, together with the parameters and a list of suscribed client threads. Applications may join a reserve specified by name. If it does not yet exist it is instantiated with parameters from the QOS configuration file; otherwise the thread joins the existing reserve.

Named reserves solve the problem of sharing between unconnected threads. For example, we may wish to share a reserve between the window manager and shells on different virtual consoles, to ensure all aspects of the user interface are responsive. Multiple reserves would be inappropriate and wasteful of guaranteed time, since only one of the consoles can actually be used at once.

Another problem addressed by named reserves is that of multithreaded applications which have different scheduling requirements for each thread. Examples include Alsaplayer or XMMS (X Multimedia System). These media player programs have a RT audio thread, a normal UI thread and several spectrum analyzer or scope threads.

Applying a single priority to the whole program in these cases without application-specific knowledge gives poor results. For example, the audio thread has SRT requirements, but the scopes are good candidates for *nice* scheduling. On the other hand, if the application contains hard-coded priorities for each of its threads, these may well conflict with those chosen by other programs. Requirements may also not be fixed, for example if the analyzer threads are of current interest to the user, smooth updates will be required and they should not be scheduled with *nice*.

Standard Linux systems provide no way for the user to set priorities or QOS, so multimedia applications often do try to select static priorities themselves. The results are often undesirable as they have no knowledge of other tasks. Furthermore, they must be running as root to do this, which is dangerous for ordinary applications.

Named reserves solve these problems, because the application may attach each type of thread to a different reserve. The parameters associated with each name can be configured by the user in advance, before the application runs.

3.7.2 Namespace management

In cases where there is no need to refer to a reserve by name, an *automatic reserve* facility is provided for convenience. One automatic reserve can be created for each running process. A null string may be used as reserve name to indicate the automatic reserve (which will actually be named `auto <pid>`, where `<pid>` is the process it relates to). The `auto` prefix is reserved for automatic reserves.

Named reserves are never removed from the namespace until an explicit `delete_reserve` call. Automatic reserves are deleted when their owner dies, if they have no remaining members at that time. Otherwise, the reserve is instead renamed according to the pid of the first remaining member. This ensures that daemons which fork and exit to put themselves into the background do not lose a reserve configured for them by the QOS manager. The transfer of ownership also maintains the relationship between automatic reserve names and live process pids, so no conflicts occur after pid's wrap around.

The admission control checks for all reserves (named or automatic) which include a QOS guarantee occur when the number of tasks subscribed to the reserve exceeds zero. The resources are freed when the number of members drops back to zero.

3.7.3 System Call Synopsis

The following data structure is used to describe the scheduling classes and parameters comprising a reserve by the Linux-SRT API: [1]

```
struct rsv_param
{
   int policy, fallback_policy;
   int period, slice;
   int static_priority;   // only used for FIFO or RR
   int nice;              // only used for OTHER or IDLE
};
```

Any of the seven different scheduling classes described in Section 3.3 can be used for *policy* and *overrun_policy*. The *period* and *cpu* fields are optional (*policy* is used at all times if they are null).

`int sched_inherit(int` *pid*`, int` *inherit*`);`

> Queries and sets the *inherit* flag for process *pid*. This determines whether child processes will inherit the scheduling properties of the parent. If it is set and the parent belongs to a QOS reserve, the children will be subscribed to the same reserve.

`int sched_query_reserve(int` *pid*`, char *`*name*`);`

> Fill in *name* with the reserve used by process *pid*.

`int get_reserve(char *`*name*`, struct rsv_param *`*rsv*`);`

> Fill in *rsv* with settings for the reserve indicated.

`int join_reserve(int` *pid*`, char *`*name*`);`

> Process *pid* joins the reserve given by *name*. An implied `leave_reserve` is performed first if necessary. If this is the first member to join a QOS reserve and allocation fails, the reserve remains uninstantiated (but still in the namespace) and the call fails.

[1] Some of the field names have been changed to make them easier to understand. The actual interface uses alternatives which correspond better with established naming practice in the kernel. We also omit many details related to error handling from the system call synopsis. Please consult the Linux-SRT documentation for exact programming information.

`int leave_reserve(int` *pid*`);`

Leave the reserve currently in use by this process. Note: when a process exits for any reason it calls `leave_reserve` automatically.

`int set_reserve(char *`*name*`, struct rsv_param *`*rsv*`, bool` *create*`, bool` *modify*`);`

This call defines a reserve with the given name (an automatic reserve can be specified by the empty string of course). Note that a subsequent and explict `join_reserve` is necessary if the same process wishes to be a member of the new reserve.

If the reserve does not exist, the create flag specifies whether to create it. If the reserve does exist, the modify flag specifies whether to modify it.

An attempt to modify an existing, instantiated reserve may fail if the QOS allocation needs to be increased. In this case the reserve retains its current values and members, but the call returns `-ENOSPC`. An existing reserve which has no members can always be modified without any error possibility.

`void delete_reserve(char *`*name*`);`

The reserve must have no members. If so it is removed from the namespace.

`int sched_yield(void);`

This is an existing system call which we have extended. If called by a process using a reserve, the effect is to immediately drain the current allocation of CPU time to zero. The reserve will be refreshed at its next epoch as usual.

This can be used by programmers to synchronize an application with its reserve, avoiding the need for low resolution and unpredictable interval timers.

3.8 Results

Here we evaluate the effectiveness of the RT scheduler, and measure the overhead of increasing the timing granularity.

3.8.1 Kernel scheduler effectiveness

The scheduler was assessed by measuring the minimum timing granularity which can be achieved by a periodic task, without incurring any missed deadlines. To eliminate noise these measurements were taken under controlled conditions (not during normal running). The results are shown in Table 3.3.

We performed eight sets of experiments. The three parameters which are varied are the periodic timer frequency (in HZ), the scheduling policy applied to the target task, and the presence or absence of a background load. The background load process, if present, is always scheduled with `SCHED_OTHER` and never blocks.

In each case, the target task was adjusted to require either 20, 40, 60 or 80 percent of the total CPU available, by blocking and unblocking periodically to simulate bursts. Various time periods were then tested (all values from 2 to 64, together with 128, 256 and 512 milliseconds). We recorded the shortest period for which the process was scheduled reliably enough to complete all its work every time period without exception (each test lasted 5 seconds).

Test Parameters			Slice			
BG Load	Timer HZ	Policy	20 %	40 %	60 %	80 %
No	100	`OTHER`	20	30	40	128
No	100	`QOS`	18	28	40	128
No	1024	`OTHER`	2	2	3	6
No	1024	`QOS`	2	2	2	2
Yes	100	`OTHER`	20	30	Fail	Fail
Yes	100	`QOS`	18	28	40	128
Yes	1024	`OTHER`	2	2	Fail	Fail
Yes	1024	`QOS`	2	5	16	18

Table 3.3: Minimum time periods (ms)

Several things are apparent from this data. Firstly, increasing the periodic timer frequency gives a corresponding improvement to the minimum schedulable time period. Secondly, when there is no background load, the use of QOS makes little difference. This is unsurprising since the BE scheduler gives our target task as much time as it wants in this case. The minimum time period of 20ms for a light task on an unloaded machine is equal to 2 ticks of the system clock (it takes 1 tick to wake the task up ready for its next burst, so a single tick period is not possible).

In the presence of a background load, the target task can still meet all its deadlines with the Linux BE scheduler, provided its CPU utilisation is less than 50% (this will ensure its counter value is higher than the background task). If it requires more than 50% CPU it fails to meet deadlines regardless of the time period, unless QOS is used. With QOS, it can indeed be scheduled, with a time period less than 18ms when the timer is set to 1024 HZ. This is suitable for video applications which require fewer than 55 frames per second, for example.

3.8.2 Kernel scheduling overheads

We have observed that the periodic timer frequency must be increased to achieve good response times, but doing so also increases the overhead due to interrupts and context switches. Table 3.4 measures the throughput in the system with different values of the timer frequency and default timeslice. One, two or five load generator tasks are simultaneously competing for the processor.

The number of tasks and default timeslice have little effect; most of the overhead is from timer interrupts and not context switches.

Periodic timer Timeslice	100 HZ 200ms	1024 HZ 200ms	1024 HZ 20ms
1 task	22.16	21.86	21.83
2 tasks	22.15	21.79	21.77
5 tasks	22.13	21.69	21.68

Table 3.4: Throughput (millions of loop iterations per second)

The results show that throughput is between 1.5 and 2% less with the higher clock frequency. This is an appreciable amount, but was just small enough to make one-shot timer programming unnecessary for our prototype. These results are based on a 133 MHz Pentium; the drop in throughput would be correspondingly less with a faster processor.

3.9 Other resources

The processor is not of course the only resource which must satisfy scheduling guarantees in order to create an effective RT system. Memory availability is of particular interest, as are network, disk and graphics processor bandwidth. Processor reservations may be replaced by more general *resource sets* [Jones95], which grant several resources at once.

Reservations may be

- *fixed* if held exclusively with space division (locked memory pages, audiochannels)
- *adapted* if there is significant overhead to reallocate (memory)
- *scheduled* if shared by time division (CPU, bus)
- *statistically multiplexed* if we rely on clients not to all request service at once (networks, buffers)

Statistical multiplexing does not work as well for CPU scheduling as for networking, because the number of entities (tasks instead of communication streams) is too small.

This dissertation is largely concerned with CPU allocation, because it is a scheduled resource. We comment briefly on the memory and network resources below.

3.9.1 Memory

Most RT systems handle memory simply using the POSIX `mlockall()` function, which ensures that RT applications stay entirely resident in memory.

More flexible solutions would be to guarantee either a certain number of page frames or a specific set of pages. In the former case it may not be easy to guess how many pages should be reserved for good performance. Adaptive methods could be used to calibrate this. If paging is allowed for RT processes, the disk bandwidth resource also becomes relevant. It is unclear how to define QOS if several *interdependent* resources are required.

Placing upper limits on the number of physical pages used by a process is also useful, for regulating BE tasks. The standard UNIX mechanism for placing limits on the number of *virtual* pages cannot distinguish between a program with a large VM footprint but a small working set, and a similarly large program which touches all its data pages in rapid succession. Unfortunately the latter may overwhelm the system by absorbing all free page frames, even when run at a very low priority.

3.9.2 Network bandwidth

Network protocol processing is usually performed inside the kernel, and the time taken not attributed to applications correctly. This is partly due to the asynchronous nature of incoming network traffic.

Most approaches to scheduling protocol processing focus on lazy receive techniques. Nemesis [Leslie96] uses packet filtering or self-selecting NICs to distribute incoming data to the correct application, where it is processed by the application thread. RT Mach also puts the protocol stack in a library [Mercer96b] so that processing is performed in user space. An enhanced version of FreeBSD applies PSS to incoming packet processing within the kernel [Jeffay98].

Real time networked applications include network audio servers and remote X clients. In general, an application may be running on a different machine from the one rendering graphics, whilst the data might be coming across the network from a third site. End to end QOS is needed to honour guarantees for such a program.

It is advantageous to treat end to end QOS as a middleware service, rather than integrated with the platform's scheduler. This is because networks are naturally hetereogeneous and we cannot expect all hosts to be running the same OS. Of course we must ensure that the OS running at each node does support QOS.

3.10 Related work

Other systems which provide QOS include RT-Mach, Nemesis and Rialto. Unfortunately none of these can run unmodified mainstream applications. This makes it unclear how well they manage typical work loads.

3.10.1 RT-Mach

RT-Mach is an extension of the Mach microkernel. The RT scheduling additions are known as Processor Capacity Reserves (PCR) [Mercer94]. PCR provides QOS and implements RM scheduling. Mach has a UNIX emulation layer, although applications which run on it are handled by the UNIX scheduler, so cannot request QOS.

3.10.2 Nemesis

Nemesis [Leslie96] is an entirely new RTOS aimed at multimedia applications, developed at Cambridge University. Scheduling is performed using the EDF algorithm. Nemesis has a *vertically integrated* structure (see Section 5.1.4). Shared servers are largely replaced with libraries; this enables precise accounting of resources and high degrees of preemptivity. Existing applications are not supported although some have been ported to run on the system.

3.10.3 Rialto

Rialto is a RTOS developed by Microsoft Research [Jones95, Jones96]. The scheduler is based on a Least Slack Time First algorithm. Rialto supports *resource sets* and *activities* in order to reserve multiple types of resource for use by a given group of threads. Compatability with a subset of the Win32 API is provided; standard Win32 applications require modifications to run on this platform.

In addition to periodic guarantees, Rialto allows for one-shot "time constraints". These are of limited utility because they may be refused at request time, requiring the use of short-term adaptive algorithms. The scheduler tries to respect periodic CPU reservations *and* meet as many time constraints as possible. Time constraints are expressed as follows:

```
feasible = BeginConstraint(start_earliest, est_cpu, deadline, critical);
...
time_taken = EndConstraint();
```

3.10.4 Exokernel

Exokernel is an OS architecture developed at MIT [Kaashoek95, Kaashoek97]. It consists of a very small kernel with functionality limited to the allocation and multiplexing of physical resources. Traditional OS abstractions are provided by shared *Library Operating Systems* which operate in user-space. The intended benefits are performance (via specialisation) and flexibility (policies can be tailored on a per-application basis).

The Exokernel scheduler is based on a cycle of slots with a static schedule. It is doubtful whether a scheduler can really be policy-free, although it does provide hooks to execute custom code before and after preemption points.

There are a number of problems with the Exokernel structure. Protecting resources without understanding how policy decisions are made requires techniques such as caching frequently used access control information in the kernel, to avoid harming performance.

Library OS's cannot trust each other, so it is difficult to share objects safely between them. There is a danger that a multitude of incompatible interfaces could arise. Tests indicate that an Exokernel system performs little better than UNIX, except on filesystem intensive applications. Finally, portability is affected since applications must be relinked if the Library OS is updated.

3.11 Real Time Linux variants

A number of existing real time systems have been based on the Linux platform.

3.11.1 RT Linux

RT Linux [Yodaiken97] takes the form of a wrapper for the Linux OS. It provides a minimal RT kernel, which runs the Linux kernel itself as a subtask when no RT task requires the processor.

If the Linux kernel calls the disable interrupts function, RT Linux emulates this with a software flag, instead of disabling the actual hardware interrupts. This allows RT tasks to be scheduled at a fine granularity without the delays incurred by disabling interrupts for long periods of time.

RT tasks do not have access to Linux syscalls, or any higher level services and APIs such as graphics and networking. Consequently, applications have to be split into a RT part (which uses almost no services) and a non-RT part (which has no timing requirements). Communication between the two is possible via message passing and shared memory.

RT Linux has been used successfully for process and experiment control, monitoring and robotics. One benefit of RT Linux over a conventional RTOS is the familiar development environment. However, it is not suitable for general purpose RT computing due to the restrictions placed on RT tasks.

3.11.2 KURT

The KURT system [Srinivasan98] is also based on Linux, but is better integrated, allowing RT processes to use normal kernel services. It makes use of two convenient kernel patches. The first patch is UTIME, which was discussed in Section 3.2. The second patch provides better support for POSIX.1b by making the improved timers accessible from user space via system calls.

KURT itself uses an explicit plan scheduler based on application schedule files. Although it does have a "periodic mode", applications must cooperate by calling KURT each period, requiring source-level modifications.

KURT can run in two modes: 'focussed' or 'mixed'. In focussed mode only RT processes are allowed to run, which helps reduce kernel crosstalk from the BE applications at the expense of a more limited system.

3.11.3 QLinux

QLinux uses a hierarchical version of SFQ [Goyal96], as described in Section 1.2.3.

It also provides a SFQ network packet scheduler and lazy receiver processing, in order to charge protocol processing to the correct task. A multi-service disk scheduling algorithm is included as well.

Chapter 4

Inter-Process Communication

A desktop environment (real time or otherwise) requires close integration between several distinct components. These include the kernel, X server, window manager, control panels and applications.

One of the strengths of UNIX is that each of these pieces of software is developed independently, and they interoperate though well defined, published interfaces. This benefits stability and allows the use of code from different vendors.

A disadvantage is that it is harder to share data, and some of the internal interfaces are not rich enough to express the interaction present in a modern desktop environment. There is a particular need for an event mechanism, to provide synchronisation for real time components.

Projects such as GNOME and KDE [Weis98] have responded to the problem, using CORBA [OMG95a] to tie components together. This is a significant improvement but still does not meet all our requirements.

4.1 Existing IPC mechanisms

Even if we focus on UNIX, a considerable range of IPC primitives are in common use.

Existing mechanisms include Unnamed pipes, Named pipes (FIFOs), UNIX domain sockets, TCP/IP sockets, File locking, System V and POSIX message passing, shared memory and semaphores, original and POSIX.1b signals.

Note that POSIX enhanced message passing, shared memory and semaphores as well as POSIX.1b signals are not currently available under Linux.

Higher level mechanisms include RPC, RMI, CORBA, and the CORBA services.

4.2 Properties of an IPC system

In this section we examine desirable properties for an IPC system. We concentrate on an events based model. UNIX does not have a standard, general-purpose event registration and notification system. This has resulted in a proliferation of specialised event systems, particularly in graphics toolkits.

The optional CORBA event service [OMG95b] has been adopted by some projects to solve this problem, although this lacks some features such as event filtering.

The Cambridge Event Architecture [Bates98] provides a more complete facility based on the publish, register and notify paradigm. Filtered, typed messages can be delivered.

4.2.1 Speed

There is a clear distinction between bulk data streams and synchronisation mechanisms. The use of sockets for synchronisation is not ideal because of the high overhead, so we require a more lightweight event system which complements the existing means of data transfer.

A balance can be met between efficiency and functionality by offering different stream “weights” (see Table 4.1 later in this chapter). Whether the server is implemented in the kernel or user space has considerable bearing on efficiency as well (see Section 4.4.1).

4.2.2 Kernel event sources

Many events of interest occur within the kernel, with little relevance to the thread which happens to be running. For example, we would like to be notified when a process changes to a different scheduling policy (the window manager uses this information to update a graphical display of real time tasks). This can occur either due to an explicit request or an automatic QOS management action.

The IPC system must therefore support events which originate from the kernel itself. This differs from mechanisms such as sockets which are expected to have two user-space endpoints.

4.2.3 Multicasting

All the primitive IPC mechanisms on UNIX assume messages are “consumed” by a single listener. We require instead that messages be distributed to any number of tasks which have registered an interest in that event type.

Another useful extension allows multiple tasks to fire a single event source. For example, an application can create an event source, listen to it and accept commands fired at it, thereby gaining language-independent scripting capabilities.

Recent work has included composite event detection [Bates98].

4.2.4 Namespace

Communication endpoints require unique and preferably meaningful names. Some mechanisms use numeric namespaces; for example IP ports, signal numbers or System V "keys". These are unattractive because they require the use of further protocols to agree on which numbers to use.

A better solution is to use hierarchical, ASCII names. These should appear somewhere within the filesystem, in order to minimize new abstractions. This conforms with the UNIX principle that most entities can be manipulated through file I/O operations.

Note that namespaces embedded within a file are unwise, because the standard tools that operate on files and directories will not work. An example of this is the Win32 registry. In order to search or selectively extract information, the usual filesystem tools must be duplicated with special embedded versions.

A disadvantage of using the filesystem is that the event queue names may then be written to disk. This occurs with UNIX domain sockets, named pipes and lock files. It is inelegant for a memory-based concept, which should not be persistent. In the event of a serious crash these files might not be cleaned up and could prevent servers from restarting properly.

Fortunately, Linux provides the `/proc` filesystem, which allows memory-based filesystem objects.

Conventions and file permissions are needed to keep the new namespace orderly. Suitable pathnames include `/proc/events/user/brian/app/netscape`, or `/proc/events/gui/mousedown`.

4.2.5 Asynchrony

Events can be received by polling, waiting or through callbacks. The ability to poll is generally useful and is possible with all the IPC mechanisms we have considered.

Callbacks are provided by signals, and are useful for major interruptions, such as restarting a process. They are less suitable for general-purpose IPC, because the asynchronous behaviour is harder to integrate with a main event loop and requires carefully placed semaphores to protect data structures.

When waiting for events it is usually necessary to wait for several sources at once. The normal primitive to wait for a set of events under UNIX is `select()`, which operates on file descriptors. This is very common because sockets and pipes are handled using file descriptors.

There are other functions such as `sigsuspend()`, which is used to wait for a signal. However it is not possible for a single thread to wait on more than one of these primitives, and application designers are reluctant to spawn multiple threads to work around blocked I/O. Compatibility with `select()` will therefore be a key requirement for any new IPC mechanism.

4.2.6 Authentication

Current IPC mechanisms do not provide integral support for authentication. In a few cases there is a fixed, implicit notion of peer identity—signals cannot come from another user's process and unnamed pipes rely on sharing file descriptors explicitly with another process.

Without authentication, servers can adopt only a few policies. Often it will be possible to accept all requests anonymously. Sockets allow one to verify that the peer is running as root if it is using a reserved port number, and this is the basis for somewhat weak security in `rlogin` and NFS. The remaining option is to defer authentication to the application. This must be based on tokens which can be passed down the pipe itself, for example passwords typed by a user or a cryptographic protocol.

The ability to look up identities is essential for servers which need to prioritise client requests according to scheduling parameters or other status information. Without this, it is not possible to automatically restrict service to a particular group of users, for example.

Flexible event sources need separate access control bits for firing sources and for listening to them. In some cases these rights must be restricted to the task which created the source, in other cases we wish to allow connections based on UID or GID.

Remote authentication services can be built on top of any suitable local-case equivalent. For example, `identd` (specified in RFC 1413) maps remote TCP connections to user names (but cannot name individual processes).

Under Linux, utilities such as `identd`, `pidof` and `fuser` use the `/proc` filesystem to identify peers. This is extremely slow for several reasons. Firstly, the data must be converted from binary to ASCII and back again. Secondly, whilst it is easy to discover the sockets held by a given client, the system is not designed to answer the reverse query. A linear search is therefore made through the list of all sockets on the system.

Furthermore, this only returns the UID of the peer task. In cases where it is necessary to identify the process itself, the lists associated with every task are scanned. The `fuser` command takes over 0.3 seconds to do this (on a Pentium 133 with 48 mostly-idle processes). We therefore need a more efficient way to discover client TID's for local connections.

4.2.7 Remote access

Of the IPC mechanisms studied, only TCP/IP sockets and higher level protocols support remote access. The use of IP as the underlying protocol is inevitable due to the necessity of Internet compatability. A remote access events system must therefore be layered, perhaps built on CORBA. Transparency is desirable, for example when using a remote X server.

4.2.8 Decoupling source and sink

Most event systems do not require a source to be aware of its listeners. However, a source must exist before you can listen to it. This restriction can be relaxed too, so that clients may listen to named sources even if they have not been created yet.

Should someone later create an event source with this name, those clients which pre-registered immediately start getting events of this type delivered along with the others. Furthermore, listeners are not necessarily deregistered if the event source task terminates.

The benefits are that servers do not have to be started before clients, and it isn't recessary to re-attach the clients if a server has to be restarted.

Decoupling can be implemented using a "shadow" event source structure, which points to a linked list of listeners as usual.

4.3 Applications using events

An events system capable of multicasting to arbitrary processes has many applications within the desktop environment. For example:

- Desktop environments, control panels, window systems, GUIs, DnD
- Disk change events
- Resource alerts (disk space etc.)
- Screen resolution, palette or font preference changes
- Audio volume change events
- IDEs (run compiler, highlight errors in editor)
- Timer events
- Applications, daemons or sessions starting and exiting
- Printer events (offline, out of paper)
- Remote control, CSCW, scripting and logging

4.4 Designing a new IPC mechanism

This section covers the design decisions which influenced the form of Linux-SRT's enhanced IPC facilities.

4.4.1 Kernel and user-space implementations

Sockets already provide most of the necessary attributes for our IPC system, including inter-host communication. Although sockets lack a multicast capability, they do provide the means to construct it using a user-space event server process listening on a well-known port. One such server is needed per host. Each client may listen to several event servers, and register interest in various streams per connection.

A user-space event server has two disadvantages: much higher event delivery overhead and the inability to export kernel event sources.

We therefore decided to separate the design into two distinct mechanisms: a mandatory kernel subsystem, and optional user-space event servers. The kernel event system is restricted to kernel sources only, and does not allow the registration of user-space sources. To do so *would* increase efficiency and the convenience of multicasting, at the expense of increased kernel complexity and a local-remote case distinction. Kernel complexity is a particularly serious concern due to the relative difficulty of maintaining it and the need for extreme reliability. Complex subsystems such as CORBA certainly do not belong in the kernel.

The kernel events device is restricted to one stream per file descriptor. It takes a long time to `select()` on hundreds of file descriptors (represented by a bit array) so this does not scale as well as a composite event queue. There is also a limit of 256 file descriptors per process (which can be raised to the `select()` set size of 1024). However we avoid duplicating mechanisms and feel that it is better to make `select()` more efficient than to use two levels of multiplexing.

The kernel events device could defer multicast abilities to the user space events server as well, having a dedicated upstream connection from which kernel events are delivered and relayed on with the others. We have chosen to provide multicast from the kernel, however. This allows kernel events to be safely used directly without a canonical user-space daemon.

A user-space event server can be implemented by communicating with different event sources and sinks (and with a kernel event uplink) over different file descriptors.

4.4.2 Heavy and lightweight streams

Several possibilities for stream weight were evaluated (see Table 4.1).

The initial implementation used “pure” events, which are limited to notification only. Other means can be used to retrieve the parameters or changed data itself. Although this is simple it results in more work being done in subsequent lookups.

Using lightweight events instead, we can for example specify that a *specific* task has changed scheduling class, avoiding a great deal of inefficiency checking all tasks to decide what has changed.

Event type	Description
STREAM_PURE	Payload: none Implementation: occurrence counter (value can be read to distinguish rapid bunched events).
STREAM_LIGHT	Payload: 32 bits Implementation: slots in an expandible circular array.
STREAM_HEAVY	Payload: variable length, with timestamp Implementation: Queues contain pointers to messages. Buffers allocated from free store, with reference counts.

Table 4.1: Stream weights

4.5 Kernel Events

We have implemented an IPC mechanism which provides notification of internal kernel events to user-space clients. The first five properties described in Section 4.2 are satisfied by this system. It uses lightweight events and is designed for high performance. Because it is a device it can be opened with normal file access commands and waited for using `select()`. The device name in the filesystem is `/dev/events`.

It is implemented as part of the "memory" device and therefore shares the same major device number as `/dev/zero`, `/dev/null`, `/dev/random` and so on. It is identified uniquely by its minor device number.

4.5.1 Naming

Event sources are identified by text strings. The kernel currently registers the sources listed in Table 4.2. A list of the event sources currently registered can be browsed in `/proc/events`.

Stream name	Description	Payload
sched-param	Scheduling parameters changed	PID
sys-exec	Occurs when a task performs an exec	PID
sched-overrun	Indicates when policing is applied/removed	PID or -PID
user	User events for synchronisation	Event type

Table 4.2: Standard event sources

Figure 4.1 shows how kernel events are used by X11 and the window manager. Both receive notification of scheduling class changes using the kernel events multicast capability. X11 also needs policing information. Each type of kernel event is carried on a different file descriptor, and `select()` is used to multiplex all IPC sources which the tasks wait for. Note: the timeouts are used by X11 for the screensaver and by the window manager to implement "autoraise".

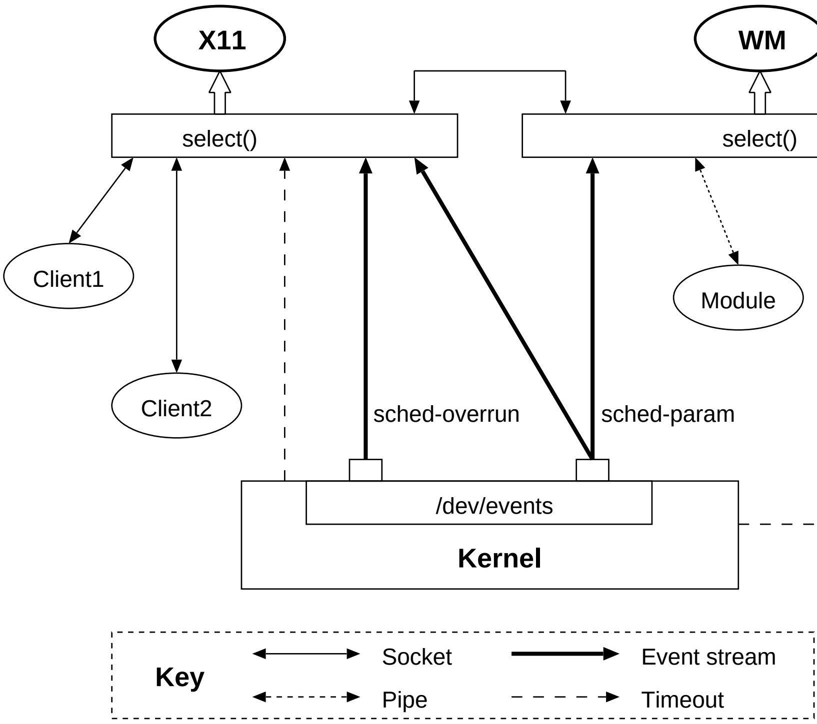


Figure 4.1: Kernel event usage

4.5.2 Interface

The kernel events system presents its API using device calls, and also communicates with the kernel in a well defined way.

External device interface

Several methods from the device driver virtual function table [Rubini98] are used.

The device implements `open()` and `release()` to keep track of its clients. It can be opened by multiple tasks, and also more than once per client. If a client process terminates, the release method is called when the associated file descriptor is destroyed, ensuring that clients are always deregistered correctly.

The device accepts a single `ioctl()` request as follows:

```
bool ioctl(int fd, EVT_IOCTL_BIND, char *source_name)
```

This is used to bind a particular connection to a named event source.

The `read()` operation returns the next message due to be delivered to the client in question, and `select()` determines whether messages are currently available.

Internal kernel interface

The following two functions are made available to the rest of the kernel:

```
struct source *create_event_source(char *stream-name)
void fire_event(struct source *src, int payload)
```

These can be called from the main kernel or from dynamically loaded kernel modules. This interface makes it easy to add new event sources without changing the event system itself.

4.5.3 Implementation

Each client has a circular buffer that initially has room for 5 messages (20 bytes). When it becomes full it is expanded to double size. There is a failsafe limit of 1280 pending messages per client, after which new events are thrown away.

The main data structures are as follows:

```
struct source
{
   struct source *next, *prev; /* list of all sources */
   struct listener *listener_list;
   struct wait_queue *wq;
   char name[0];
}

struct listener
{
   struct client *ptr;
   struct listener *next, *prev;
}

struct client
{
   struct source *src;
   int *buffer;
   int buffer_capacity, next_msg_slot, queue_len;
}
```

A pointer to the `client` structure is stored in the private data field for each `file` structure manipulated by `/dev/events`.

4.6 Authenticated sockets

An extension to the socket IPC mechanism has been developed, which identifies the peer task across such a connection. This allows servers to make access control decisions and is also the basis for QOS transfer from client to server.

As was noted in Section 4.2.6, there is normally no efficient way to do this. The source of the problem is that there is no back pointer from the socket (or matching inode) structure to the open file structure.[1]

Authenticated sockets solve this problem, and make the information available in several ways. The `sockpeerinfo()` system call is most efficient, but the information can also be retrieved from extended versions of `/proc/net/tcp`, `/proc/net/udp` and `/proc/net/unix`.

The new facility maps a socket onto the peer's TID, so in particular the target PID can be determined. This may be used to look up other information such as the user name, groups and reserve held by the peer.

Authenticated sockets work with TCP or UDP sockets which are connected to a local endpoint, and with UNIX domain sockets.

Remote TCP connections are not handled directly, although this could be added with an identification server similar to `identd`, returning TID's in addition to user names.

Reverse lookups always return the process which *created* the socket (although it may share its file descriptor with other processes). If the original process has terminated, an error code is returned. TIDs are never reused, so this can always be determined.

4.7 Related work

The kernel events system is similar to Alan Cox's *netlink* device. The main difference is that netlink is stream orientated, allowing 2-way exchange of arbitrary data; whereas kernel events emit fixed size messages. Netlink messages are consumed when read, instead of being multicast to listeners. Netlink also uses the minor device number to distinguish streams, but kernel events employ a single device number with a human-readable namespace.

[1] In certain cases a back pointer does exist, for example when garbage collecting UNIX domain sockets or if an asynchronous notification list has been set up for SIGIO.

Chapter 5

Scheduling Servers

When applications rely on other processes to provide services, quality of service guarantees for the applications may fail to be met as a result. In this chapter we explore ways of maintaining quality of service across task boundaries and describe a solution which works well with existing client-server systems.

5.1 Operating System Structures

The way in which access to services is provided by the platform determines the QOS transfer technique which will be appropriate. Here we identify four main operating system structures. In each case we must account for all CPU usage, including overheads, and charge it to the most appropriate party.

5.1.1 Single domain

Older operating systems, or those running on embedded hardware with no MMU, employ a single protection domain. System services are supplied by library routines.

It is easy to provide QOS in this environment (assuming the OS supports preemptive scheduling), because services run in the same thread as the application.

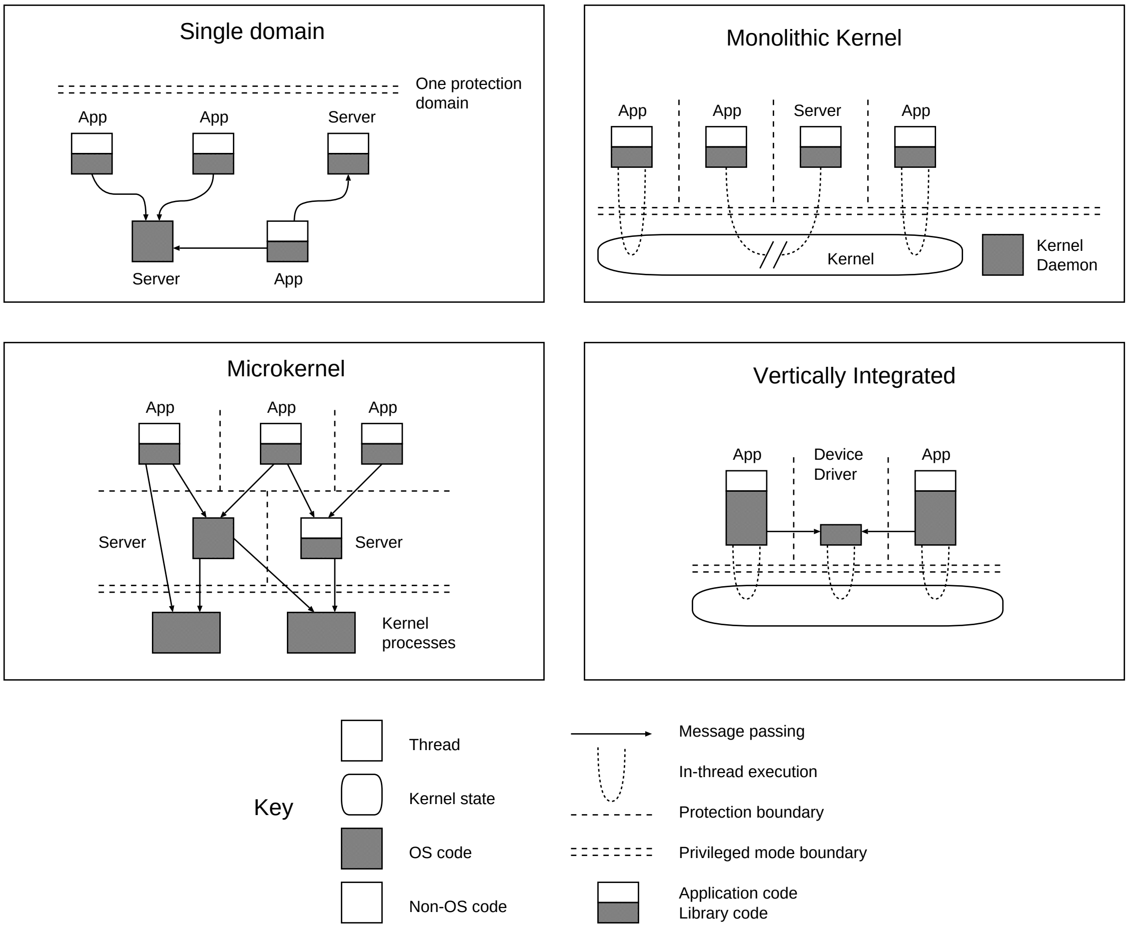
Such systems are now rare, however, due to their lack of robustness and inability to support multiple users safely.

5.1.2 Monolithic Kernel

Traditional UNIX operating systems are based around a fairly large privileged kernel, which is entered via trap instructions. Multiple protected address spaces are supported.

Services supplied by kernel traps are executed in the application's thread just as library routines are—so it is again clear which task to attribute the CPU usage to. However, it is hard to provide accurate QOS guarantees, because processes running in the kernel cannot be preempted at will.

Figure 5.1: Operating System Structures



5.1.3 Microkernel

Microkernel operating systems such as Mach have small kernels which provide scheduling, memory management and message passing. Additional OS functionality (file systems, networking and graphics) is deferred to user-level server processes.

QOS reserves must be passed between clients and servers, in order to correctly attribute work done to the client process.

5.1.4 Vertically integrated

A more recently proposed structure is used by the Nemesis [Leslie96] operating system. The vertically integrated architecture is again based on a small kernel. In addition, the work done by servers is limited to initial access control only. Once a client has been authenticated, common data path operations are performed in the client's thread by library routines. For example, rendering code can be moved from the X server into the X library.

In this case there is no need to transfer reserves between threads.

Protection between clients is more difficult with this model, unless custom hardware is used. The access control server for a particular resource may avoid this by installing trusted code or packet filtering into the device driver at connection establishment time.

Another drawback is that common client-server software must be restructured to run on such a system, making porting difficult.

5.2 Client-Server Architectures

One of our objectives is to support soft real time scheduling on general purpose client-server systems. Now, shared servers cause inaccurate scheduling whenever their clients have different scheduling requirements (which is *always* the case, even for a BE dynamic priority system).

There are several possible ways to ensure that servers respect priority distinctions:

(a) Eliminate the server

(b) Use a migratory thread implementation

(c) Make the server multi-threaded

(d) Perform local scheduling within a single-threaded server

Note that all of these require some kind of modification to the server itself; we can't apply these techniques without access to the source code.

5.2.1 Migratory Threads

The principle behind migratory threads is to allow execution domains (threads) to cross to other protection domains (address spaces) and back again. A client can thus make a request to a server, execute server code and receive a response without performing a full reschedule.

Arguments are passed on a stack provided by the client, and the context switch overhead is that which is involved with reloading the virtual memory registers. To clients, the mechanism resembles an extension of kernel traps to an arbitrary number of levels or server domains.

Migratory threads have been implemented on DEC workstations by the Lightweight RPC system at the University of Washington [Anderson90]. A similar idea is that of *paths* as found in the Scout operating system [Mosberger96]. Paths allow scheduling properties to be associated with data flows rather than with processes.

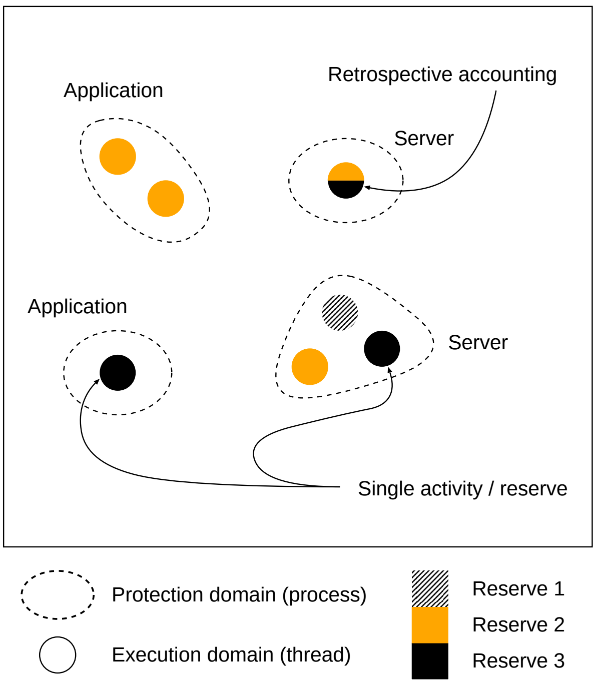
These architectures solve the problem of accounting for client-server interactions, combining the client thread processing attributes of a vertically integrated system with the strong protection offered by a microkernel. Unfortunately there are no widely used systems based on this.

5.3 Multi-threaded servers

If the server process is multi-threaded, resource usage can be accounted to clients correctly using Linux-SRT's reserve abstraction (see Section 3.7). QOS contracts are allocated to reserves instead of individual threads, so reserves form scheduling domains and threads correspond to execution domains. The threads belonging to a particular reserve will normally be cooperating in order to achieve some goal, for example a real time UNIX pipe.

In the case of a multi-threaded server, we can assign the server thread working for a particular client to the same reserve as the client itself. Reserves natually span several protection domains in this case, and all the work done for a particular client is charged to the same resource allocation. Figure 5.2 illustrates these relationships.

Figure 5.2: Servers and Reserves



The Linux-SRT implementation is based on a reserve structure which contains a linked list of subscribed tasks. A system-wide list links all the reserves together so that they can be refreshed periodically without reference to the processes themselves.

5.3.1 Joining reserves

Threads may join a reserve in several ways:

(a) Any application may make a `join_reserve()` syscall to coordinate its own threads.

(b) Child processes belong to the same reserve as parents if the inherited scheduling flag is set for the parent.

(c) Servers may make an explicit `join_reserve()` request in order to associate one of its worker threads with a new client.

(d) The connection can be made automatically by the IPC mechanism whenever another process is contacted.

The Processor Capacity Reserves [Mercer94] system implements reserve sharing between server and client threads based on a modified RPC primitive. Linux-SRT currently makes use of the first 3 techniques.

5.3.2 Scheduling within a reserve

In some systems the client reserve may be "handed over" to the server for the duration of the call. This is quite effective but restricts us to blocking IPC only.

A more general approach is to allow multiple threads to tap a shared reserve simultaneously. In this case there must be a policy for scheduling threads *within* the reserve. Possible solutions include the following:

(a) Proportional share. This is easily achieved by subdividing the original reserve. Whilst it does allow the client to continue processing we have no way of knowing what proportion to use.

(b) FCFS. This will occur if we don't perform preemptive scheduling between the threads at all. It's a reasonable approach given that the threads are supposed to be working on the same problem and will presumably block when they need data from the others.

(c) Fair share. The Rialto [Jones96] operating system uses this policy between threads in an activity. It is more flexible than the first two policies.

(d) Dynamic priority. This is the approach used by Linux-SRT, because it is already supported by the BE kernel scheduler.

5.4 Single-threaded servers

Although multi-threaded servers work well, they are not always appropriate. If a server handles many short requests, a single-threaded version is likely to be more efficient. Multi-threaded servers incur overhead performing IPC between their threads, even if a suitable set of worker threads are forked in advance. Many existing servers are single-threaded.

We therefore need a way to schedule RT clients of a single-threaded server, and attribute the work done on their behalf to the proper client reserves. A single-threaded server cannot switch between different reserves, since it would then be completely blocked when a client reserve it was using ran out.

Example — X11

XFree86 is an example of a single-threaded server with very high processor usage (about 40% during MPEG video playback on a desktop machine). Throughout this section we will use X to illustrate our techniques for handling single-threaded RT servers. This is appropriate both as an example and because it is in practice the main obstacle to real time scheduling on the desktop (see Section 2.6.3 for server CPU usage figures).

To test the effect of using the X server on scheduling guarantees, we started several video player applications simultaneously and assigned one of them a static real time priority level. The real time player was observed to achieve exactly the same frame rate as the others.

The reason for this is that the real time task requests graphics rendering and then blocks waiting for the server's response. The X server typically has the lowest dynamic priority (being the busiest task in the system) and hence is not scheduled until all the other tasks have run, and put their own X requests into the queue. The graphical tasks are thus effectively scheduled on a round-robin basis irrespective of priority.

5.4.1 Master-Slave scheduling model

Ideally we would like to multiplex each resource (such as the CPU) in only one place—in this case the kernel scheduler. Where this is not possible, it is necessary to synchronise the scheduling decisions made throughout the system, so that applications will have access to all the resources they need in order to make progress during their time slice.

We have developed a master-slave system to perform this synchronisation. In our system the kernel is the "master" scheduler, and the single threaded servers are slaves. The interactions between master and slave are illustrated in Figure 5.3.

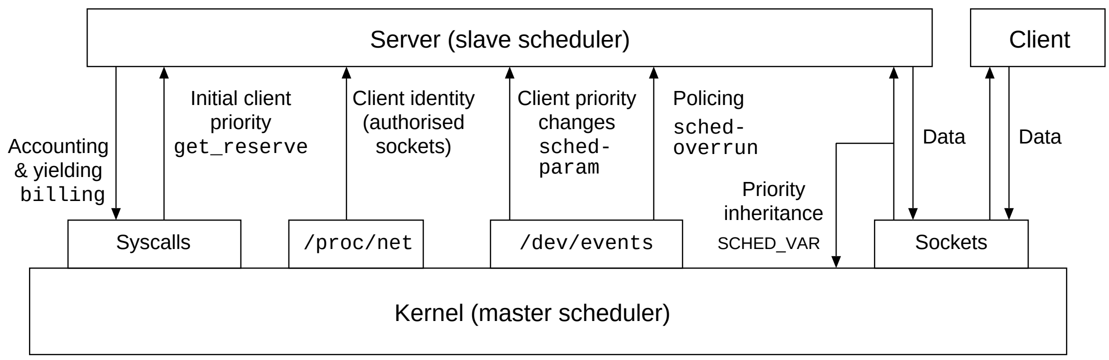


Figure 5.3: Master-slave scheduling data-paths

Example — X11

We have applied our master-slave scheduling model to the X server. The implementation consists of modifications to the platform independent parts of the X server code [Angebranndt94]. Some additional requests were incorporated into the X protocol [Scheifler94] and Xlib [Gettys96] for control and test purposes. This modified X server was first described in a paper by the author [Ingram99]. More details are given below.

5.4.2 Connection establishment

When a new client connects to a RT server, the server must determine the client's identity, scheduling class and priority. This can be done using authenticated sockets (see Section 4.6).

Example — X11

Figure 5.4 shows the sequence of events which occur when a new X client connects to the server. This is complicated by the fact that the X server normally does not know which process corresponds to each client; once they are authenticated it treats them all equally. The authenticated socket mechanism is used to provide this information, by reading an extra field in `/proc/net/tcp` which specifies the TID.

1. The server begins with a file descriptor for the new connection.
2. `getpeername()` is called to discover the socket address of the client.
3. The server checks this is an address for a *local* TCP/IP or UNIX connection, and extracts the port field.
4. The `/proc` filesystem is read to discover which process opened this connection.
5. `sched_query_reserve()` and `get_reserve()` are called to look up the scheduling parameters for the client.

Figure 5.4: New X connections

5.4.3 Prioritising clients

Once the client priority has been established, requests from different clients can be prioritised by the server.

The `sched-param` kernel event is used to notify the server of any subsequent changes to the scheduling parameters; this can be monitored by reading the events device in the server's `select()` loop. Because such changes are infrequent events, handling them has little effect on performance.

Example — X11

The X protocol reference manual [Scheifler94] remarks that:

> "Whether or not a server is implemented with internal concurrency, the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams)."

Therefore, although it is not possible to interrupt a request part way through, we may reorder the request queue to favour particular clients whilst meeting the requirements of the existing protocol.

The Linux-SRT X server thus handles requests according to client priority, using a priority queue structure. The mechanism is similar to that used by the X Synchronization Extension [Glauert91].

5.4.4 Retrospective accounting

The method we have chosen to properly account for CPU time used by the server is to charge client reserves retrospectively.

Since the server is assumed to be single-threaded, it cannot switch clients whilst processing a request. The granularity at which scheduling occurs is thus determined by the request service times. Reserves are debited after each request, and may drop below zero as a result.

Retrospective accounting contrasts with a pure QOS system which can preempt at any instant to achieve precise CPU allocation. A UNIX best effort (dynamic priority) or fair-share scheduler is also retrospective, because it adjusts priorities in accordance with recent CPU usage, to achieve the desired CPU allocation.

Timescales differ considerably, though. Linux recalculates priorities around five times per second. Timing data for single-threaded servers shows that typical requests take only a few microseconds to process. This means that servers can react very quickly to clients which overrun their reserve. Response times can be guaranteed by augmenting reserves slightly to allow for this lag. Retrospective accounting is therefore suitable for RT scheduling when performed at suitably short time-scales.

It should be noted in passing that retrospective accounting is also applicable to scheduling operating system activities (although we do not cover this). Incoming network packet processing may require significant work before we know which process to charge, and makes it difficult to schedule proactively.

Example — X11

X request service times were measured in order to assess the accuracy of the retrospective accounting system. The mean service time was 6μs. Process scheduling intervals are therefore much longer than X requests. For example, given 10 real time tasks running at 70 Hz, a typical X request takes 0.4% of a time-slice. Retrospective accounting is therefore likely to work well; the lag between kernel and server schedulers will be small.

Figure 5.5 investigates the tail of the distribution. Logarithmic observed frequencies are plotted against request length. 99.6% of requests take less than 200μs. Some requests take over 1ms but are less common (about 1 request in 2870). No request was observed to take more than 3ms on a Pentium 133. The X server used is non-accelerated, so rendering for this test is done entirely by the CPU and not by the draw engine on the graphics card.

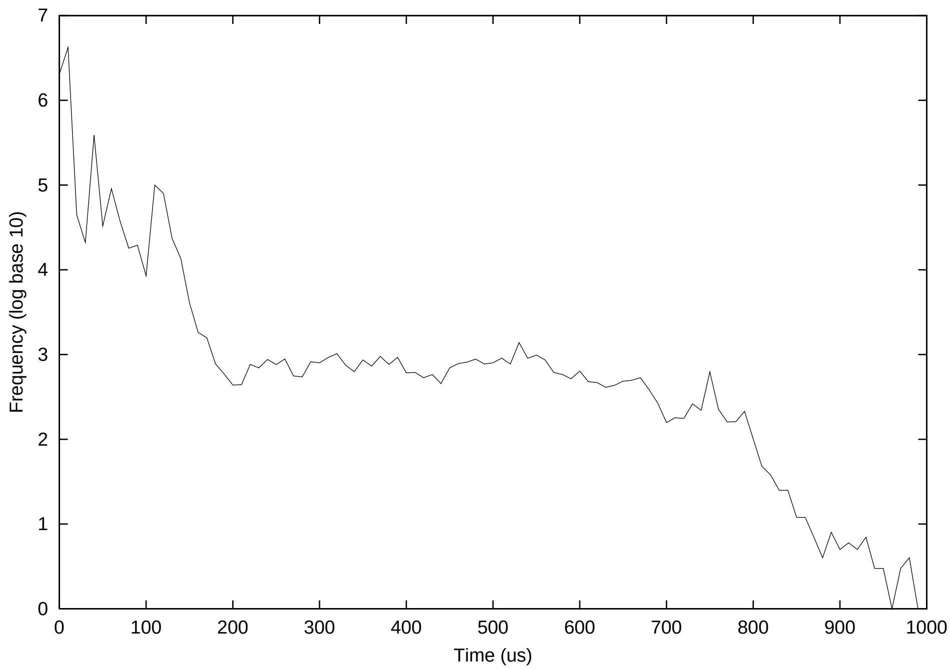


Figure 5.5: X Request Service Times

5.4.5 Billing system call

The slave scheduler passes accounting information to the kernel with a new `billing()` system call. This measures the amount of CPU time which the calling process has consumed since the last time it called for a checkpoint, to microsecond accuracy.

The standard timing mechanisms cannot supply this information. Calling `gettimeofday()` to report wall clock time within the server would not take into account times when the server is preempted. The `times()` system call reports CPU usage instead but has a resolution of at best 10ms, because it works by polling the active task at each timer interrupt. `billing()` is implemented by calling `do_gettimeofday()` within the scheduler and thus does not suffer from these problems. The accumulated time is noted whenever a task switch occurs.

The call synopsis is as follows:

`int billing(`*reset*`,` *pid*`,` *yield*`);`

Return value: the CPU time used by the process making the call since the last reset, in microseconds.

reset: specifies whether to reset the elapsed time to zero.

pid: if non-zero, the time value returned is charged by the kernel to the process specified by *pid* (typically the server calls `billing()`, and *pid* identifies one of its clients). You should always set *reset* to 1 in this case as well, so that CPU time is never charged for twice. The kernel uses a PID to `task struct` hash table to find the process quickly.

yield: only applicable when the process making the `billing()` call has scheduling policy `SCHED_VAR`; this returns it from a high to a normal priority (see Section 5.4.7 on Priority Inheritance).

Example — X11

The X server uses the `billing()` call to return accounting information to the kernel, so that graphics processing is charged to the correct client's reserve. The server also spends time reading from sockets and performing general book-keeping. This is charged to the next client in turn, so that overheads are shared in approximate proportion to the number of requests made.

In practice it is unnecessary to report accounting information after every request, so this is done in batches to reduce overhead.

Originally the `billing()` call passed a list of (*client*, *CPU usage*) pairs, and this was done at one of three times: (i) when the server was about to go idle, (ii) when all RT clients had been processed and (iii) periodically (between requests, every 500μs—a time chosen to be less than a tick). The cost of managing the linked list was found to exceed the time saved in reducing system calls, though. The call was therefore changed to the current version, which only reports on one client's usage. `billing()` must be called more frequently, but runs quickly, with less complexity in the server.

Previous client		Next client	Action
BE client	→	Another BE client	No special action.
BE client	→	QOS client	`billing(1,0,0);`
QOS client	→	Another QOS client	`billing(1,pid,0);`
QOS client	→	BE client	`billing(1,pid,1);`

Table 5.1: X server actions between clients

The `billing()` calls now made by the X server when switching between clients are shown by Table 5.1. Note that `billing()` does not have to be called between successive requests from the same client (and to do so would introduce unacceptable overhead).

5.4.6 Policing

Linux-SRT uses rate monotonic scheduling, so process priorities are static. The only state which needs to be communicated regularly from the master scheduler to the slave is therefore the times at which a client has overrun its time contract, or resumed normal scheduling. Clients with exhausted reserves should not be selected for processing by servers.

In our initial implementation, the X server polled the kernel between requests, returning this information with a system call. This has been replaced by the kernel events system, which notifies the server of policing and other scheduling events, and is much more efficient.

Policing events are generated by the `sched-overrun` source. The parameter specifies the PID in question (positive for policing, negative for reactivation). One event is sent out for each process in the reserve which is effected.

5.4.7 Priority inheritance

In any priority-based environment, a situation in which a high priority process has to wait for a low priority one is known as *priority inversion.* This may occur due to contention for any shared resource, such as a semaphore or server. Although undesirable, some degree of priority inversion is unavoidable if access to the resource must be serialised.

In some situations a queue of clients may result in priority inversion for an unbounded length of time. The well known priority inheritance protocol (PIP) [Sha90] is used to prevent this. Generalising, we can identify four types of PIP, shown in Table 5.2. Multi-threaded servers are easier because the PIP occurs exactly once at connection establishment time for each client.

Server	Static priority clients	QOS clients
Single threaded	The server is elevated to the priority of the highest priority client currently waiting	Server uses `SCHED_VAR` scheduling class (see below)
Multi threaded	Server threads have the same priority as their respective clients	Server threads join client reserves

Table 5.2: Priority inheritance techniques

If PIP is built into IPC mechanisms, we need to determine when a client is waiting for service and when it has finished being serviced. This necessitates different approaches for RPC, socket protocols and other types of communication.

The SCHED_VAR scheduling class

To solve the priority inheritance problem for RT servers, we use a new scheduling class called SCHED_VAR (variable priority). This has two states, in which it has an effective priority of either zero or 200. The higher priority exceeds any static priority (the maximum under Linux is 99) and is below than the range of priorities used by SCHED_QOS.

UNIX sockets connected to a SCHED_VAR process perform an additional check when data is sent to them. If the sending task has QOS available, the peer process is switched to high priority mode. There is no immediate reschedule because it is better to wait until the sending client has blocked, so it can batch several requests together.

The server must switch back to the low priority mode explicitly when changing from QOS to BE client request processing (using the `billing()` call). The kernel may then reschedule the server as for any SCHED_OTHER task.

The choice of the fixed priority level 200 was made for simplicity and to reduce communication overhead. The effect is that RT client-server processing is performed in a lazy fashion; all possible guaranteed work is done in the clients first. This reduces unnecessary context switching without missing deadlines.

5.4.8 Results

In order to test the modified X server, a number of video players were run simultaneously (we used `xanim` together with a short Quicktime movie clip). The time for the movie to play to completion was noted. With a small number of players full frame rate was achieved so the duration was always the same, but with many players the movie took longer to complete as the frame rate slowed down (our players did not skip frames).

The experiment was then repeated with one video player flagged for real time server scheduling. The performance of the boosted application and of the others was measured. In contrast to the use of real time kernel scheduling alone, making X aware of its clients' priorities has a significant effect (see Table 5.3).

Number of concurrent players	All treated equally	Prioritised player	Not prioritised
1	12s	12s	12s
4	12s	12s	12s
8	24s	12s	29s
12	37s	12s	46s
16	50s	12s	64s

Table 5.3: Times to play video clip (in seconds)

This experiment illustrates how one could keep the frame rate high on the current speaker during a video-conference, or on the channel of interest when receiving TV or video.

Chapter 6

Adaptation

Adaptation can be used to determine unknown scheduling parameters, and adjust to gradually varying requirements. Although our prototype does not perform adaptation, it is likely to be a necessary component of any completely automated QOS manager. Adaptation helps free users from the need to understand or configure QOS. We therefore consider the technique in some detail.

An adaptive scheduler observes tasks as they run, measuring CPU usage and deducing their timing requirements. These values can then be guaranteed for future runs.

Regular periodic tasks often have spikes of activity due to sporadic external events. We can use probabilistic adaptation to filter these out. It is important that we quantify this ("probabilistic guarantees" have been suggested in some cases with no way of discovering the actual probability of missing a deadline). A suitable measure is the greatest acceptable chance of an overrun event, or equivalently the mean inter-failure time.

6.1 Workload classification

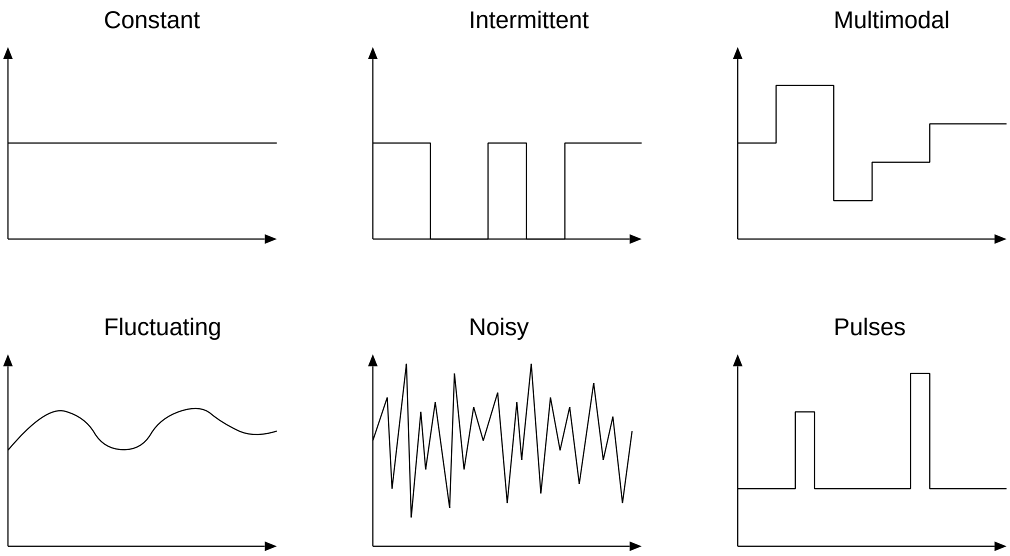
Before considering adaptive algorithms it is necessary to understand how CPU usage varies over time for different applications. Figure 6.1 begins by establishing an idealised classification of possible behaviours (CPU usage is sketched against time in each graph).

If we examine CPU usage on a single processor machine at fine enough time scales, any of these graphs will of course resemble pulses between 0 and 1, since only one task is running at once. At longer time scales we may see apparently noisy behaviour, before one of the other patterns becomes apparent.

Processes which exhibit noisy behaviour at time scales longer than their required response times will not be good candidates for adaptation or periodic scheduling in general.

Applications with fluctuating usage will benefit from adaptive scheduling in particular. Multimodal tasks are likely to experience temporary failure whilst the system adjusts to a mode change, unless the application is able to change its own contract at these

Figure 6.1: CPU usage classification



points. A common form of bimodal behaviour is shown in the "intermittent" graph; this corresponds to applications which have an idle state and a busy state, for example media players which can be paused.

High pulses will occur if a special operation is being carried out, rather than the ongoing processing of a data stream. This may be in response to a user interface action, for example. In such a case the application will generate a load of 100% until the operation is complete; this must not be confused with its normal behaviour.

It is quite possible that several of these characteristics will apply to one program, for example fluctuating behaviour with pulses. When an application is preempted because it has overrun its QOS contract, it may be difficult to distinguish between a pulse (which should be filtered out) and a gradual increase in usage to which the system should adapt, since we cannot tell how high the spike would have been without preemption.

6.2 CPU usage patterns

The next four figures show traces from real applications running on Linux-SRT. The graphs show how much CPU time the program consumed during 400 consecutive time periods. Each experiment is repeated three times, with periods of 10ms, 50ms and 250ms. The total experiment lengths are therefore 4, 20 and 100 seconds respectively. We are interested in the application's natural pattern and not crosstalk with other programs, so the target application was given a large (80%) QOS guarantee.

Figure 6.2 traces an mp3 audio player process. The bottom two graphs both show that the average CPU utilisation is 24%. The graph is quite noisy, with spot utilisation in the range [0, 60%] over 10ms time periods. Since the application possessed an 80%

guarantee, all its bursts must be shorter than 6ms. With 50ms periods, utilisation is in the range [12%, 40%] and at 250ms it reaches a more consistent [20%, 26%].

Figure 6.3: shows a MIDI synthesizer. This process has burst lengths greater than 10ms, as shown by the high peaks in the first graph. Its natural period appears to be 11 Hz. The lower graphs show an average CPU usage of around 40%. The usage fluctuates much more than the mp3 player does; even with 250ms time periods the range is a full [20%, 52%]. This is due to the changing complexity of the piece of music throughout the 100 second test and indicates that adaptation would be advisable.

The middle graph shows an apparent periodic behaviour at 1.5 Hz, which is a sampling artefact. The 50ms time period is approximately half the natural period of the process, so the usage rises and falls in alternate periods, covering a [0, 80%] range. The 1.5 Hz beat is caused by phase changes between the application and sampling processes.

This illustrates the importance of choosing the correct period for a process which is to be continuously adapted, since a period which is too small requires an unnecessarily large slice allocation to avoid missing apparent deadlines. This may be as undesirable as the poor scheduling caused by a period which is too large. Another consequence is that adaptation cannot be applied straightforwardly to Jubilee-based or Harmonic period systems (see Section 1.3.2).

Figure 6.4 examines a Quicktime movie player. The top graph reveals a natural period of 10 Hz, and the bottom two confirm an average CPU usage of 12%. The middle graph illustrates severe sampling artefacts once again, with a range of [0, 24%]. The 250ms graph narrows the range down to [9%, 17%].

Figure 6.5 traces an Afterstep window manager process. As this is a BE application with sporadic utilisation, the traffic shown here is essentially arbitrary. We include it to show the contrast with periodic processes only.

6.3 Initial adaptation

The simplest use of adaptation is to calibrate a program for the machine on which it is running, thus accomodating hardware differences. This is only necessary once, when the program is installed. To do so, the program is run and the system instructed to observe its CPU requirements over a period of time.

The alternatives to initial adaptation include a prior analysis of the program's performance, scaled by the various hardware parameters. In most cases this is much too difficult due to the complexity of the program. Another possibility is to pass the problem onto the user, who will then have to resort to trial and error, a time consuming process. Finally, in some cases it may be better to time a benchmark case instead of running the program normally, though this requires specially written support from the application.

The initial adaptation problem is two dimensional, since we must choose a period and a slice. During adaptation the overrun policy should be void, in order to establish the required slice. If deadlines are being missed (indicated by the process being policed

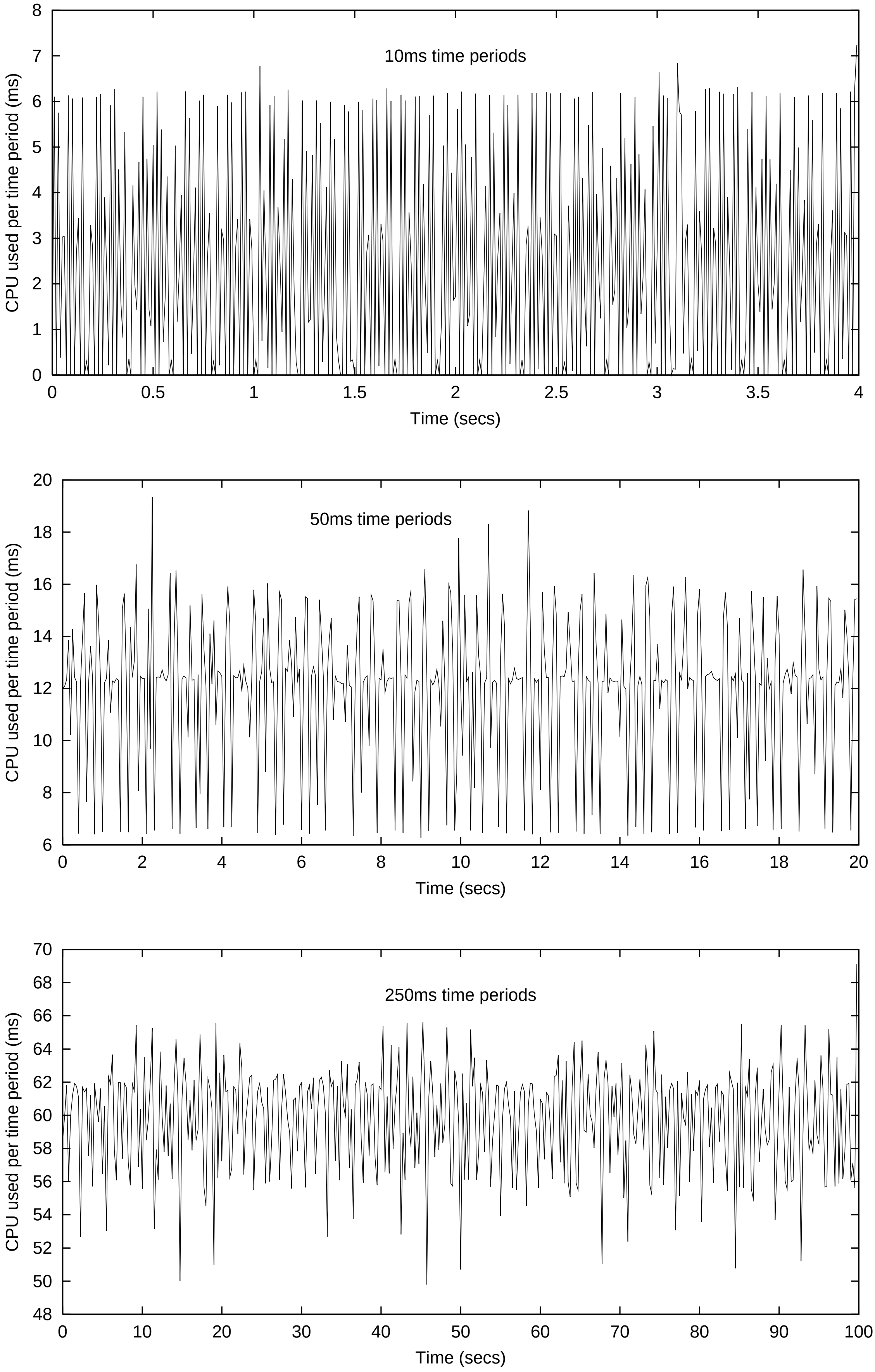


Figure 6.2: CPU traces for kmp3

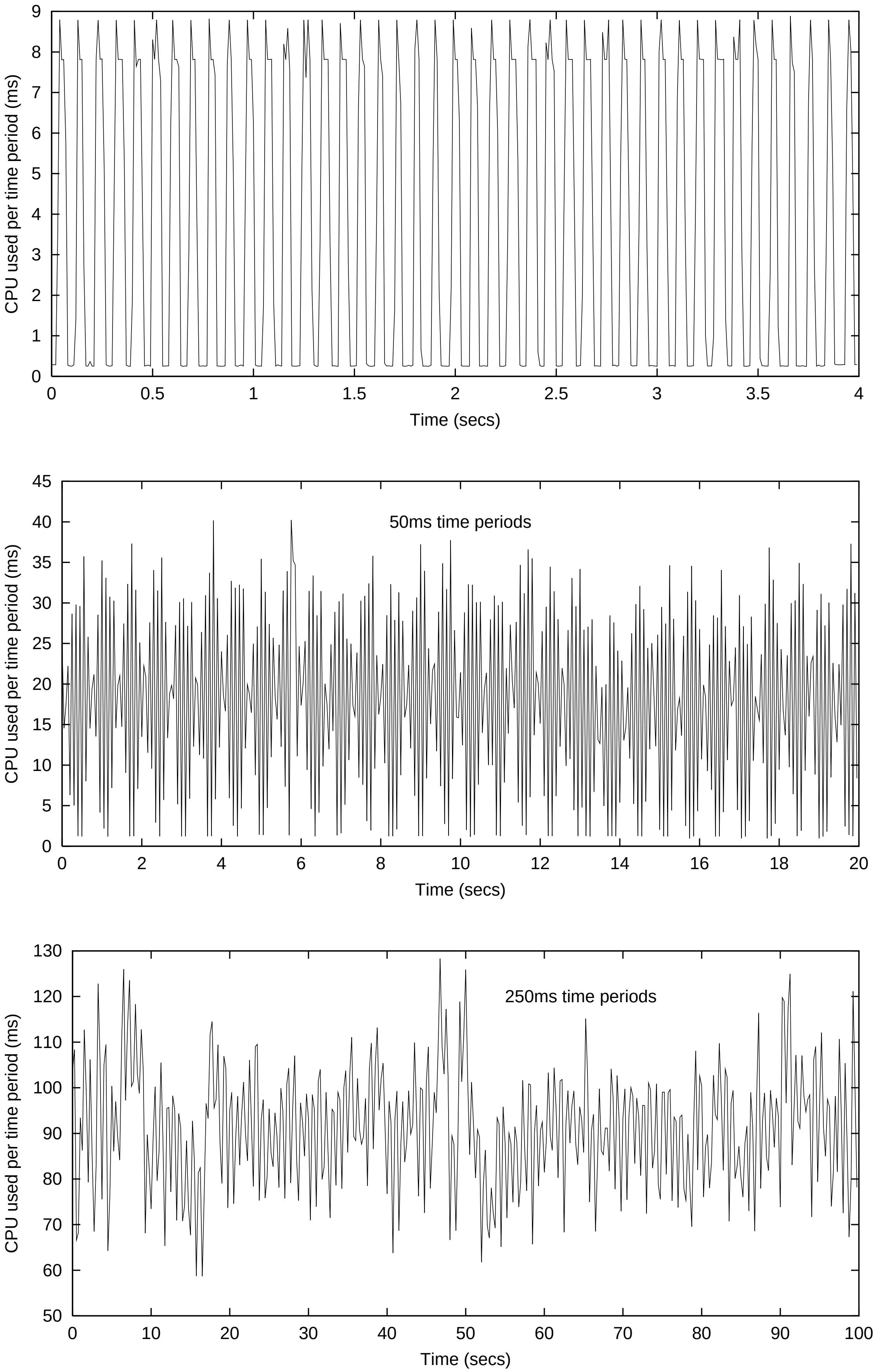


Figure 6.3: CPU traces for kmidi

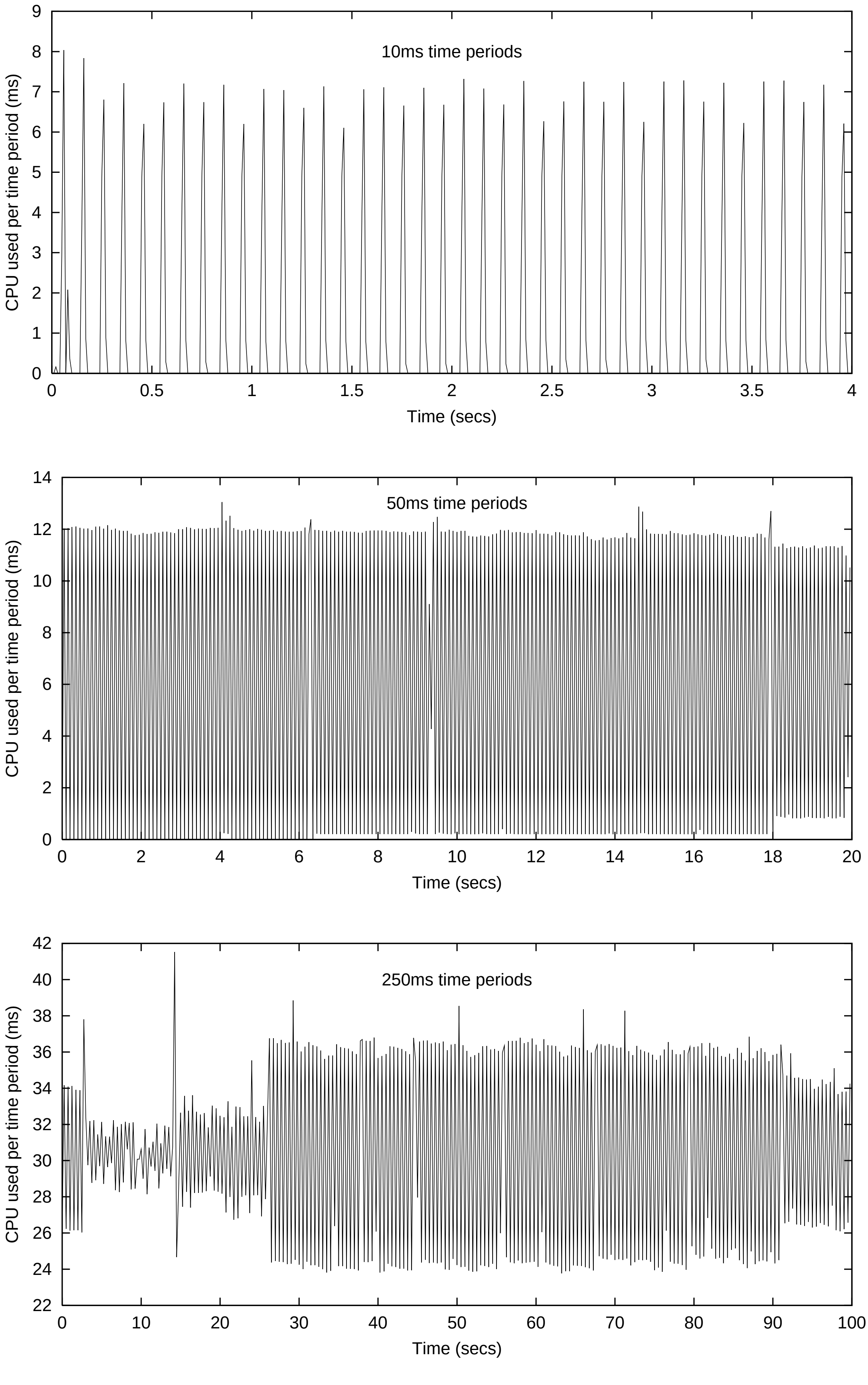


Figure 6.4: CPU traces for xanim

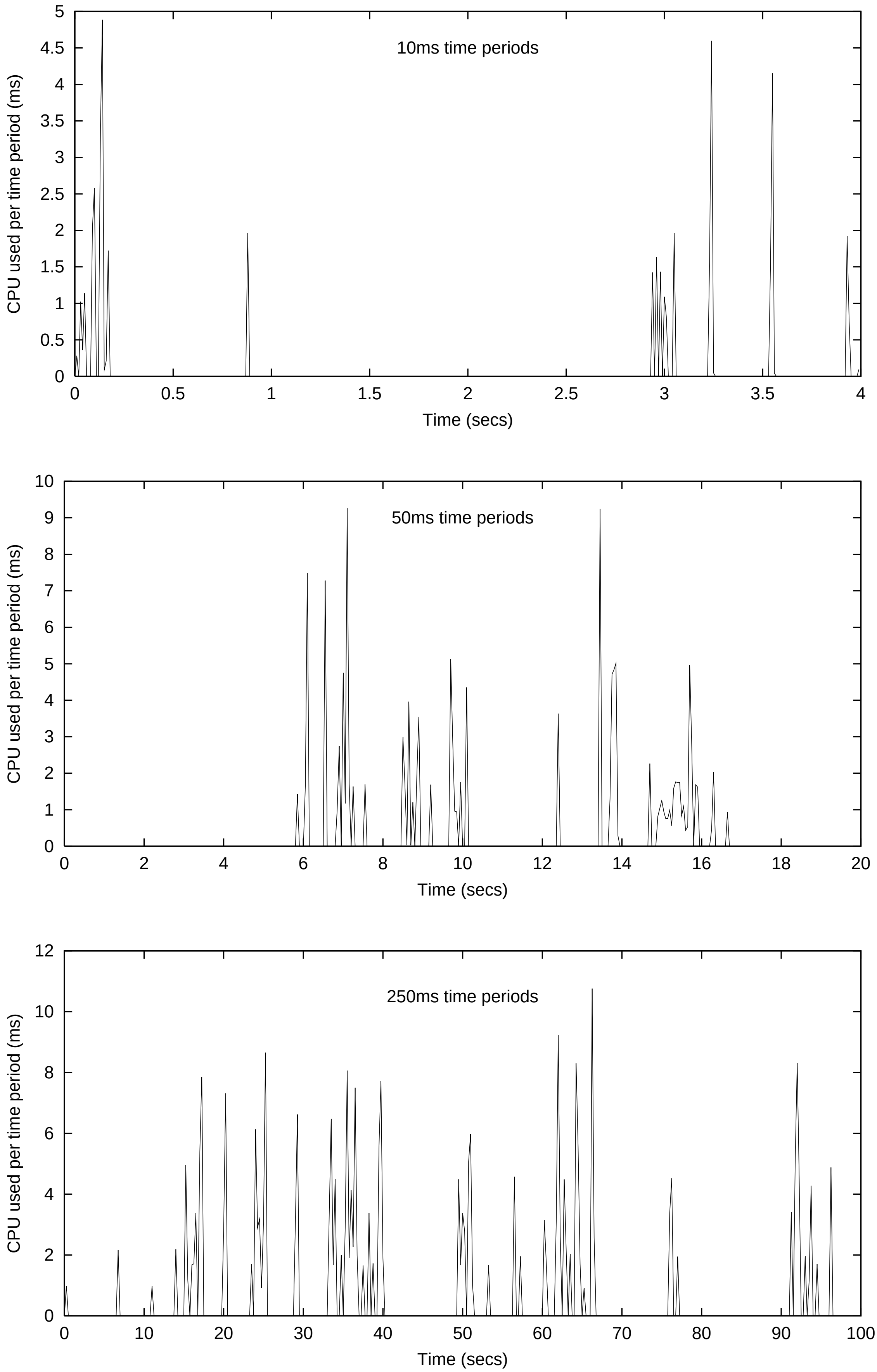


Figure 6.5: CPU traces for Afterstep (BE, aperiodic)

before blocking) the algorithm may try to increase the slice *or* decrease the period. If the usage varies considerably between periods it should increase the period. The adaptation process may potentially be quite efficient, although if it fails to converge we can fall back on sampling the parameter space exhaustively, since there are no particular time constraints for initial adaptation.

Once calibrated, the reserve is written out to the QOS configuration file. A probabilistic measure can be used.

6.4 Continuous adaptation

Continuous adaptation is necessary for any RT task whose workload varies considerably over time. Some examples are:

- Animations with a variable number of moving objects
- Graphical applications whose window can be resized
- Audio players which experience quiet periods

Of course, continuous adaptation should not be used for all tasks; we must provide hard guarantees for those which need them.

The technique works by automatically and gradually increasing the reservation for tasks which frequently run out of time, and decreasing it for those which block without using their full allocation. Again it will be advantageous to use a probabilistic model, aiming for the target miss rate specified by the user.

Periods in which the task uses very little CPU should be discounted if possible, so that we do not adapt down to zero during idle times. This is discussed further in the mode changes section below.

Periods are not adapted continuously; these are specified by the application or user at install-time, or during initial adaptation. Only slices are adapted at run time.

Adaptation should not be carried out during an application's startup phase, in which CPU usage is likely to be uncharacteristic. We can avoid this either by waiting a few seconds, or observing when the CPU usage settles down.

Continuously adapted contracts obviously do not provide firm guarantees whilst adapting, because it takes some time to adjust when demands increase. However, they do isolate the program from load variations in all the *other* tasks running on the system. Temporary failure whilst adapting can be made less likely by reserving slightly more CPU than the task requires, and using damping to prevent reservations being decreased too hastily.

Another disadvantage is that there is no guarantee that the system *can* augment the contract when it needs to. If a request to scale up fails, we prefer to retain the old parameters (so we don't lose the contract altogether).

When adapting continuously, the system should record a history of CPU usage to a log file on disk (possibly via `syslogd`). From time to time (at boot, or triggered by `cron`), these log files can be parsed and then deleted. The results are incorporated in a permanent statistics file which can be used to provide initial values next time the program runs.

6.5 QOS-Aware Applications

Linux-SRT has been designed to work with existing applications. However, new applications may be written specifically with QOS in mind. There are some benefits to doing so.

Applications which are QOS-aware can explicitly specify which threads are RT, and notify the OS of mode changes. They can translate from application QOS (in terms meaningful to the user, such as frames per second) to kernel QOS.

QOS-aware applications may also use performance feedback information provided by the OS. This has two distinct purposes: to allow them to control their own QOS, or to adapt their behaviour. Both these possibilities are optional.

6.5.1 (a) Application control of QOS Management

In this case the application's contract is continuously adapted as in Section 6.4, except that the program is responsible for making adjustments to its own reserve, instead of the QOS manager. It gets performance feedback from the kernel in order to make informed decisions about the reserve it needs.

There are two possible ways the system can react if a task runs out of time for a certain period, and has to be preempted:

1. Checkpoints: the task resumes obliviously next period. The QOS manager waits until it finally does block (or reaches its "checkpoint"). The total time taken is measured and reported to the application later through normal IPC mechanisms. It can use this as an estimate to enlarge its reservation.
2. Activations: the program is notified that it overran at the start of its next period, so that it can abort the part-finished computation and begin the next period's work immediately. Implementing this is more difficult under UNIX. The adaptive algorithm can then increase its reservation slightly for the new period.

The advantage of putting applications in the adaptation feedback loop is that it solves the mode-change problem through the use of application-specific knowledge.

The disadvantage is that it is difficult to integrate user preferences. Worse, applications don't have global knowledge, so they can't pick suitable priorities or revoke and negotiate based on other tasks and policies. Economic methods partially solve this by making applications aware of current resource "pricing", although this is only a single scalar. Our design therefore does *not* allow applications to perform their own QOS management.

6.5.2 (b) Support for Adaptive Application Behaviour

The second way in which an application can make use of performance feedback is by adapting its own behaviour. For example, it may switch to a cheaper algorithm if the required resource isn't available. Continuous Media applications can often run successfully at different rates, by adjusting frame rate or resolution. To take advantage of this, they should be configured with both a minimum and ideal QOS level, so that the QOS manager can reduce their reservation when it needs to shed load.

Note that this is the opposite process to the continuous adaptation described in Section 6.4. In that case the system adapted to the application's changing conditions, here the application adapts to the system instead.

The advantage of this strategy is that something intelligent happens when sufficient resource is not available. Of course, in some cases nothing intelligent *can* be done.

Scheduler activations have been used to enable adaptive application behaviour, and also to implement application-specific thread schedulers without the disadvantages of pure user-level thread packages [Leslie96, Anderson92].

Adaptive application behaviour is beneficial and harmless, because it cannot adversely affect other programs. It also requires little support from the kernel, since applications can detect whether they are keeping up with the workload using ordinary timers anyway.

6.6 Mode changes

Application mode changes cause sudden jumps in CPU usage and behaviour. Specially written QOS-aware programs may take advantage of knowledge about their own modes by requesting a different level of QOS just before a mode change. In our system we assume applications are *not* QOS-aware, though. Ideally the system should detect mode changes itself.

General-purpose mode change detection is not possible, because spikes and fluctuation can easily be mistaken for a new mode. The simplest form of mode-change support is idle-state detection, which is feasible (it is used to remove silent periods in audio compression). It is also useful in a RT system.

For example, often it is convenient to keep several different media player tasks (midi, CD, etc.) open at once. Typically only one such program will be playing at a time. A media player only needs its reserve whilst playing, and there typically won't be enough resource for them all to hold a reserve. Idle-state detection allows them to run simultaneously, each with their own QOS contract. This is similar to the way the audio device is shared (software mixers notwithstanding); it is allocated when needed and released when the stream is paused.

Another use of idle-state detection is to cancel the reserves of processes which have quit or crashed in some way but not yet exited.

Chapter 7

Policies

7.1 Application requirements

Before considering global QOS management policies, we focus on the needs of individual tasks. In this section we look at various typical applications, and determine suitable scheduling policies for them individually. The basic policies were described earlier in Sections 2.2, 2.4 and 3.3.

7.1.1 Real time tasks

CD burner	Hard QOS guarantee, initial adaptation
Arcade game	QOS guarantee
Midi player	Continuously adapted QOS
Full-screen movie	`SCHED_FIFO`, high static priority
Window manager	Optimistic QOS, small slice
Video phone	QOS guarantee, short period

Most of these real time tasks are best handled using QOS. The full-screen movie is an exception, because we are not interested in the parameters and are unlikely to run any other multimedia applications at the same time. This is most easily expressed with a static priority.

A video phone is different from a video on demand (VOD) movie because it has specific latency constraints. It does not matter when watching a movie if it is lagged a few seconds behind the broadcast source, so we can buffer well ahead to prevent underflow, and give the decompression task a relatively long period. Video phones need a round trip time of about 0.1 seconds for people to perceive them as real time, which implies a fairly small time period at each end, once communication delays are accounted for. Platforms with poor timing granularity, such as Linux, are therefore better suited to VOD than video phones.

7.1.2 Best effort tasks

Text editor, Compiler	SCHED_OTHER
Java applet	SCHED_OTHER with limit
Screensaver	SCHED_IDLE
Animated screensaver	QOS, small slice
Galaxy simulation	SCHED_IDLE, high weight
RC5 challenge	SCHED_IDLE, low weight

Best effort tasks are usually scheduled with the dynamic priority SCHED_OTHER policy. In the case of Java applets we are additionally concerned with *limiting* execution rates, rather than providing lower bounds. This is possible because Linux-SRT uses its policing and accounting mechanisms to support rate-based limits for *all* scheduling classes, including BE ones. Traditional support for this is crude indeed (UNIX can stop a process when it has consumed some total amount of CPU time since it started, but cannot provide an ongoing rate limit).

Large or multi-user systems require several levels of background priority. For example, we may be running time-consuming scientific calculations together with less serious activities, such as a distributed brute-force RC5 cracking process. This can be achieved using SCHED_IDLE combined with different weighting factors provided by *nice*.

An interesting curiosity is the animated screensaver. This is a notorious consumer of unnecessary cycles, but if it is made *nice* any other activity on the machine causes unpleasant, unpredictable jerkiness on the display. The two sensible options appear to be either not to use it, or in fact to make it real time! With a small slice and a hard limit (non-optimistic) we can ensure both smoothness and a restricted impact on system load.

7.2 QOS Management

A key concern for a desktop RT system is the calibration of QOS parameters. This must be done semi-automatically in order to be non-intrusive enough for everyday use, and to cater for novice users. However, applications usually do not know what their own QOS requirements are.

A QOS management component must take into account policies from several different sources:

- Default preferences (created at install time by the application)
- Static user preferences (importance and QOS)
- Dynamic performance feedback (adaptation)
- A stored history of previous requirements for this application

- The system load
- Application requests (typically due to mode changes, such as a video window being resized or paused)
- Dynamic overrides by the user

Modifications to applications are not necessary to enable QOS. The task fork code in Linux-SRT recognises binaries by name and looks up the parameters from a configuration file (a "policy editor" front end could be provided to edit this). QOS-aware applications may register themselves with the system at install time, and suggest appropriate defaults. Other applications won't be registered until the user adds them to the configuration file.

7.2.1 Desktop cues

Some systems have attempted to derive scheduling hints from the state of windows in the desktop environment. Microsoft Windows NT [Solomon98] makes decisions partly based on which window has the input focus and also on the presence of raised dialog boxes.

Weights can be applied to the "active" window, for example. We suspect this is not a good idea because input focus and scheduling are naturally distinct concepts. It's particularly distracting to tie them together if a no-click-to-focus mouse policy is used.

An ability to suspend or renice a task if a window is unmapped (completely invisible due to being on a different virtual desktop or minimized) can certainly be useful for graphical tasks, however.

Desktop hints are limited in usefulness by the lack of a one-to-one relationship between threads and windows.

7.2.2 Choosing QOS parameter values

We have seen that QOS scheduling is open, explicit and predictable; however it lacks intelligence. There is no simple way to choose the scheduling parameters for each task. By contrast, the UNIX scheduler is flexible and self-adjusting.

The need for automatic configuration of QOS parameters has not been well addressed to date. This restricts the application of QOS to commercial operating systems. We need to add a software layer called "QOS management" to bring back the ease of use of traditional schedulers, without losing the ability to make guarantees.

Centralisation

Much work has been done on decentralised QOS management. A common approach is to form a tree of scheduling domains [Ford96, Goyal96]. The available CPU time is split at each node and delegated to the child schedulers, in a similar way to proportional share group scheduling, but with the option of using a different scheduling policy at each node.

The decentralised approach is normally chosen to avoid a fixed set of built-in policies within the kernel scheduler. There are clear advantages to allowing an application to schedule its own threads, but this can be done with hooks (scheduler activations) and doesn't require a full tree hierarchy. Furthermore, any real time characteristics which a child scheduler would like to guarantee must also be provided by its parent, and hence the root scheduler in the first place.

We have adopted a centralised QOS management strategy, because the most effective scheduling requires *global* knowledge about the running task set.

One example of the difficulties caused by local scheduling is the inability to provide *nice* or `SCHED_IDLE` behaviour. A task may defer to its siblings under a single node of the tree, but if they are idle it will always consume the full resources allocated to its parent scheduler. It has no way of indicating to the rest of the tree that it wishes to donate CPU time to any more important tasks which exist elsewhere.

A more flexible method of decentralised QOS management can be created using an adaptive economic model [Stratford99]. In this case resources such as the CPU have a "cost" which depends on the demand for them. The price of each currency is a form of feedback which allows load information to cross the boundaries imposed by strict hierarchical reservations.

7.3 Admission control

Admission control policies specify for each application whether to reject it at startup if guaranteed time isn't available, or to run it without QOS. Similar choices exist for a mode change. Other possible actions are to suspend the process until resources become available, or to steal them from other processes (see below).

Admission control should only be performed at times where possible failure is acceptable. Practical experience also suggests that the system should always provide feedback to the user when admission control fails to establish a contract (see Chapter 6). In some cases an unseen process may still be holding a reserve, blocking a new contract the user expects to be granted easily.

Extra flexibility can be introduced by allowing tasks to specify both an *ideal* and a *minimum* level QOS. [1] For example, a simulator may have an ideal of 25 FPS and a minimum of 10 FPS. Similar techniques have been used by the QOS manager for

[1] If adaptation is used, these values may be derived from a range of acceptable overrun probabilities specified by the user (say between 0.2% and 2%).

Processor Capacity Reserves [Mercer96a]. Tasks with fixed workloads and no choice of algorithm simply specify the same value for ideal and minimum QOS.

We then have the option of scaling down existing contracts in order to admit new tasks, while still observing minimum levels. Adjustment polices should specify which tasks to degrade first, and whether to start new arrivals at minimum QOS in preference to reducing any existing tasks below their ideal levels. A further possibility is to mark some tasks as revocable-by-stopping (these tasks are paused until resource becomes available again).

7.3.1 Order dependence

An important question is whether the final QOS allocations for a given task set should depend on the order in which each task was started.

At first glance a lack of dependence on starting order appears to be a useful property, since it frees the user from worrying about the order in which tasks are launched by startup scripts and so on. It emphasises the *importance* of tasks over the length of time they have been running.

We note that in order to arrive at the same result regardless of starting order, it may be necessary to revoke or reduce earlier contracts to make room for more important, later processes.

The alternative is to maintain a FCFS scheme, which does not require revocation but is order dependent. We believe that order dependence (at least to some degree) is the correct solution, and a key principle of Linux-SRT is that contracts are *never revoked*. This is required in order to ensure predictability for tasks with guarantees, a central concern.

There are disadvantages which must be observed, particularly on a multi-user system. These are less serious if contracts have been established with explicit fallback levels, timeouts or adaptation criteria—since we *may* then revoke them, at least partially. Such techniques provide flexibility in the system, and can be appropriate for non-critical RT tasks.

Order dependence can be extremely useful. On a single user system in particular, ultimately users will perform some kind of scheduling discipline themselves; by deciding how many processes to run at once, whether to quit some of them or ultimately to log out if performance is unacceptable. This argues in favour of order-dependence. It's more intuitive to start with the essential processes and then add load until newcomers can't be accommodated, than to have the system shift allocations and alter existing guarantees.

The user may even be frustrated if they try starting tasks in a different order deliberately to achieve an unusual scheduling effect, but the system persists in adjusting to the same endpoint. An order dependent system is more comprehensible and psychologically acceptable.

7.3.2 Alternatives to QOS

Sometimes it is more natural to express *importance* rather than order-dependent CPU shares. Two ways of doing so are static priorities and weighting factors.

It is important that any static priorities be specified by the QOS manager (see Section 8.2) and not by the application, since global knowledge is essential to form a static priority schedule. There is no safe way to do this on a multi-user system, so special authorisation is required.

Some systems have assigned tasks timing deadlines *and* a measure of importance; these schedulers attempt to balance both factors [Mosberger98, Nieh97].

Proportional share scheduling uses a weighting factor for each task. We could envisage a new policy, `SCHED_PROP` perhaps, to provide this capability. There are only a few cases in which this is clearly preferable to the policies we have discussed in detail. One advantage of weighted scheduling is that it is easy to translate a user's imprecise notion that a given task is 'important'; ad-hoc requests to assign better service to a running task can be interpreted by increasing its weighting.

Recall that the *nice* parameter in Linux acts similarly to a weighting factor for CPU-bound tasks. This could be better defined, capable of larger and smaller weights, with meaningful timespans for the decay of priorities (not depending on ticks and magic numbers). Alternatively, weights can be built on top of the RM scheduler. Either way it is possible to avoid the frequent reschedules which occur when lottery scheduling is used, as well as the need for compensation tickets [Waldspurger94].

7.4 Access Control

Access control for QOS reserves is partly an authorisation procedure, and partly a resource allocation problem.

Resources such as BE CPU and network bandwidth are subject to time division allocation, whereas disk space and physical RAM pages are based on space division. Guaranteed CPU time falls into the space division category because reserves have relatively long lifespans. This makes flexible access control difficult. Some of the solutions we consider include probabilistic access, resource exchanges, timeouts, and credit-based systems.

7.4.1 Capabilities

Ordinary users must have some authority to establish QOS contracts in a general purpose RT system. Capabilities provide a convenient way of separating such powers from root user status.

Capabilities designate the rights held by a running process. They are also associated with executable programs, which provides a fine-grained alternative to SUID. They may be assigned based on identity (User ID) or roles (Group ID). User-based protection works well because it corresponds with obvious real world entities (and is already well supported by UNIX). More complicated nested schemes do not have this advantage.

Specific resources are associated with each capability. This separates the concept of identity from the currently available privileges and is motivated by the principle of least required privilege.

Linux Capabilities are based on POSIX.1e, with some additions. They are represented by a 128-bit set. See Table 7.1 for some examples. Capabilities with the prefix `CAP_SCHED` are unique to Linux-SRT; the only standard capability relating to scheduling is `CAP_SYS_NICE`. The Windows NT [Solomon98] "increase scheduling priority" privilege is similar.

In Linux-SRT the `CAP_SCHED_STATIC` capability is required to use static priorities, and this is not generally permitted for normal users. Even with this capability it is not possible to consume all of the available processor time without `CAP_SCHED_LIMIT`, which overrides the reservation limits.

We use pure capabilities at the kernel API (system call) level, to keep it policy free. More complicated access control based upon group membership can be implemented with privileged user level tools.

CAPABILITY	EFFECT
`CAP_SYS_NICE`	May use negative nice values
`CAP_SYS_TIME`	Allow manipulation of the system clock
`CAP_SYS_QUOTA`	May change disk quotas
`CAP_SCHED_ADMIN`	Allows use of the calls `billing()` and `join_reserve()`
`CAP_SCHED_QOS`	May request QOS
`CAP_SCHED_STATIC`	Allows use of static priorities
`CAP_SCHED_LIMIT`	May exceed scheduling limits

Table 7.1: Example Linux and Linux-SRT capabilities

7.4.2 Multiple users

The access control problem is more difficult for a multiuser system. There is a tradeoff between fairness and making effective use of the CPU. If the total resource available is R and there are n users logged on, a "fair" solution would be to supply each user with a guaranteed *share* of R/n. However, many users are normally idle, so these shares are unnecessarily small.

The alternative is to allow variable size shares for users, allocated on a FCFS basis. It is reasonable to tolerate some unfairness. The "service neutrality" argument [Liedtke99] points out that denial of service conditions equivalent to large RT reservations are possible using many BE processes anyway.

If we vary the share assigned to each user, the system becomes not only unfair but unpredictable as well, since users are then affected by each others actions. One way to make this more acceptable is to keep the user informed of resource availability.

Negotiating resource limits for a user at login time is not a good solution. The user may log out and in again, be logged in several times simultaneously, or stay logged in for as long as they like, so login times are arbitrary for this purpose.

Multiuser revocation

As discussed in Section 7.3.1, revocation of individual task contracts is not allowed. The reduction or revocation of a user's reserve when new users log in is therefore problematic, because we may need to revoke task contracts as a consequence.

A popular solution is to give all contracts a timeout value. This ensures that users cannot hold resources forever. In a credit-based system, long timeouts may cost more (consuming more of the user's apparent share), which encourages frequent renegotiation. A fundamental problem is that applications prefer long timeouts, but short timeouts are required to share the CPU more effectively.

A better approach may be to use automatic idle time detection. We can "suspend" the contracts of those tasks which are not making significant use of their reserves, and no others. Suspending contracts may not be permitted for all tasks. In an economic system, the certainty of guarantees could be traded against their slice.

Existing shares can be scaled down to accomodate new users, provided that no task contracts are revoked in the process. In the meantime, the new arrival may have less than their fair share of time. We can bring them into line when other tasks quit. Eventually all existing users could have the minimum permitted share. Further new users must then be admitted without guaranteed time.

7.4.3 Resource exchanges

Several methods of fairly allocating non-schedulable (space-division) resources are based on resource exchange in an economic system.

A system created at IBM [Liedtke99] allows unlimited memory pinning by exchanging unpinned pages for locked ones.

The same idea could be applied to allow a user to obtain a greater `SCHED_QOS` allocation at the expense of their `SCHED_OTHER` service. This could be achieved by introducing a reduced weighting factor for such BE tasks.

Extensions to multi-resource lottery scheduling [Sullivan99] have been developed to allow resource exchanges and keep allocation fair. For example, CPU time can be traded for additional disk bandwidth or RAM pages. Brokers are used to coordinate resource exchanges.

Chapter 8

User Interface

8.1 Overview

The main aim of the Linux-SRT user interface is to achieve high levels of automation whilst satisfying the user's specific real time requirements. Adjustments are typically made once at configuration time. Simplicity is paramount.

The user interface also has interactive components, which provide manual control and user feedback. Although it has proven useful to override QOS management in rare cases, interactive controls are also essential for experimentation prior to making persistent configuration changes.

Feedback to the user is useful for troubleshooting purposes, but is not needed continuously. A graphical QOS display helps keep the user's mental model of the system close to the actual one. For example, the user can clearly see why the QOS manager is having trouble admitting a new program.

8.2 AutoQOS

When a program is started with the `exec()` syscall, a configuration database is checked to select suitable scheduling parameters, which are applied to the task immediately. This was originally performed by the kernel, but is now implemented in user-space by a daemon listening to `sys-exec` kernel events. Tasks receiving automatic QOS allocations are subject to admission control in the usual way. The mechanism is referred to as *AutoQOS*.

System default settings are read from the file `/etc/qosrc`, and user-specific ones from `~/.qosrc`. Program names and argument lists can be specified using UNIX shell style wildcards. Specific cases can be listed later on to override more general matches. A flag provides the option disable the normal behaviour of applying QOS parameters automatically when the tasks starts; in this case they are stored for use later in case the user chooses to activate RT scheduling for the task whilst it is runnning.

The AutoQOS daemon, `autoqosd`, is started when the system boots and runs continuously. When it starts up it reads the `/etc/qosrc` configuration file. Any named reserve definitions present are passed onto the kernel at this point.

Subsequently, `autoqosd` monitors application startup using the `sys-exec` kernel event. It applies the scheduling parameters you specify to programs automatically once they start running (however they are launched). Automatic reserves (see Section 3.7.2) are used to generate names for the reserves created by AutoQOS in this way.

If you modify `/etc/qosrc`, you can send `autoqosd` a `SIGHUP` to force it to read the configuration file again.

8.2.1 QOS configuration file

The global QOS configuration file is `/etc/qosrc`. Lines in the qosrc file must conform to one of the following formats (anything else is treated as a comment):

```
1. "pattern" nice pri per P cpu Q A I
2. "pattern" <reserve>            A I
3. <reserve> nice pri per P cpu Q
```

Lines of types 1 and 2 associate a program with a reserve (automatic or named respectively). They are used by the control program `setp` when invoked with the "lookup reserve" option, and by `autoqosd` when tasks perform an `exec` call.

Lines of type 3 define a named reserve for later use (no QOS is allocated until a task joins that reserve). They are read by `autoqosd` when it starts up, in order to pre-register the named reserves with the kernel. Applications may join them later and the user can select them conveniently when making interactive scheduling adjustments.

`A`	=	Auto
`I`	=	Inherit
`P`	=	Policy
`Q`	=	Overrun policy
`nice`	=	nice
`pri`	=	static priority
`per`	=	period (ms)
`cpu`	=	slice (%)

Table 8.1: `/etc/qosrc` configuration file key

The meanings of the fields on each line are given in Table 8.1. The policy codes for the `P` and `Q` fields are as follows: F = FIFO, R = RR, O = OTHER, Q = QOS, I = IDLE, P = PAUSE.

"Auto" applies parameters at task startup (otherwise they are only applied at the user's request, via setp or the window manager integration). "Inherit" specifies if children should inherit these parameters.

Example

```
# /etc/qosrc
#
# cmdline       nice  pri  per   P  cpu   Q   A I
# ----------------------------------------------

"xmms*"          0    0   50   Q   50   P   1 1
"kmp3*"          0    0   50   Q   25   O   1 1
"xanim"          0   30   50   F   50   O   1 1
"*realplay*"   -20    0   50   O   90   I   1 1
"xlock"          0    0  100   O   10   I   1 1
"krecord"       <fifo>                      1 1
"kmedia"        <fifo>                      1 1
"*"             <default>                   0 1

<default>        0    0   50   Q   50   O
<ui>             0    0   50   Q    5   O
<idle>           0    0    0   I    0   P
<pause>          0    0    0   P    0   P
<fifo>           0   30    0   F    0   P
<safe>           0   30   50   F   90   O
<ceiling>        0    0   50   O   75   P
```

8.3 Command line interface

The CLI includes a program to set scheduling parameters, another to display them, and a load generator for benchmarking.

8.3.1 Control program - `setp`

The control program can be used to launch new processes (using `exec`) with particular scheduling parameters applied, or “attach” to processes which are already running. It is primarily useful within scripts. Any scheduling class can be requested, so tasks can be launched with a given QOS or at POSIX priorities or nice factors.

```
Synopsis: setp [options] <function>

Functions:
       e <command> <arg1> <arg2> ... = exec command
       a <pid> = attach to arbitrary process

Options:
       -t         = Toggle between normal and specified scheduling
       -f <value> = POSIX realtime priority (FIFO policy)
```

```
-n <value> = nice [-20, 19]; same as Linux setpriority
-r <name>  = join the reserve specified
-l         = lookup and join the reserve configured by /etc/qosrc
-z         = return to normality
```

8.3.2 Status information - `viewp`

This utility can be instructed to behave in a similar way to either `ps` or `top`, depending on whether continuous monitoring is required. In both cases it displays scheduling class, parameters and performance figures. An option is provided to list only the "interesting" tasks, i.e. those with non-default parameters or significant CPU usage.

8.3.3 Load generator - `loadgen`

The load generator runs in a busy loop to simulate various kinds of CPU usage in a reproducible way. Options supported are as follows:

- Continuous or bursty usage (configurable period and slice)
- Progress indicators (CPU and wallclock times)
- Forking multiple tasks (used to test timeslicing overhead)

8.4 Monitor program

The GUI monitor program is based on *qps*, a graphical version of `top`, using the Qt toolkit [Dalheimer99]. Figure 8.1 shows a typical display.

The snapshot in Figure 8.2 highlights the more important scheduling fields (policy, POSIX priority, period, slice, percentage CPU use and command name). In this case three raytracing processes have been given different scheduling classes to illustrate the effects on CPU allocation.

Monitoring is performed by reading the `/proc` filesystem. The refresh rate, displayed fields and sort key are configurable. A bar graph in the top right shows the proportion of time allocated to QOS reserves.

The monitor program also permits interactive adjustment of QOS parameters. All the scheduling attributes for a task can be adjusted in a dialog box (Figure 8.3). The user can also select groups of tasks and apply changes to all of them at once.

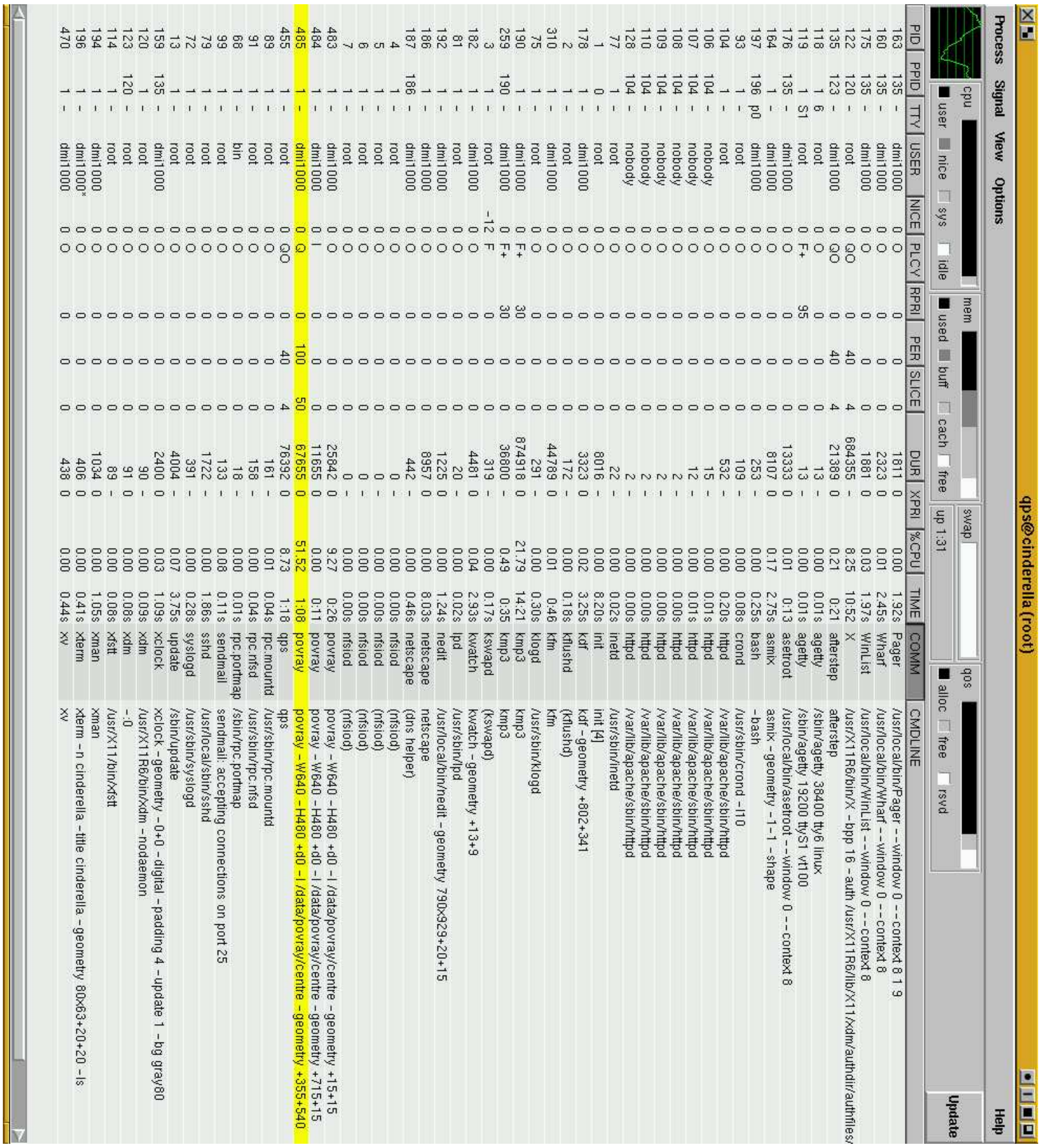
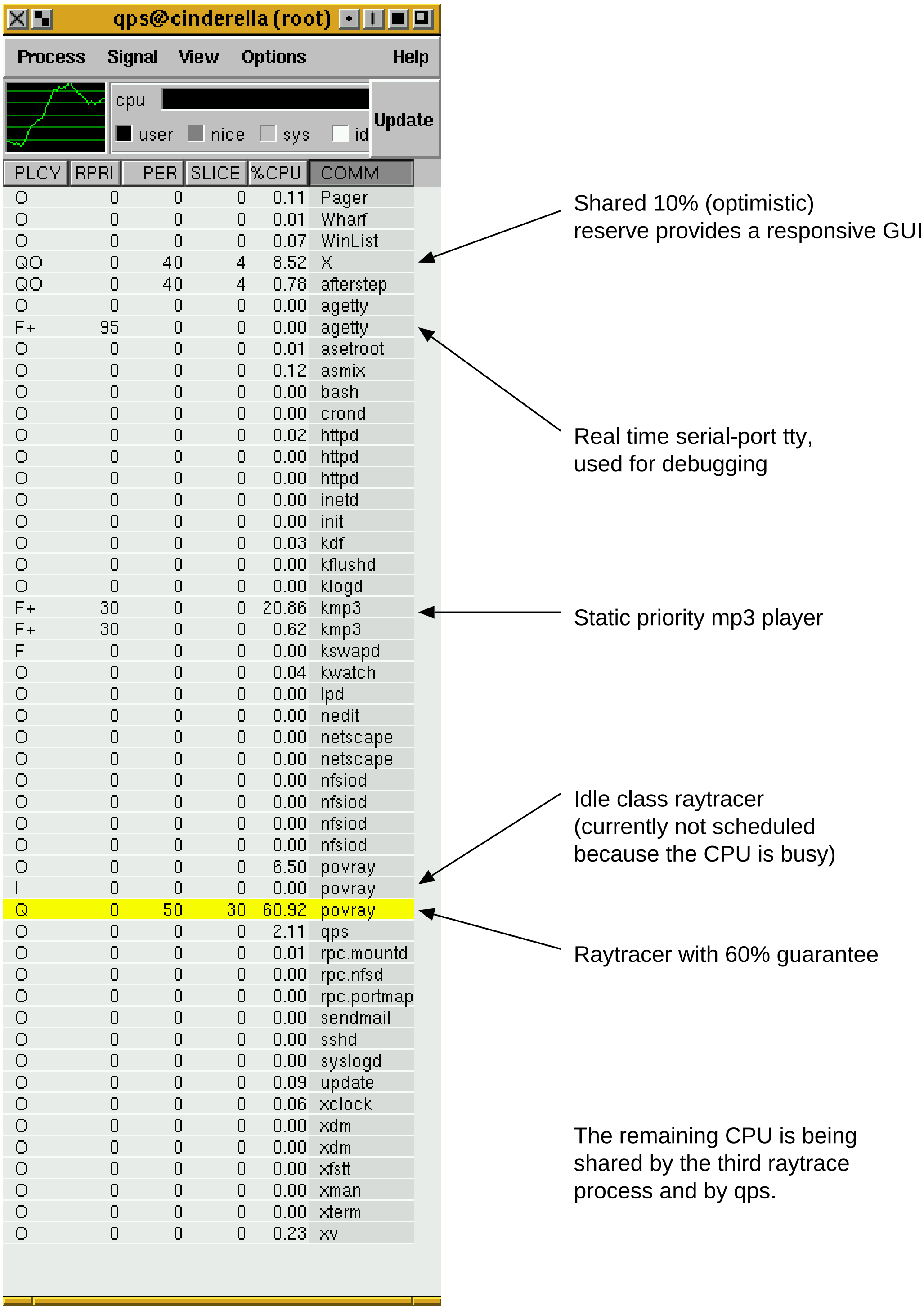


Figure 8.1: Monitor program



PLCY	RPRI	PER	SLICE	%CPU	COMM
O	0	0	0	0.11	Pager
O	0	0	0	0.01	Wharf
O	0	0	0	0.07	WinList
QO	0	40	4	8.52	X
QO	0	40	4	0.78	afterstep
O	0	0	0	0.00	agetty
F+	95	0	0	0.00	agetty
O	0	0	0	0.01	asetroot
O	0	0	0	0.12	asmix
O	0	0	0	0.00	bash
O	0	0	0	0.00	crond
O	0	0	0	0.02	httpd
O	0	0	0	0.00	httpd
O	0	0	0	0.00	httpd
O	0	0	0	0.00	inetd
O	0	0	0	0.00	init
O	0	0	0	0.03	kdf
O	0	0	0	0.00	kflushd
O	0	0	0	0.00	klogd
F+	30	0	0	20.86	kmp3
F+	30	0	0	0.62	kmp3
F	0	0	0	0.00	kswapd
O	0	0	0	0.04	kwatch
O	0	0	0	0.00	lpd
O	0	0	0	0.00	nedit
O	0	0	0	0.00	netscape
O	0	0	0	0.00	netscape
O	0	0	0	0.00	nfsiod
O	0	0	0	0.00	nfsiod
O	0	0	0	0.00	nfsiod
O	0	0	0	0.00	nfsiod
O	0	0	0	6.50	povray
I	0	0	0	0.00	povray
Q	0	50	30	60.92	povray
O	0	0	0	2.11	qps
O	0	0	0	0.01	rpc.mountd
O	0	0	0	0.00	rpc.nfsd
O	0	0	0	0.00	rpc.portmap
O	0	0	0	0.00	sendmail
O	0	0	0	0.00	sshd
O	0	0	0	0.00	syslogd
O	0	0	0	0.09	update
O	0	0	0	0.06	xclock
O	0	0	0	0.00	xdm
O	0	0	0	0.00	xdm
O	0	0	0	0.00	xfstt
O	0	0	0	0.00	xman
O	0	0	0	0.00	xterm
O	0	0	0	0.23	xv

Figure 8.2: Monitoring real time parameters

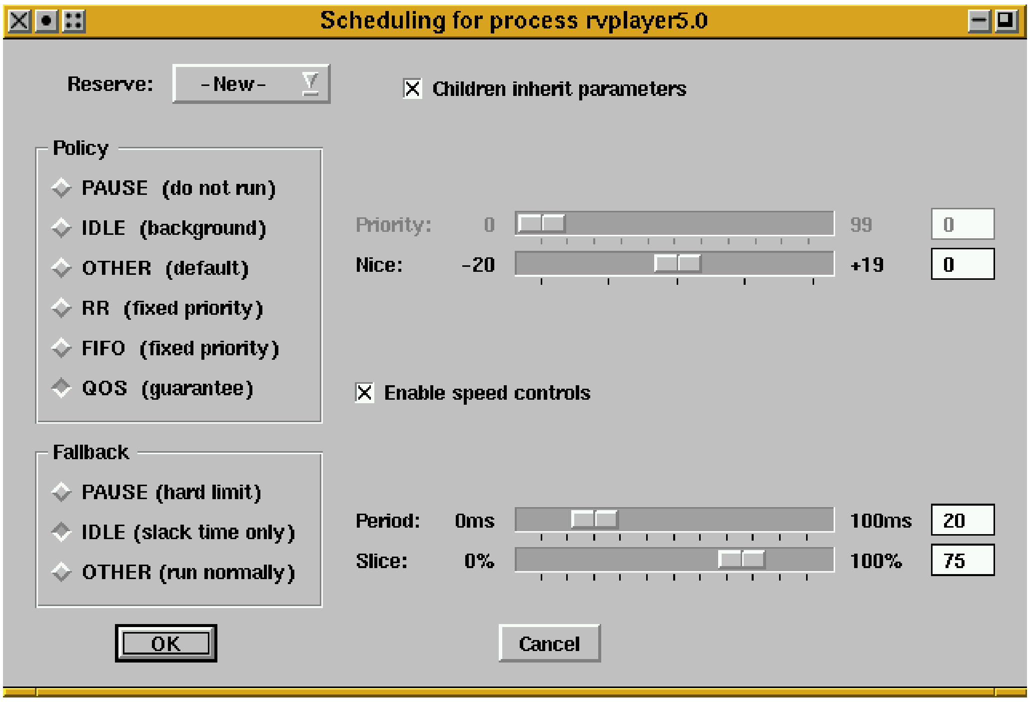


Figure 8.3: Scheduling parameters dialog

8.5 Window management

Standalone monitor programs are useful, but too intrusive for everyday use on a desktop system. For this reason Linux-SRT supports window manager integration. Initially we used fvwm 1.2 to test this; the current version uses Afterstep 1.8. The functions are relatively easy to port to different window managers however. Allowing the window manager to provide input to the QOS management system enables a direct manipulation interface which is much more natural for occasional on-the-spot changes.

8.5.1 Identifying clients

The window manager uses titlebar buttons to display information about the quality of service associated with each window, and to permit dynamic adjustments.

To do this, the WM maintains a mapping between windows and their respective client PIDs. In the case of Afterstep, the PID is added to the `ASWindow` structure, and a hash table permits efficient lookups by PID.

The mapping is evaluated lazily with a new Xlib function, `XGetClientIdentity()`, and its supporting X request. The server returns the PID it established at client connection time using authenticated sockets.

Under the present system the window manager can control scheduling for processes running on the same machine only. Using TIDs instead of PIDs would correct this. Remote X sessions already work provided the enhanced X server is used at the terminal.

This requirement could also be removed by storing TIDs in window manager hints (which are supported by all X servers), thus making the custom X request obsolete.

The new data paths connected to the window manager are summarised in Figure 8.4.

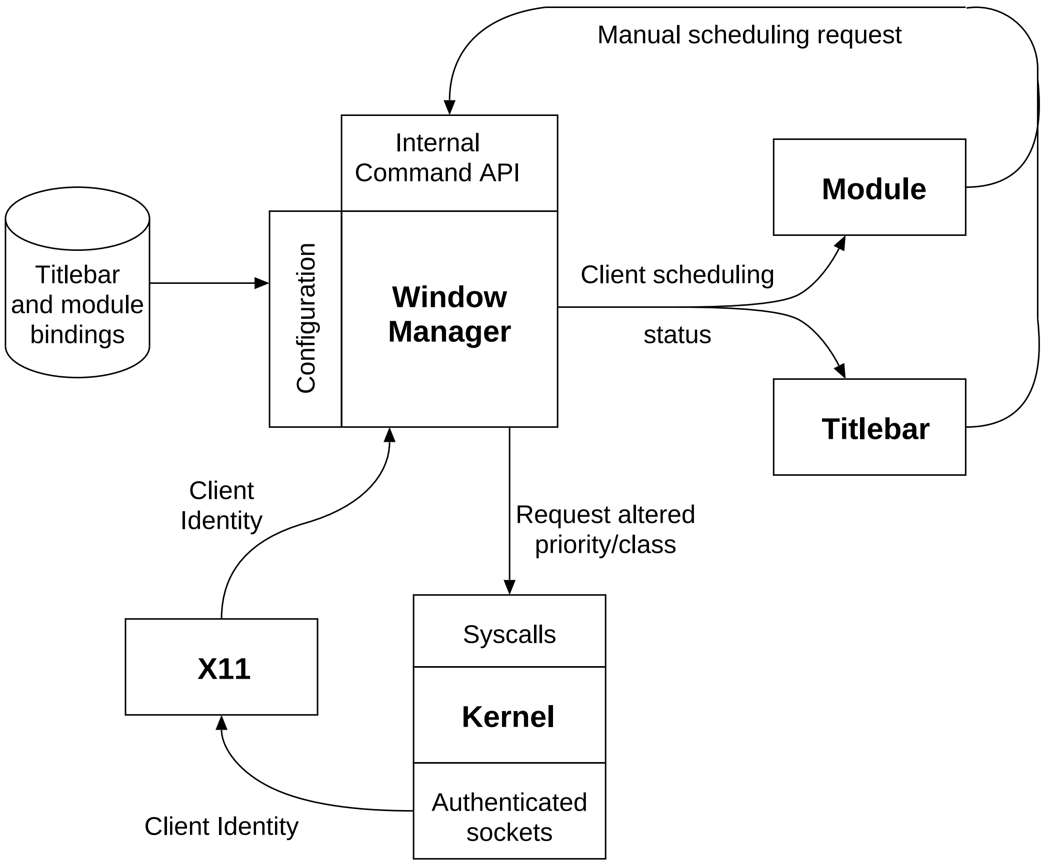


Figure 8.4: Window manager interaction

8.5.2 Titlebar buttons

The button images used are shown in Figure 8.6(a). Each button has a normal appearance and a sunken alternative one, which is used when it is clicked. Figure 8.6(b) shows the standard window decorations, using the mixer task as an example. Overlaid copies of the window are included in each of the five possible service states recognised by the window manager.

8.5.3 Modules

An alternative to titlebar buttons is to use window manager modules. These communicate with the window manager using pipes and can trigger the same set of internal commands.

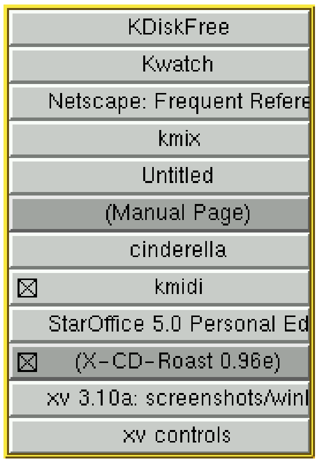
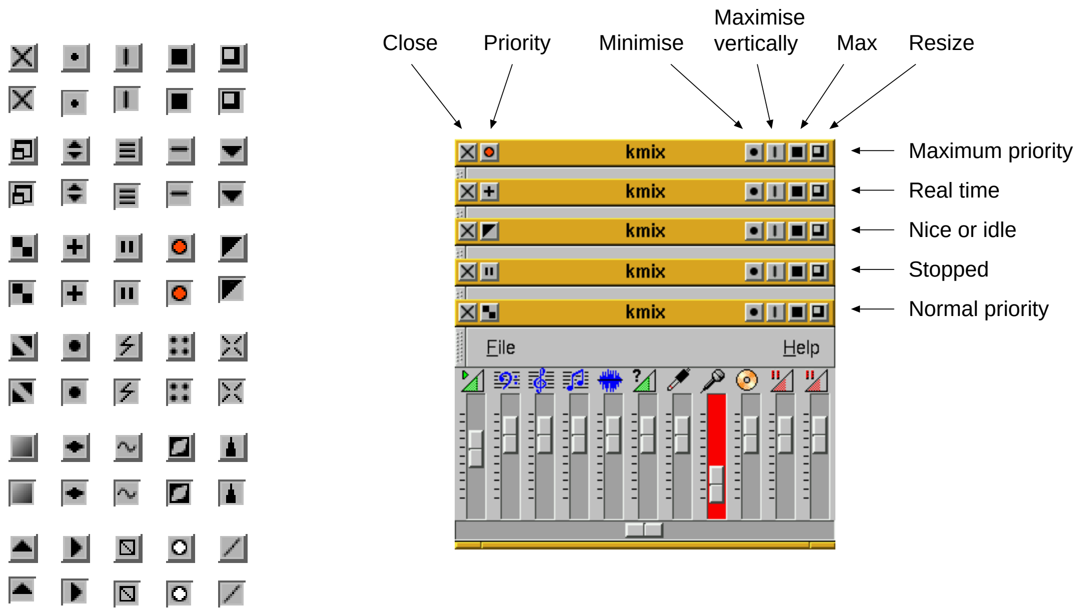


Figure 8.5: Window list module



(a) Image set

(b) Titlebars

Figure 8.6: Titlebar buttons

Figure 8.5 gives an example of this. It shows a version of FvwmWinList which has been modified to provide more information. Minimised windows are shown as darkened buttons, and real time scheduling status is flagged with a small icon to the left of the name.

8.5.4 Control

It would be possible to provide detailed control over the various scheduling parameters, although these are not something the desktop user needs to know about. However, users have asked for a single control which can be used to make the QOS "better" for a particular task as required [Mosberger98].

The bindings used at present are listed in Table 8.2. These can be configured in the window manager's `feel` file. Rather than complicating the window manager with additional built-in functions to control scheduling parameters, we make use of its ability to launch arbitrary external programs. We have enhanced this so that Afterstep will replace any occurences of the string `<pid>` with the PID of the process which owns the window in question. The modified command is then executed. Of course this would not be possible without `XGetClientIdentity()` and authenticated sockets.

For example, the left mouse button binding is achieved with the following line from the `feel` file (notice that we use the `setp` control program and pass the PID as a parameter):

```
Mouse   1   3   A   Exec "" setp -t -l a <pid>
```

Mouse button	Function
LMB	Toggle real time status
Shift-LMB	Toggle FIFO scheduling
MMB	Toggle idle scheduling class
Shift-MMB	Toggle nice scheduling class
RMB	Toggle pause scheduling class
Shift-RMB	Pop-up scheduling dialog

Table 8.2: Mouse button bindings

If the task is listed in the QOS database, the "real time" command enables or disables its contract. Tasks which are not listed receive a standard reserve instead (25% with a `SCHED_OTHER` fallback policy, by default).

When the user clicks on a titlebar control, the relevant internal command is executed. A check is made to see if the window PID is known (not true if the task is running remotely). Afterstep then requests new scheduling parameters for it.

The authority of the user running the window manager is used to determine if the scheduling adjustment is honoured (this will generally be the same as the user running the client).

Afterstep does not immediately change the button image, other than presenting a lowered 3D look to confirm the actual mouse click. If the request is successful feedback will arrive as for any scheduling class change (see below), thus closing the loop and providing confirmation to the user.

8.5.5 Feedback

Many kinds of feedback could be provided by window decorations, but simplicity is also very important here. For example, an indicator to show whether a reserve is currently under-utilised or overrunning would be too detailed. We use a single button to display broad scheduling classes and status, as shown in Figure 8.6(b).

The following features allow titlebars to properly reflect window service-class status:

- An extended `TitleButton` option in Afterstep's `look` file allows the user to specify multiple images for each button at configuration time. This is a widely useful feature; for example Microsoft Windows substitutes images for maximise buttons in accordance with the current state. Linux-SRT allows up to 10 buttons, with 10 possible states each.
- A new option in Afterstep's `feel` file called `ButtonFeedback` specifies which buttons should update their images in accordance with particular kinds of state change. We support maximisation, window shade, and stickyness in addition to scheduling class.
- When Afterstep calls `select()` to wait for events, it includes a file descriptor attached to the `sched-param` kernel event source provided by `/dev/events`.

- On notification of a scheduling parameter change, it queries that particular task's reserve to discover the new values.
- The new button image is substituted and the titlebar redrawn.

Chapter 9

Conclusion

9.1 Summary

We have shown that SRT scheduling can be implemented effectively on a conventionally structured, general purpose desktop platform.

Precise scheduling depends on the timing granularity of the kernel, yet even a monolithic kernel proved capable of guaranteed performance with typical multimedia applications, once support for QOS was added. Existing shared servers can be adjusted to respect client priorities; we have demonstrated this technique with the X server.

Linux-SRT has been in everyday use for the last 18 months, as a normal working environment as well as a real time and development system. The system can be downloaded from `http://www.uk.research.att.com/~dmi/linux-srt/`. It has been successfully downloaded and installed on standard Linux boxes without requiring any technical support.

By keeping scheduling constraints separate from functionality, we have shown that it is not necessary to sacrifice binary compatibility with existing applications. We have also investigated automated scheduling policies and made the user interface simple for novice users to understand.

RT applications are becoming increasingly commonplace, but general purpose operating systems do not provide any kind of appropriate support for them. Integrated real time functionality must be considered an essential requirement for future systems, alongside existing metrics such as performance, reliability and scalability.

9.2 Further work

There are several directions for further work in this area. Firstly, we have not considered how to adapt the kernel scheduler to a multi-processor system. Integrated scheduling with the non-CPU devices in the system is also a difficult problem which has not been addressed.

A more flexible security model would be necessary to grant and protect QOS facilities in a true multi-user environment. End-to-end QOS over networked RT systems is another logical development, particularly in the context of broadband Internet access, although this does depend on the availability of suitable RT operating systems.

Finally, there are sure to be many avenues of further research into QOS management policies, as the needs of RT desktop applications and users become apparent.

Bibliography

[Anderson90] "Lightweight Remote Procedure Call." Brian Bershad, Thomas Anderson, Edward Lazowska, Henry Levy. *ACM Transactions on Computer Systems.* Feb. 1990.

[Anderson92] "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism." Thomas Anderson, Brian Bershad, Edward Lazowska and Henry Levy. *ACM Transactions on Computer Systems.* Feb 1992.

[Angebranndt94] "Definition of the Porting Layer for the X v11 Sample Server." Susan Angebranndt, Raymond Drewry, Philip Karlton, Todd Newman, Bob Scheifler, Keith Packard, David Wiggins. *X Consortium Standard.*

[Barham96] "Devices in a Multi-Service Operating System." Paul Barham. *University of Cambridge Computer Laboratory Technical Report No. 403* July 1996.

[Bates98] "Using events for the scalable federation of heterogeneous components." John Bates, Jean Bacon, Ken Moody, Mark Spiteri. *Proceedings of the Eighth ACM SIGOPS European Workshop.* September 1998.

[Be99] BeOS. *Be, Inc.* Online, `http://www.beos.com`

[Beck98] "Linux Kernel Internals", Second Edition. Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verwoner. *Addison-Wesley,* 1998.

[Comer96] "Internetworking with TCP/IP", Second Edition, Volume 3, BSD Socket Version. Douglas Comer, David Stevens. *Prentice-Hall,* 1996.

[Dalheimer99] "Programming with Qt." Matthias Dalheimer. *O'Reilly,* May 1999.

[Ford96] "CPU Inheritance Scheduling." Bryan Ford, Sai Susarla. *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI).* October 1996.

[Gallmeister95] "POSIX.4: Programming for the Real World." Bill Gallmeister *O'Reilly,* January 1995.

[Gettys96] "Xlib - C Language X Interface, X Version 11, Release 6.3." James Gettys, Robert Schiefler. *X Consortium Standard.*

[Glauert91] "X Synchronization Extension Protocol, Version 3.0 (X11 R6.3)." Tim Glauert, Dave Carver, Jim Gettys, David Wiggins. *X Consortium Standard.*

[Goyal96] "A Hierarchical CPU Scheduler for Multimedia Operating Systems." Pawan Goyal, Xingang Guo and Harrick Vin. *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI).* October 1996. (See also `http://www.cs.umass.edu/~lass/software/qlinux/`).

[Ingram99] "Soft Real Time Scheduling for General Purpose Client-Server Systems." David Ingram. *7th Workshop on Hot Topics in Operating Systems (HotOS-VII).* March 1999. (See also `http://www.cl.cam.ac.uk/users/dmi1000/linux-srt/`).

[Jeffay96] "A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems". Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, Greg Plaxton. *Proceedings of the 17th IEEE Real-Time Systems Symposium.* December 1996.

[Jeffay98] "Proportional Share Scheduling of Operating System Services for Real-Time Applications." Kevin Jeffay, Donelson Smith, Arun Moorthy, James Anderson. *Proceedings of the 19th IEEE Real-Time Systems Symposium.* December 1998.

[Jones95] "Modular Real-Time Resource Management in the Rialto Operating System." Mike Jones, Paul Leach, Richard Draves, Joseph Barrera. *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V).* May 1995.

[Jones96] "An Overview of the Rialto Real-Time Architecture." Mike Jones, Joseph Barrera, Alessandro Forin, Paul Leach, Daniela Rosu, Marcel-Catalin Rosu. *Proceedings of the Seventh ACM SIGOPS European Workshop.* September 1996.

[Kaashoek95] "Exokernel: An Operating System Architecture for Application-Level Resource Management." Dawson Engler, Frans Kaashoek, James O'Toole. *Proceedings of the Fifteenth Symposium on Operating System Principles (SOSP).* December 1995.

[Kaashoek97] "Application Performance and Flexibility on Exokernel Systems." Frans Kaashoek, Dawson Engler, Gregory Ganger, Hector Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, Kenneth Mackenzie. *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP).* October 1997.

[Kay98] "A Fair Share Scheduler." J. Kay, Piers Lauder. *Communications of the ACM.* January 1988.

[Khanna92] "Real time scheduling in SunOS 5.0." Sandeep Khanna, Michael Sebree and John Zolnowsky. *Proceedings of the Winter 1992 USENIX Conference.* Jan 1992.

[Leffler89] "The Design and Implementation of the 4.3BSD UNIX Operating System." Samuel Leffler, Marshall McKusick, Michael Karels, John Quarterman. *Addison-Wesley,* 1989.

[Leslie96] "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications." Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, Eoin Hyden. *IEEE Journal on Selected Areas in Communications (JSAC).* September 1996.

[Liedtke99] "How to Schedule Unlimited Memory Pinning of Untrusted Processes or Provisional Ideas About Service-Neutrality." Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, Yoonho Park. *7th Workshop on Hot Topics in Operating Systems (HotOS-VII).* March 1999.

[Liu73] "Scheduling Algorithms for Multiprogramming in a Hard Real-Time environment." C. Liu, James Layland *Journal of the ACM.* Jan 1973.

[Lynx99] LynxOS. *Lynx Real-Time Systems, Inc.*
Online, `http://www.lynx.com`

[Mercer94] "Processor Capacity Reserves: Operating System Support for Multimedia Applications." Clifford Mercer, Stefan Savage, Hideyuki Tokuda. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems.* May 1994.

[Mercer96a] "Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach." Clifford Mercer, Chen Lee, Ragunathan Rajkumar. *Proceedings of Multimedia Japan.* March 1996.

[Mercer96b] "Predictable Communication Protocol Processing in Real-Time Mach." Chen Lee, Katsuhiko Yoshida, Cliff Mercer, Ragunathan Rajkumar. *Proceedings of the Real-time Technology and Applications Symposium.* June 1996.

[Mosberger96] "Making paths explicit in the Scout operating system." David Mosberger, Larry Peterson. *Proceedings of Operating Systems Design and Implementation (OSDI) 96.* October 1996.

[Mosberger98] "BERT: A Scheduler for Best-Effort and Real time Paths." Andy Bavier, Larry Peterson, David Mosberger. *Princeton University TR 602-99.* August 1998.

[Nieh97] "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications." Jason Nieh, Monica Lam. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP).* October 1997.

[OMG95a] "The Common Object Request Broker: Architecture and Specification", Revision 2.0. *OMG Standard.* July 1995.

[OMG95b] "CORBAservices: Common Object Services Specification", Revised Edition. *OMG Standard.* March 1995.

[QNX99] *QNX Software Systems Ltd.* Online, `http://www.qnx.com`

[Rubini98] "Linux Device Drivers." Alessandro Rubini. *O'Reilly,* February 1998.

[Scheifler94] "X Window System Protocol (X11 R6.3)." Robert Scheifler. *X Consortium Standard.*

[Sha90] "Priority Inheritance Protocols: An Approach to Real-Time Synchronisation." Lui Sha, Ragunathan Rajkumar, John Lehoczky. *IEEE Transactions on Computers.* September 1990.

[Solomon98] "Inside Windows NT", Second Edition. David Solomon. *Microsoft Press,* 1998.

[Srinivasan98] "A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software." Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, Douglas Niehaus. *Fourth IEEE Real-Time Technology and Applications Symposium.* June 1998. (See also `http://hegel.ittc.ukans.edu/projects/`).

[Stratford99] "An Economic Approach to Adaptive Resource Management" Neil Stratford, Richard Mortier. *7th Workshop on Hot Topics in Operating Systems (HotOS-VII).* March 1999.

[Sullivan99] "Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling." David Sullivan, Robert Haas, Margo Seltzer. *7th Workshop on Hot Topics in Operating Systems (HotOS-VII).* March 1999.

[Sun99a] ChorusOS. *Sun Microsystems, Inc.*
Online, `http://www.sun.com/chorusos`

[Sun99b] JavaOS. *Sun Microsystems, Inc.*
Online, `http://www.sun.com/javaos`

[Waldspurger94] "Lottery Scheduling: Flexible Proportional-Share Resource Management." Carl Waldspurger, William Weihl. *Proceedings of the First Symposium on Operating System Design and Implementation (OSDI).* November 1994.

[Waldspurger95] "Stride Scheduling: Deterministic Proportional-Share Resource Management." Carl Waldspurger, William Weihl. *Technical Memo MIT/LCS/TM-528, MIT Laboratory for Computer Science.* June 1995.

[Weis98] "K Desktop Environment Object Model: KOM/OpenParts." Torben Weis. August 1998. Online at `http://developer.kde.org`

[WindRiver99] VxWorks. *Wind River Systems, Inc.*
Online, `http://www.vxworks.com`

[Yodaiken97] "Introducing Real-Time Linux." Michael Barabanov, Victor Yodaiken.
The Linux Journal, Issue 34. February 1997.
(See also `http://luz.cs.nmt.edu/~rtlinux/`).