

# Soft Real Time Scheduling for General Purpose Client-Server Systems

David Ingram

*University of Cambridge Computer Laboratory  
Pembroke Street, Cambridge, CB2 3QG, United Kingdom*

dmi1000@cam.ac.uk

## Abstract

*This paper considers the problem of scheduling Real Time applications on a general-purpose Operating System. The approach taken is to modify a mainstream system, in order to realize the benefits of Real Time scheduling whilst preserving all existing software. Particular care has been taken to make important servers aware of their clients' Quality of Service, without major restructuring. We have implemented these ideas in a system called Linux-SRT.*

## 1. Introduction

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

A further distinction is drawn between Hard and Soft Real Time applications. A Hard RT application is one which will fail completely if any timing guarantee is not met. A Soft RT application is more tolerant and experiences only temporary failure if a short-term scheduling deadline is missed; it is not necessary to abort the application. We are concerned specifically with SRT systems.

Conventional UNIX schedulers are designed as “black box” systems which make reasonable decisions when dealing with BE tasks. The only parameter which can be adjusted is the “nice” setting. This is rarely satisfactory for prioritizing RT applications because its effect depends in a complex way upon the entire system load.

### 1.1. Quality of Service

An accepted basis on which to schedule RT tasks is to replace process priorities with a more general measure for

Quality of Service (QOS). QOS can be described by a pair of parameters such as (CPU share, time period). For example, a contract of (15%, 20ms) specifies that a task must be allocated at least 3ms of time in every 20ms interval (assuming that it is ready to run). This amounts to a guarantee on both throughput and timeliness.

Scheduling with QOS can be done using the Rate Monotonic (RM) or Earliest Deadline First (EDF) algorithms [8]. These methods can be shown to guarantee previously established QOS contracts, under certain assumptions (for example processes must be independent). Although not always true, these assumptions can be relaxed in order to adequately cover real systems. The scheduler is efficient because preemption occurs only when necessary, giving a timing granularity as fine as required, but no finer.

QOS scheduling is open, explicit and predictable, but has no built-in intelligence. The scheduler has to be told what parameters to use for each task, either by the user or by QOS management software.

### 1.2. Motivation

The most common computing platforms available today consist of personal computers or workstations. Most users are non-technical, and run pre-packaged software on standard hardware. In the past these platforms did not need advanced features such as a RT OS. The applications being run in this context now include both BE and RT processes, however. Furthermore, there are so many RT applications in common use that it is no longer sufficient to assume only a single RT process will be running at any one time (which could be reasonably handled by static priorities).

Applications which particularly benefit from RT scheduling include the following:

- Continuous Media (CM) processing: sound synthesis (MIDI, mp3), Internet audio (radio stations, phones),

video (conferencing, mail, movies on demand, news broadcasts)

- User interfaces (voice control, window managers)
- RT Games and Virtual Environments
- Applets and emulators (“temporal sandboxes”).

Note that in the last example we may be more concerned with *limiting* execution rates, rather than providing lower bounds. 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 a reliable ongoing rate based limit).

We can't always choose a purpose-built RT OS to run SRT applications. Users already try to run these applications on standard systems, with varying success. Applications such as videoconferencing are in fact likely to be *more* useful on standard desktops than on dedicated RT platforms. Additionally users want to run RT applications continuously whilst performing other activities (word processing, compiling, etc). We therefore require predictable behaviour whilst handling a mixture of simultaneous RT and BE processes.

### 1.3. Commercial Real Time Systems

Commercial RT systems to date have been tuned for fast response times by reducing context-switch delays and interrupt latencies. RT tasks are scheduled using static priorities, but there is no support for QOS. They have microkernel architectures to ensure a high degree of preemptivity, and priority inheritance is used to improve client-server interactions. Examples of this kind of system include QNX and LynxOS.

The lack of QOS makes it necessary to keep to a fixed task set which is known at system design time, and test that scheduling constraints are being met empirically, or through offline analysis. If this is not done, only a single RT task can be given guaranteed priority, which is inappropriate for the dynamic task set found on a PC environment.

Solaris [6] also has some RT features. Interestingly the kernel, although monolithic, is fully preemptive. This attractive structure seems to have been used more to achieve high degrees of SMP scalability than for RT purposes, though.

### 1.4. Problems

A conventionally structured OS presents several problems as far as implementing RT behaviour is concerned:

- Processes running in kernel mode cannot be preempted by more urgent ones in user space, or by others in kernel mode. Interrupts can also be disabled for long pe-

riods of time (up to  $400\mu s$  in the disk device drivers [13]).

- Server processes (such as the 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. SRT System Architecture

This section presents a design for a general purpose UNIX-compatible system which supports SRT applications. It addresses (or at least mitigates) the problems identified above.

### 2.1. Scheduling Servers

As was noted above, shared servers lead to poor scheduling whenever their clients have different scheduling requirements (which is *always* the case, even on a non-RT dynamic priority system). As a concrete example we shall examine the X server. This is particularly relevant because it can account for 50% of the load on a desktop machine running graphical applications.

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

- (a) Eliminate the server
- (b) Make the server multi-threaded
- (c) Perform local scheduling within a single-threaded server
- (d) Account retrospectively for short-term usage
- (e) Use priority inheritance

The first option is to limit the server's responsibility to initial access control only, removing it from the data path for most operations [11]. The majority of the functionality which was provided by the server is then performed in the client thread itself, through calls to library routines. For example, rendering code can be moved from the X server into the X library. Protection between clients is a problem, and has to be managed by allowing the access control server to install some trusted code or packet filtering into the device driver at connection establishment time. It's hard to do this efficiently and safely without custom hardware.

Multi-threaded servers are a good solution, and some X servers have been written this way. The approach is then to apply QOS contracts to groups of threads, called *activities*. For example, an X client and the thread responsible

for it within the X server fall into the same activity and are scheduled from the same resource allocation. A drawback is that servers may incur additional overhead performing IPC between their threads. Some restructuring of the server is obviously also required.

The remaining options address the problem of single-threaded servers directly. Implementing scheduling within a single-threaded server is quite easy. For example, The X protocol explicitly allows the request stream to be re-ordered, provided that, for each client, requests are processed in the order they arrived [12]. Thus it is possible to give certain clients preferential treatment without violating the protocol, and this is the method we have applied to the X server.

## 2.2. Kernel Scheduler

Our architecture makes use of standard POSIX RT scheduling, which provides high static priorities. Only a few changes to the POSIX scheduler are required in order to support QOS with Rate Monotonic (RM) scheduling. This is because the RM algorithm is based upon static priorities (which also makes it attractive for scheduling servers, because they do not have to be informed of any priority changes). *Admission control* code can automatically choose these priorities in such a way that the tasks with shorter time periods have higher priorities.

RM scheduling is known to have a worst case schedulability of 69% [8]. This means that no tasks will miss their deadlines provided the control program only admits RT tasks whose combined load is less than 0.69. All remaining time can of course be used for BE applications, which will continue to make up a significant fraction of the overall load.

The two modifications required to enable QOS are (i) *accounting*, and (ii) *policing*. Accounting in UNIX consists of noting which process is active at each clock tick (every 10ms). Because this is so infrequent, the estimates for CPU consumed are only valid over very long time spans. Much more accurate accounting can be provided by reading the processor's Time Stamp Counter each time a reschedule is done. Policing is the capability to preempt processes immediately if they have not blocked before their CPU allocation for the current time period is exhausted. Aperiodic timers such as those provided by UTIME [13] can be used to implement this feature.

## 2.3. Use of slack time

BE tasks are scheduled in the normal way whenever no RT task with contracted time left is ready to run. This is important to ensure exact reproduction of familiar scheduling behaviour, so that legacy systems do not require re-tuning.

The 69% limit on RT task reservations required by the RM algorithm ensures that starvation of the BE tasks cannot occur. Unallocated time also subsumes scheduler overheads.

A real-time task is said to be *running optimistically* [11] if it has exceeded its current QOS allocation but the kernel has decided to schedule it anyway. Each RT process has a flag which indicates whether it should be considered for this, or suspended until the next time interval instead. The latter option is useful if QOS is being used to *limit* the CPU consumption of a process to a fixed amount, the former if we are trying to provide a *minimum* level of service for a task.

Optimistic RT tasks do 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 take outright priority over optimistic RT ones, or a long background job would prevent any optimistic scheduling for the (presumably important) RT tasks. The solution is to temporarily demote RT tasks which have overrun to BE status, placing them under the remit of the standard BE scheduler.

## 2.4. Scheduler synchronisation

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.

In our system the priorities are static; the principal state which needs to be communicated from the kernel (master) scheduler to the server (slave) schedulers is the list of clients that have overrun their time contracts for their current period.

After servicing each request, a single-threaded server such as X calls the kernel requesting current scheduling information. This is done through a new system call. The kernel returns a list of clients which have currently been policed and must therefore *not* be scheduled by the X server (even if no other clients have pending requests). It also indicates whether it is currently running tasks optimistically, and if so for which tasks this is a valid thing to do. In this case the X server may process requests optimistically as well, on a FCFS basis. If this were not done, slack time could not be used for RT graphics.

When the X server polls the kernel status, it passes as an argument the PID of the client which it just completed a request for. The kernel can then compute the CPU time the X server has spent since it last made this call. All of this time can be retrospectively accounted to the reserve for the client which was just serviced (assuming it was a RT process). This ensures that graphics processing is not ex-

empt from QOS accounting. The X server's book-keeping overheads are shared between clients.

Retrospective accounting works well because the average X request service time is very short (see Table 1). This is no accident because single-threaded servers are typically used for efficiency where requests are short but numerous. Some QOS reserves may go slightly below zero before pre-emption can occur as a result; this is ignored (negative reserves could be retained to increase fairness slightly). Short service times also ensure that imprecision caused by the lag between two schedulers is not great.

The X server itself must be run as a maximum priority RT task whenever there is at least one client which is a RT task, with guaranteed time available, and outstanding X requests.

No QOS contract or policing is applied to the X server because it cannot be determined in advance what its requirements will be. Fortunately this is not necessary because time is accounted to client reserves, which are themselves policed. When there are no RT clients with outstanding requests the X server is scheduled as a normal task. Thus it is treated exceptionally only when doing guaranteed RT work, and should exhibit familiar behaviour in all other cases. Outstanding requests from RT tasks running optimistically do not cause the X server to enter high-priority mode.

### 3. Implementation

A prototype implementation called *Linux-SRT* has been constructed to evaluate these ideas. It is based upon a standard Linux [3] system, with modifications to the kernel and X server, together with user-space control programs.

Currently the X server scheduling has been implemented, but the standard kernel scheduler is still used (so the prototype cannot yet validate our model for synchronising the two schedulers).

*Linux-SRT* runs all existing Linux software with normal performance and stability. It does not have to be switched into a special RT "mode", and RT processes have access to all normal services and APIs. Any program can have a quality of service associated with it and be scheduled as a RT process, without the author needing to anticipate this or add any periodic hooks to their code.

We have examined the system's behaviour using a typical range of software (web browsers, office suites, movie players and a raytracer) in addition to simple benchmarks. *Linux-SRT* is used as our normal platform for development and every-day computing tasks.

#### 3.1. Linux-SRT X server

The following sequence of events occurs when a new X client connects to the server:

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 connection, and extracts the port field.
4. The `proc` filesystem is read to discover which process opened this connection.
5. `getpriority` is called to look up the POSIX priority for this client.
6. The server sets the X priority for this connection to the same value as the POSIX priority.

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. To make the reverse lookup possible, the implementation extends the information exported by the kernel via the pseudo-file `/proc/net/tcp` so that it includes the process identifiers of socket creators.

The X server then prioritises all subsequent requests from this client. The mechanism is similar to that used by the X Synchronization Extension [4]. The priority can be changed by the control program at any time.

#### 3.2. Management components

A control program is provided in order to set priority levels in the system. It adjusts POSIX RT, Linux and X server priorities via the QOS parameters. The control program can launch new processes with specified QOS and attach to currently running processes.

All aspects of the RT system can also be controlled through a GUI which we have based on the KDE process manager. We are also experimenting with binding the control program to buttons on window title bars (via some window manager extensions) to allow more direct manipulation.

A simple but useful form of non-interactive QOS management is provided by a modified `exec` system call, which applies default QOS parameters to programs at startup. These values are read in from a configuration file.

### 4. Evaluation

We believe that *Linux-SRT* overcomes three barriers which have hindered widespread use of previous RT systems:

- (i) binary compatibility with all software
- (ii) *simplicity* and ease of implementation

- (iii) provision of management tools and automation (“How do I actually make use of this facility?”)

Before implementing Linux-SRT, we made various performance measurements on a normal Linux system to assess the feasibility of our method. These measurements were made on a 133 Mhz Pentium PC, based on long-term normal usage of the system.

**Table 1:** Timing measurements

Operation	Duration
“do-nothing” syscall overhead	1.7 $\mu$ s
gettimeofday syscall	2.8 $\mu$ s
median X request service time	1.5 $\mu$ s
mean X request service time	6.0 $\mu$ s

The time taken to perform the `gettimeofday` syscall was important, since this is used by the modified X server to obtain timestamps at roughly microsecond accuracy. Allowances for this were made in computing the other results. Note that X requests are in fact serviced very quickly, the mean time being 6 $\mu$ s. Process scheduling intervals are three orders of magnitude longer than X requests.

#### 4.1. Results

In order to test the system, 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 set to maximum POSIX RT priority, with its X priority increased, and then with both. In each case the performance of the boosted application and of the others was measured.

It was found that for this test POSIX RT priority made no difference at all. However, the X priority made a significant difference (see Table 2).

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

### 5. 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 realistic work loads, or indeed whether the right problems are being solved.

**Table 2:** Times to play video clip (in seconds)

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

RT-Mach is an extension of the Mach [1] microkernel. The RT scheduling additions are known as Processor Capacity Reserves (PCR) [9, 10]. PCR provides QOS and implements RM scheduling. Unfortunately applications running under Mach’s UNIX emulation layer are controlled by the UNIX scheduler, and cannot make use of PCR.

Nemesis is a *vertically structured* [11] OS. In this architecture shared servers are avoided, and data-path processing is moved into user-space libraries where possible. This restructuring makes porting applications difficult, as discussed in §2.1.

Rialto is a RT OS which uses a Least Slack Time First scheduler [5]. Partial compatibility with the Win32 API is provided, but applications still need modifications to run on this platform.

An alternative to QOS as presented here is Proportional Share Scheduling [14], which has been implemented under FreeBSD.

#### 5.1. Linux-based systems

Two existing systems (RT Linux and KURT) have been based on the Linux platform. RT Linux [2] provides a minimal RT kernel, which runs the Linux kernel itself as a sub-task when no RT task requires the processor. If the Linux kernel tries to disable interrupts, RT Linux emulates this in software. This avoids the reduction in timing granularity for the RT tasks which is normally caused by suspending hardware interrupts.

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. However, it is not suitable for general purpose RT computing due to the restrictions placed on RT tasks.

The KURT system [13] is better integrated with standard Linux, allowing RT processes to use normal kernel services. It works by programming the timer chip to interrupt aperiodically, in order to overcome the kernel's fixed 10ms timing resolution. The usual 10ms ticks are simulated so that the rest of the system is unaffected.

KURT uses an explicit plan scheduler. Although it does have a "periodic mode" the applications must still cooperate by calling KURT functions, requiring source-level modifications.

## 6. Future Work

There are many interesting avenues for future work on this platform. For example, a more flexible security model is required to safely grant limited QOS facilities to normal users on a multi-user workstation. Consideration also needs to be given to the use of QOS on multi-processor (SMP) machines. To this we can add some interesting networked cases, since 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.

Resources other than the processor[s] also need to be scheduled in order to create an effective RT system. These include memory, network bandwidth, and dedicated graphics acceleration hardware. For example, the memory resource can be effectively scheduled by requesting lower and upper bounds on the number of *physical* pages a particular process may use simultaneously.

More work also needs to be done on QOS Management. It is important that QOS parameters be selected automatically in most cases, both in order to be sufficiently non-intrusive for everyday use and to cater for non-expert users.

## 7. Conclusion

We have shown that it is possible to support effective SRT scheduling on a conventionally structured, general purpose operating system. Even single-threaded servers can be modified to respect client priorities; we have demonstrated this technique with the X server. Our system is in everyday use and required small changes to the OS but no changes to applications. Now that RT applications are commonplace it is important that desktop systems provide such facilities.

## Acknowledgements

This work was supported by the UK EPSRC. I am grateful to my supervisor, Jean Bacon and to Ken Moody for reading drafts of this paper. I would also like to thank those working on the Nemesis Project for some interesting discussions.

## References

- [1] M.Accetta, R.Baron, D.Golub, R.Rashid, A.Tevanian, M.Young. "Mach: A New Kernel Foundation for UNIX Development". *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-112.
- [2] M. Barabanov, V. Yodaiken. "Introducing Real-Time Linux". *The Linux Journal*, Issue 34, Feb. 1997.
- [3] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verwoner. *Linux Kernel Internals*, 2nd Ed. Addison-Wesley, 1998.
- [4] T. Glauert, D. Carver, J. Gettys, D. Wiggins. *X Synchronization Extension Protocol V3.0 (X11 R6.3)*. X Consortium Standards Document.
- [5] M.Jones, J.Barrera III, A.Forin, P.Leach, D.Rosu, M.Rosu. "An Overview of the Rialto Real-Time Architecture". *Proceedings of the Seventh ACM SIGOPS European Workshop*, Sept. 1996.
- [6] S.Khanna, M.Seabee, J.Zolnowsky. "Realtime scheduling in SunOS 5.0". *Proceedings of the Winter 1992 USENIX Conference*, pp375-390, San Francisco, CA, Jan 1992.
- [7] S. Leffler, M. McKusick, M. Karels, J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [8] C. Liu, J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Journal of the ACM*, Vol. 20, No. 1 Jan 1973.
- [9] C. Mercer, S. Savage, H. Tokuda. "Processor Capacity Reserves: Operating System Support for Multimedia Applications". *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.
- [10] C. Mercer, R. Rajkumar. "An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management". *Proceedings of the Real-Time Technology and Applications Symposium*, May 1995.
- [11] T. Roscoe. *The Structure of a Multi-Service Operating System*. Cambridge University Computer Lab Technical Report 376. Aug 1995.
- [12] R. Scheifler. *X Window System Protocol (X11 R6.3)*. X Consortium Standards Document.
- [13] B. Srinivasan, S. Pather, R. Hill, F. Ansari, D. Niehaus. "A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software". *Fourth IEEE Real-Time Technology and Applications Symposium*, June 1998.
- [14] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, C. Plaxton. "A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems". *17th IEEE Real-Time Systems Symposium*, December 1996.