Appeared in *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 30-June 1, 2001.

Two Case Studies in Predictable Application Scheduling Using Rialto/NT

Michael B. Jones
Microsoft Research, Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
mbj@microsoft.com
http://research.microsoft.com/~mbj/

John Regehr
Department of Computer Science, Thornton Hall
University of Virginia
Charlottesville, VA 22903-2242, USA
john@regehr.org
http://www.cs.virginia.edu/~jdr8d/

Stefan Saroiu
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350, USA
tzoompy@cs.washington.edu
http://www.cs.washington.edu/homes/tzoompy/

Abstract

This paper analyzes the results of two case studies in applying the Rialto/NT scheduler to real Windows 2000 applications. The first study is of a soft modem—a modem whose signal processing work is performed on the host CPU, rather than on a dedicated signal processing chip. The second is of an audio player application. Both of these are frequently used real-time applications—ones running on systems that were not designed to support predictable real-time execution. To function correctly, both applications require that ongoing computations be performed in a timely manner. In both cases, we first measured an original version designed to run on Windows 2000, and then modified the application to take advantage of ongoing CPU Reservations provided by the Rialto/NT scheduler. We report on the benefits and problems observed when using reservations in these real-world scenarios. In both cases, we found that a real-time scheduler can provide the needed predictability for the application in the presence of competing applications, while also providing other benefits, such as minimizing the soft modem's impact on the scheduling predictability of other computations in the system. We also describe the methodologies we used to analyze the real-time behavior of the operating system and applications during these studies, including the use of instrumented kernels to produce execution traces.

1. Introduction

Novel implementations of two real-time scheduling abstractions were developed within the Rialto real-time operating system [2]: *CPU Reservations* and *Time Constraints*. These abstractions allow activities to obtain minimum guaranteed execution rates with application-specified reservation granularities via CPU Reservations, and to schedule tasks by deadlines via Time Constraints, with on-time completion guaranteed for tasks with accepted constraints.

We implemented these abstractions within a research version of Windows 2000 called Rialto/NT [4]. While implementing the Rialto scheduling abstractions in Rialto/NT involved solving several interesting engineering and research problems, this work just provides a means to larger ends. The main goal of Rialto/NT has always been to bring the benefits of predictable scheduling to Windows 2000 applications. This paper presents results obtained when applying Rialto/NT's CPU Reservations to two different applications designed for Windows 2000 that require predictable execution to function correctly: a ***soft modem*** and an ***audio player***.

The remainder of this paper is structured as follows: Section 2 presents the background behind these studies. Section 3 describes the methodology and tools used. Section 4 is the Soft Modem application study. Section 5 is the Audio Player application study. Section 6 presents related work. Section 7 discusses possibilities for further related research. Finally, Sections 8 and 9 present the contributions and conclusions from these studies.

2. Research Background

2.1 Commodity Operating Systems and Real-Time Applications

General-purpose operating systems such as Windows 2000, Linux, and Solaris are increasingly being used to run time-dependent tasks such as audio and video processing despite good arguments against doing so [7]. This is the case even though many such systems, and Windows 2000 in particular, were designed primarily to maximize aggregate throughput and to achieve approximately fair sharing of resources, rather than to provide low-latency response to events, predictable time-based scheduling, or explicit resource allocation. Nonetheless, since these systems are being used for time-dependent tasks, it is important to understand both their capabilities and limitations for such applications.

2.2 Windows 2000 Scheduling Structure

Windows 2000 scheduling is described in detail in [8]; a brief overview follows.

Windows 2000 has 31 priority levels. Priorities 1-15 are *variable* levels; thread priorities in this range are dynamically adjusted to increase responsiveness. For example, quantum size is increased for threads in the foreground process, priority may be boosted upon wakeup, and priority is boosted for threads that have been ready to run, but not scheduled, for several seconds. The latter heuristic is designed to break priority inversions by giving starved threads a chance to release shared resources they may be holding. This heuristic is effective, although we

will see in Section 5.4.2 that inversions are not broken quickly enough to be useful for multimedia applications.

Priorities 16-31 are *real-time* priorities. Quanta and priorities of threads in this range are not adjusted—the scheduler simply runs the threads at the highest priority in a round-robin manner.

2.3 Rialto/NT Real-Time Scheduling

Rialto/NT [4] was designed and built to combine the benefits of today's commodity operating systems with the predictability of the best soft real-time systems. Rialto/NT supports simultaneous execution of independent real-time and non-real-time applications. These goals are achieved by computing a deterministic schedule that meets the declared requirements of all admitted real-time tasks whenever the set of real-time applications changes.

In Rialto/NT, threads make ***CPU Reservations*** to ensure a minimum guaranteed execution rate and granularity. A CPU Reservation request is of the form *reserve X units of time out of every Y units for thread A*—requesting that for every time interval of size *Y*, thread *A* be scheduled for at least *X* time units, provided it is runnable. For example, a thread might request 800µs every 5ms, 7.5ms every 33.3ms, or one second every minute.

The current implementation of Rialto/NT has two restrictions: (1) CPU reservations must have values that are integer multiples of milliseconds, since they are driven off the periodic Windows 2000 clock and (2) the period of a reservation must be a power-of-two multiple of a millisecond, due to a choice of algorithms within Rialto/NT. Rialto/NT schedules a thread by raising the thread's effective priority as seen by the Windows 2000 scheduler to 30 (the second highest priority in the system).

Rialto/NT also implements deadline-based ***Time Constraints*** which, although not used in this paper, are also described in [4].

3. Methodology and Tools

3.1 Instrumented Windows 2000 Kernel

We took measurements using a "perf kernel"—an instrumented version of Windows 2000 that was developed in order to understand and tune the OS. The perf kernel is capable of logging a wide variety of events to a physical memory buffer and then dumping them to disk for post-processing. During our experiments, we used predefined perf kernel functionality to log all deferred procedure calls (DPCs), thread context switches, thread and process creations and deletions, and synchronization events. We also logged application-specific data such as modem hardware register values and audio starvation events.

The instrumented kernel offers the same performance as a regular kernel when no events are being logged. Furthermore, logging an event is fast: it took 549ns on the 450 MHz Pentium II used in the soft modem study.

Logging produced around 10MB of data per minute. After dumping the binary event logs to disk and converting them into a text format, we post-processed the output with Perl scripts that filtered out uninteresting data and converted the remainder into a more readable format.

3.2 Remoteres

We implemented a small program called **remoteres** that is able to begin and end CPU Reservations for any thread in the system. Using this simple tool, we gave reservations to various threads and watched what happened when there was contention. **Remoteres** was very useful as a tool for experimenting with real-time performance: not only did it keep recompiles out of our critical path, but we could also try out different reservations without even restarting an application.

4. Soft Modem Application Study

Soft modems use the main CPU to execute modem functions traditionally performed by a digital signal processor (DSP) on the modem card. Soft modems have enjoyed large success in the home computer market. Two reasons for this are low cost and the flexibility of migrating to newer technologies by simple software upgrade.

This study presents detailed performance characteristics and resource requirements of a popular soft modem. Unfortunately, insufficient space was allowed to include the full study results, but they are summarized below. A detailed presentation can be found in [5].

Given recent advances in CPU processing power, the impact of a soft modem on the throughput of the system is reasonable—we measured a 14.7% sustained CPU load on a 450 MHz Pentium II. Because soft modems need periodic real-time computations on the host CPU in order to maintain line connection and transmit data, a mechanism ensuring predictable scheduling is essential.

While the soft modem's 14.7% CPU load is not high *per se*, a problem with the vendor version is that all of this time is spent in interrupt context. Once connected, the execution of the interrupt handler typically lasts 1.8ms with a repeatable worst case of 3.3ms during connection.

This study shows that signal processing in interrupt context is not only unnecessary but also detrimental to the predictability of other computations in the system. While DPCs and priority-based scheduling cause milder side effects upon the rest of the system, they nevertheless suffer from some of the same drawbacks as the original version. This study supports the conclusion that CPU Reservations provide a good answer to the predictability problems that can be caused by a soft modem.

5. Audio Player Application Study

5.1 Experimental Setup

All audio application performance results were measured on a dual-processor 333 MHz Pentium II PC with 128MB of memory. Although the machine normally uses both processors, all results reported in this study were run in uniprocessor mode.

5.2 Windows Media Player

Windows Media Player plays a variety of streaming audio and video file formats such as MP3, WAV, AVI, and MPEG-2. All experiments reported in this section were performed while playing an MPEG-2 layer 3 (MP3) audio file under Media Player version 6.4. We chose an audio application because the human ear is very sensitive to anomalies in audio playback; in this domain we expect essentially flawless real-time performance.

The Windows Media Player is structured as a group of cooperating threads that performs tasks such as reading encoded data from disk, decoding the data and sending it to a sound driver, and updating the graphical front-end.

5.2.1 Windows Audio Architecture

Windows 98, Windows ME, and Windows 2000 contain the Windows Driver Model audio architecture [6] that performs mixing functions in software, so that a potentially unlimited number of software sound sources can be converted into a single stream for delivery to sound hardware. The kernel audio mixer has tight end-to-end latency requirements since applications may generate sounds in response to user actions. If the delay between action and sound is longer than a few tens of milliseconds, they are not perceived as being simultaneous.

The kernel mixer uses three or four 10ms buffers. Consequently, if the kernel mixer thread (which should run every 10ms at priority 24) is not scheduled for about 30ms, audible glitches will follow. We added code to the kernel mixer causing it to emit a "kernel mixer starvation event" to the perf kernel log when it ran out of data; these appear in some of our execution traces (Figures 5-2, 5-4, and 5-5). This was useful because the kernel mixer is the most latency sensitive part of the Media Player, and sound glitches were virtually guaranteed to happen when it starved. However, while kernel mixer starvation was a sufficient condition for glitches, it was not necessary.

5.2.2 Media Player Thread Structure

Period (ms)	Priority	Name
10	24	Kernel Mixer
45	8	User Interface
100	15	Multimedia Timer
100	9	MP3 Decoder
500	8	Unknown
2000	8	Disk Reader

Table 5-1: Media Player thread structure

Media Player creates five threads while playing an MP3. Four of these threads and a kernel mixer thread will concern us for the next few sections.

Kernel mixer thread: The kernel mixer thread runs every 10ms at priority 24. It is latency-sensitive, and will cause sound glitches if starved for more than 25-30ms.

User interface thread: A priority 8 Media Player thread runs every 45ms in order to control and update the Media Player's user interface. It is always awakened by a priority 19 CSRSS thread. (CSRSS is a system server that, among other jobs, performs console I/O.) When this thread is starved, Media Player only updates its GUI every three seconds or so, when the Windows 2000 starvation avoidance logic boosts its priority.

Timer thread: A multimedia timer thread runs every 100ms at priority 15. It awakens the MP3 decoder thread.

MP3 Decoder thread: A priority 9 Media Player thread runs every 100ms. Most of the Media Player's CPU time is spent in this thread decoding MP3 data. It is not very latency-sensitive—after being starved briefly, it runs for long enough to catch up when next scheduled.

Disk I/O thread: A priority 8 thread wakes up every 2000ms in order to read MP3 data from disk.

5.3 Audio Study Experiments Run

Our testing strategy was to listen to an MP3 audio stream using the Windows Media Player under various conditions. For purposes of this experiment, we consider the Media Player to be working if there were no audible glitches or detected kernel mixer buffer starvations during a 1-minute period. Although we report only on a single trial of each experiment, we repeated them enough times to verify that the results reported are typical.

We chose to use audible glitches as our principal application quality metric because some Media Player tasks have enough internal buffering that there is not always a strong correspondence between missed task deadlines and degradation in audio quality. Therefore, number of missed deadlines, average task lateness, and other traditional metrics would not accurately measure what we are actually interested in: the relationship between scheduling predictability and perceived application quality.

We modeled contention with CPU intensive applications by writing a simple program that spins at a given priority while the Media Player is running. Table 5-2 lists, for each experiment, the conditions under which Media Player was run, and the resulting behavior.

Experiment #	Competitor Thread Priority	Decoder Thread Reservation	Kernel Mixer Thread Reservation	Audible Glitches	Kernel Mixer Starvations Detected
1	-	-	-	0	0
2	8	-	-	0	0
3	10	-	-	many	many
4	9	-	-	4	many
5	10	40/1024	-	4	many
6	10	40/1024	1/16	0	0
7	10	20/512	-	0	0
8	10	1/16	-	0	0
9	9	1/16	-	1	0

Table 5-2: Experiments run

Experiment 1: *No competitor—Media Player running by itself.* With no contention everything worked fine.

Experiment 2: *Priority 8 competitor.* Result: It works fine. Explanation: The priority 8 Media Player threads do not need much CPU time, so operating with a competitor at the same priority presents no problem.

Experiment 3: *Priority 10 competitor.* Result: The Media Player doesn't work at all. Only short bursts of music are heard every 5 seconds or so. Explanation: Several important Media Player threads run at priorities less than 10; these are almost completely starved by the priority 10 competitor and only get to run every 5 seconds or so when the Windows 2000 starvation avoidance logic boosts them to a high priority for a few quanta.

Experiment 4: *Priority 9 competitor.* Result: There were three ~0.5s dropouts and one 4-second dropout. 373 kernel mixer starvations were logged. Explanation: bugs in Media Player, which we discuss in Section 5.4.2, caused the dropouts.

Experiment 5: *Priority 10 competitor. Media Player decoder thread has a reservation of 40ms/1024ms.* Result: There were 3 barely audible glitches and one obvious one. Explanation: The kernel mixer starves several times because, during its reserved time, the decoder thread runs for long enough to make the kernel mixer thread miss its deadlines. This is discussed in Section 5.4.2.

Experiment 6: *Priority 10 competitor. Media Player decoding thread has a reservation of 40ms/1024ms; kernel mixer thread has a reservation of 1ms/16ms.* Result: It works fine. Explanation: There are two effects here. One is that because of the reservation, the kernel mixer cannot be starved by a boosted Rialto/NT thread. The other is that the kernel mixer reservation causes the reservation for the decoder thread to be fragmented—this means that it receives CPU time more evenly than when it is the only reservation in the system.

Experiment 7: *Priority 10 competitor. Media Player decoding thread has a reservation of 20ms/512ms.* Result: It works fine. Explanation: The decoder thread runs often enough that it doesn't run very long at priority 30, and therefore doesn't interfere with the priority 24 kernel mixer thread.

Experiment 8: *Priority 10 competitor.* Media Player decoding thread has reservation of 1ms/16ms. Result: It works fine. Explanation: Same as previous experiment.

Experiment 9: *Priority 9 competitor. Media Player decoding thread has a reservation of 1ms/16ms.* Result: 1 audible glitch. Explanation: The Media Player decoder thread fails to decode enough data because of a bug in the Media Player; we discuss this in Section 5.4.2.

5.4 Analysis of Audio Study Results

In general, the results of our experiments were as expected: in the presence of contention the priority-based scheduler did not give enough CPU time to the Windows Media Player, but when application threads were given appropriate reservations they were able to meet their deadlines. However, we also encountered some interesting and unexpected situations.

5.4.1 Results We Expected

Although experiment 1 generated no surprises, we include its execution trace as a baseline in Figure 5-1. Note that the CPU spends most of its time running the idle thread, the kernel mixer thread runs every 10ms, and the Media Player threads have a regular timing structure.

Experiment 3 also offered few surprises. In competition with a priority 10 thread, the Media Player threads were not able to run most of the time. Figure 5-2 shows a long stream of starvation messages that are interrupted

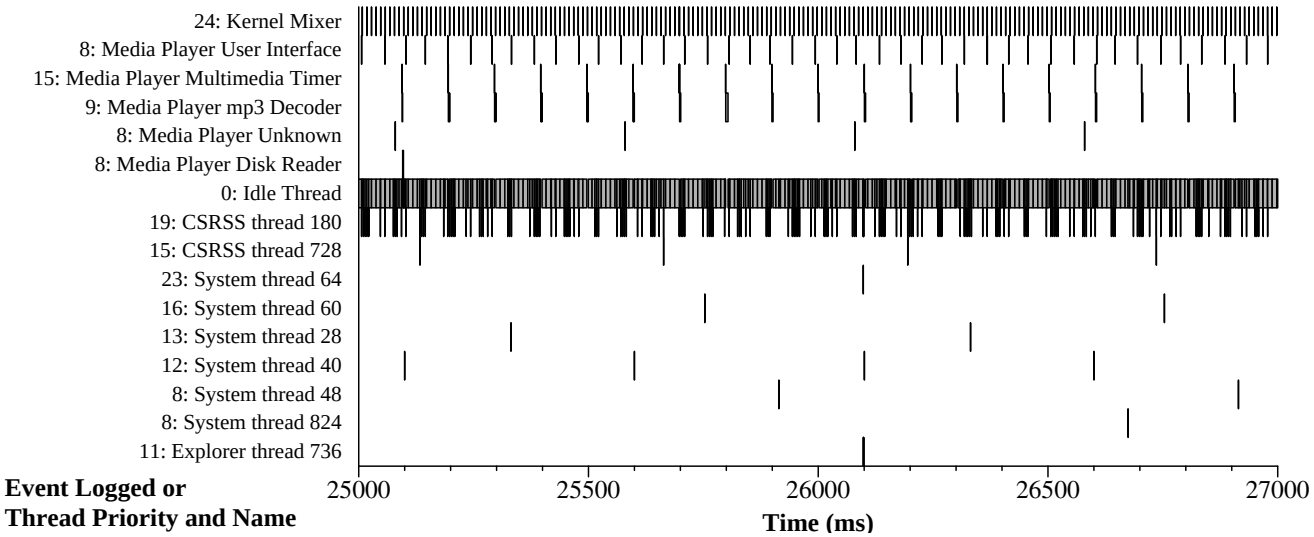

Figure 5-1: Execution trace gathered during experiment 1: Media Player with no contention

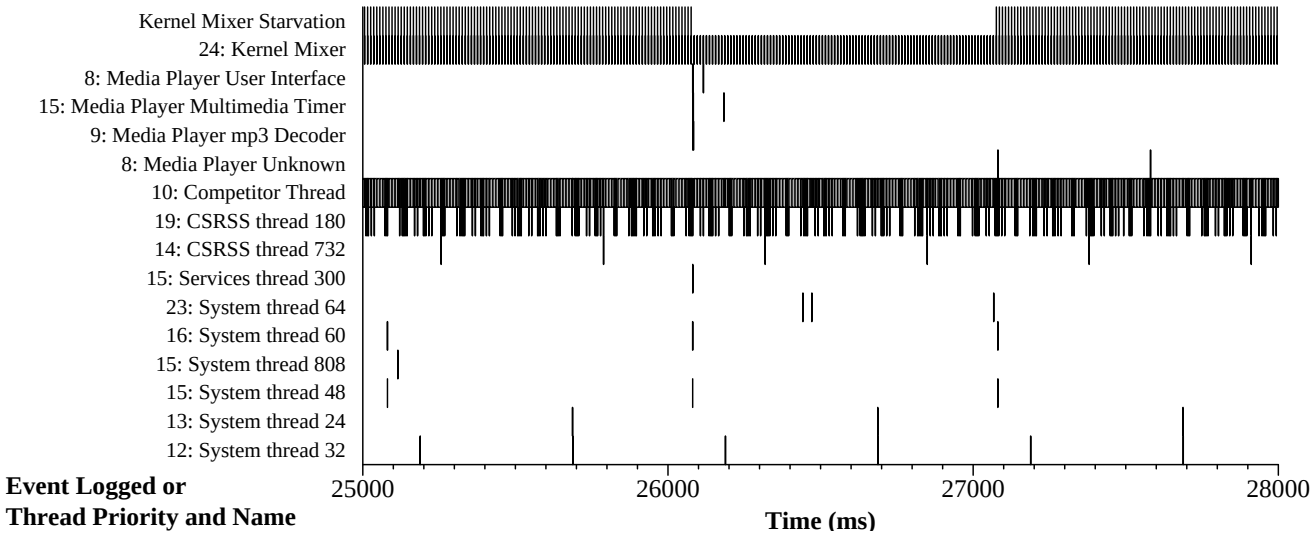

Figure 5-2: Execution trace gathered during experiment 3: Media Player being starved by a priority 10 competitor

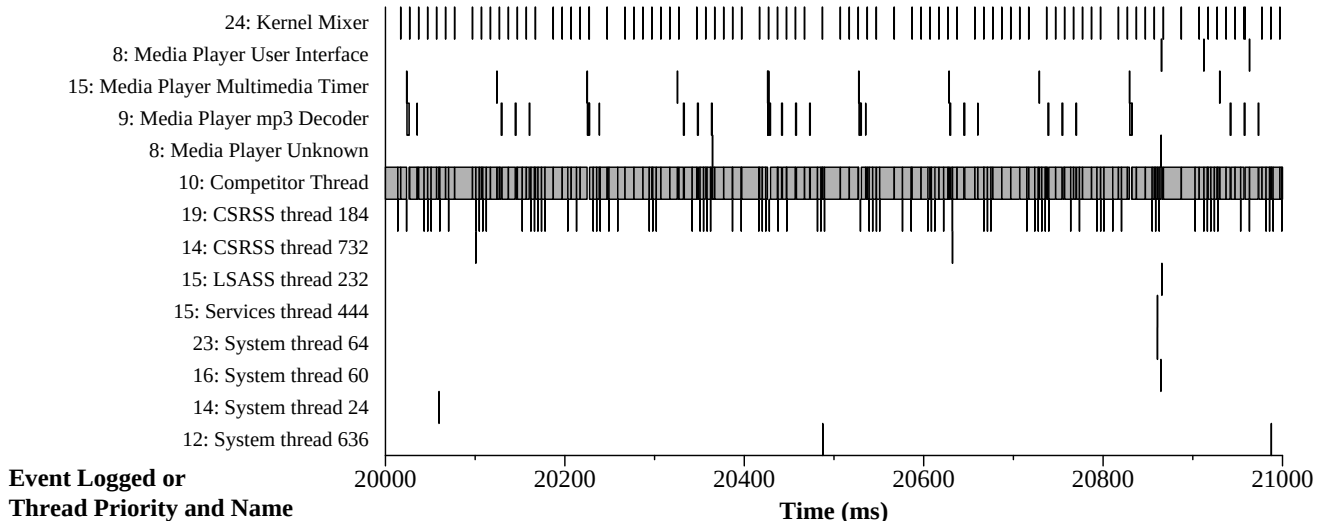

Figure 5-3: Execution trace from experiment 8: Media Player decoder thread has a 1ms/16ms reservation, while competing with a priority 10 thread

just after 26 seconds into the run when the starvation avoidance logic boosts the priority of the starved Media Player threads—they run briefly and then resume starving. The sound that this experiment produced was a long sequence of clicks and pops with brief bursts of music when the application was able to run.

It is interesting to compare the pattern of thread executions in Figure 5-3 (experiment 8) with the ones in Figure 5-1 (experiment 1). The orderly time-slices are gone, replaced with an interference pattern between the 100ms "natural" period of the MP3 decoder thread and the 1ms/16ms reservation that we gave it. The multimedia timer expirations are still orderly. This is because Windows multimedia timers run at priority 15 and therefore always immediately preempt the competitor thread. The timer thread runs only long enough to awaken the decoder thread, which runs in several subsequent time-slices since the 1ms slots are not individually long enough for it to complete its work.

5.4.2 Results We Did Not Expect

By giving the Media Player decoding thread a reservation, we were able to ensure that it was allocated sufficient processing time. In experiment 5, we gave it a reservation of 40ms/1024ms—this is much longer than its normal period, but short enough that it was able to keep its buffer of decoded data from becoming empty. However, each time it ran, it ran for so long (while boosted to priority 30 by Rialto/NT) that it starved the kernel mixer thread! This shows that giving reservations to some real-time threads and not others is potentially dangerous: it is possible to make the situation worse instead of better.

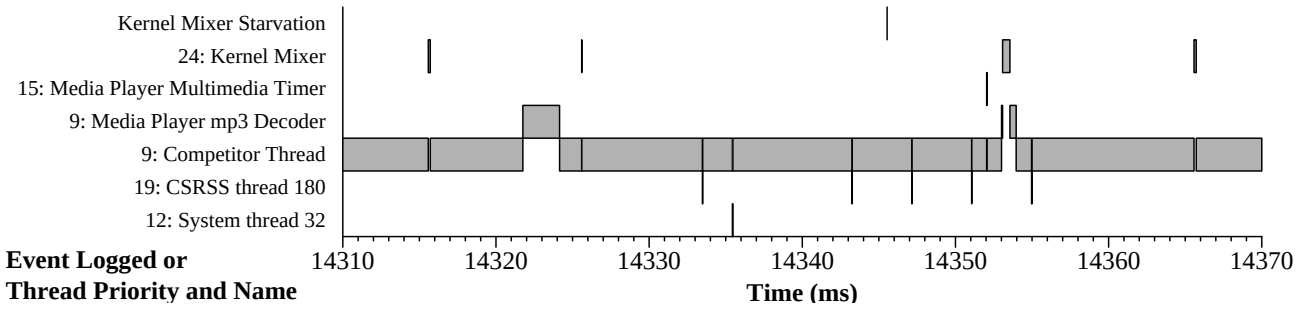

Figure 5-4: Execution trace from experiment 4: a priority inversion: thread 708 is blocking the higher priority thread 772 between times 14325 and 14353

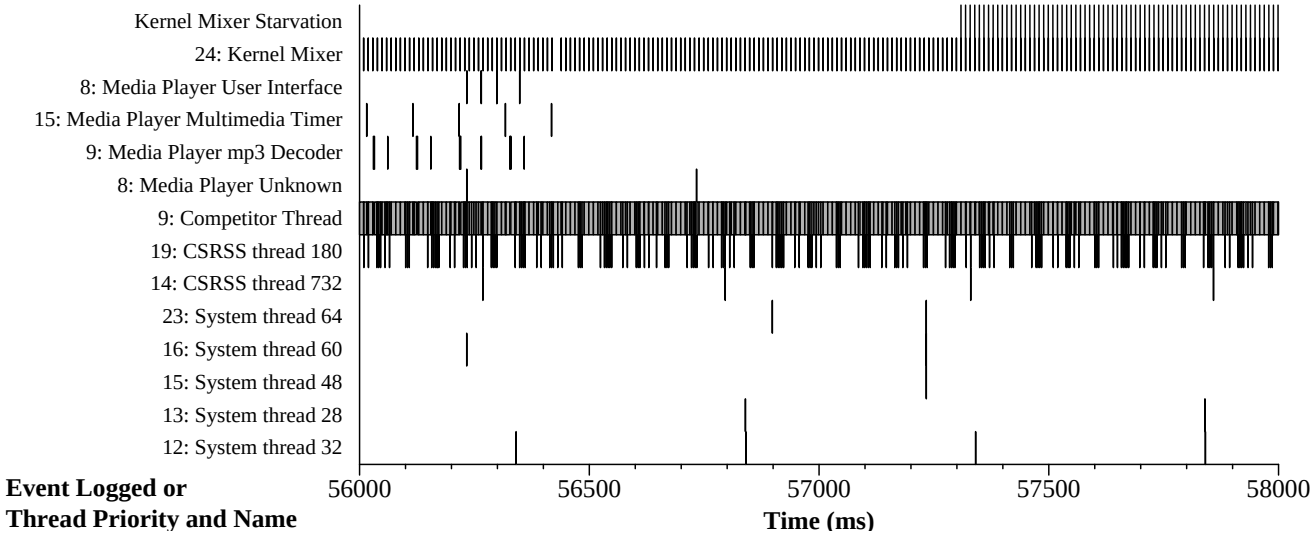

Figure 5-5: Execution trace from experiment 4 showing a deadlock

In experiments 4 and 9, buffer under-runs or audio glitches were detected even when we would have expected Media Player to work. These can be traced to at least two bugs in the Media Player implementation. The more interesting bug is a priority inversion: the kernel mixer thread and the user-level decoding thread both frequently get or set the current position in the audio stream; the stream data structure is protected by a blocking mutex. If the lower-priority decoder thread is preempted while holding this lock, the kernel mixer thread will block on the mutex when it next tries to enter the critical section.

Figure 5-4 shows an example taken from experiment 4, in which the decoder thread (at priority 9) competes for the CPU with a priority 9 spinning competitor thread. A priority inversion begins around 14324 milliseconds into the run when the competitor thread preempts the Media Player MP3 decoder thread while it is holding the lock it shares with the kernel mixer thread. At around 14325, the kernel mixer thread wakes up and blocks on the mutex almost immediately; it subsequently misses its next two invocations—this causes a buffer under-run to occur at time 14345. Finally, around time 14353 the competitor thread's quantum expires and the decoder thread gets to run. It soon releases the lock and is preempted by the kernel mixer thread, which runs briefly and then sleeps again, allowing the decoder thread to continue.

Since the decoder thread and the competitor thread are both at priority 9, they preempt each other often, offering many opportunities for the inversion to occur. In fact, it happened three times during our 1-minute test. The Windows NT performance group worked around this inversion by increasing threads' priorities when they grab the lock that is shared with the kernel mixer thread—this is a one-shot implementation of the priority ceiling protocol. We did not use this workaround during our experiments because we wanted to show that CPU Reservations permit an alternative workaround to the priority inversion: when we give a fine-grained reservation to the decoder thread (say 1ms/16ms, as in experiment 8), this bounds the length of the inversion to 16ms—not long enough to be harmful. This can be seen in Figure 5-3: the kernel mixer thread misses an execution slot many times, but it never misses more than one slot. This is consistent with a priority inversion that can easily exceed 10ms but will never reach 20ms. We believe that the fine-grained CPU Reservation in this experiment that rendered the inversion

harmless also triggered the inversion much more often by increasing the number of preemptions (and hence, the probability of a preemption while the shared mutex was held). We do not have a good understanding as to why the priority inversion did not cause starvations in experiment 7; perhaps the were few enough preemptions caused by a priority 10 competitor (as opposed to priority 9) that the inversion just did not manifest itself.

Another Media Player bug that we observed was a circular wait among Media Player threads. Figure 5-5 shows this occurring: around 56300-56400 milliseconds into the run the Media Player user interface, multimedia timer, and decoder threads all block, each waiting for one of the other threads to wake it up. About two seconds later the kernel mixer runs out of data and begins to continuously starve. The deadlock is broken 4 or 5 seconds later when a timed wait expires, and things return to normal for a while. We never saw this deadlock occur on an unloaded system, but the presence of a competitor thread changed the sequence of events, exposing the bug.

These two bugs perfectly illustrate the difficulty of writing correct programs in the presence of many cooperating and synchronizing threads at different priorities.

5.5 Investigative Methods

Rather than using the Media Player source code, we took a reverse-engineering approach to understanding how it works, using the perf kernel. This would have been a bad idea if we had wanted to understand its algorithms. However, we were interested in its dynamic timing behavior—something that is readily observable by watching when threads execute, but which would have been difficult to discern from the source code. In particular, when things went wrong and there were priority inversions and deadlocks, looking through the perf kernel dumps lead us directly to the problems rather than forcing us to infer what had happened from secondary clues.

5.6 Audio Player Conclusions

Without CPU Reservations, the Windows Media Player works only when the priority-based timesharing scheduler happens to give its threads enough CPU time to meet their requirements. We believe that CPU reservations offer a better solution than relying on threads in an open system to spend little time running at priorities higher than the default priority. By having applications (or software running on their behalf) explicitly make their requirements known to the system, a deterministic schedule can be constructed to meet their needs.

6. Related Work

While significant effort has been put into adding real-time extensions to commodity operating systems, we are not aware of any comprehensive study of commonly used applications requiring real-time execution on such systems. We believe that such application studies will contribute to understanding both the strengths and the weaknesses of particular real-time solutions when applied to commodity operating systems, providing much needed input to their designers and engineers.

Predictable thread scheduling does not ensure predictable application execution if portions of the system not under control of the scheduler can introduce long latencies; [1] and [3] study the sources of and quantify the degree of this potential problem.

7. Further Research

Our studies are one step in understanding the benefits to applications of using real-time schedulers on general-purpose operating systems. Soft modems and digital audio are ideal applications for evaluating real-time system abstractions due to their precise timing requirements.

We are interested in investigating the effectiveness of using Rialto/NT to increase the predictability of software DVD movie playback. Unlike audio playback and software modems, this application consumes a substantial fraction of modern CPUs (unless there is hardware acceleration) and so is likely to cause overload and contention.

Finally, this research could be extended to the newly proposed software-based Digital Subscriber Line (soft DSL) [9]. While CPU requirements for soft DSL are much higher, they possess some of the same real-time characteristics as soft modems, making them ideal candidates for understanding the benefits and limitations of real-time schedulers.

8. Contributions

In summary, our contributions have been:

- To show that a software modem driver that was designed to run CPU-intensive signal processing in interrupt mode can be modified to run in a DPC, in a thread scheduled by Windows 2000, and in a thread scheduled by Rialto/NT. The versions of the driver that run in thread context are less detrimental to the predictability of other activities running in the system than the original version and the DPC version. Furthermore, this benefit was achieved without significant loss of modem throughput.
- To study the sensitivity of soft modem performance to the amount and period of CPU reservations. This study is useful not only to understand the performance characteristics of a software modem, but also as a demonstration of an analysis that should be performed for other real-time applications. This analysis is needed in order to determine (1) the smallest amount of reserved CPU time and (2) the coarsest period at which that amount of time must be reserved that allows the application to generate its full value. Reserving a larger amount than this wastes resources, and reserving at a smaller period will increase the number of preemptions in the system, increasing context switch overhead and reducing cache effec-

tiveness, and therefore reducing the amount of useful work that can be performed.

- To show that the predictability of a complex GUI-based application, the Windows Media Player, can be increased by assigning CPU reservations to its most time-sensitive threads. In the absence of CPU reservations, Windows Media Player only works when other applications do not happen to spend much time running above priority 8 (the default thread priority in Windows 2000).
- To show that CPU reservations can be applied to a complex, multithreaded application without modifying the application (and in fact, without even restarting it). Rather than understanding this large application (that uses several layers of middleware and device drivers) at the source level, we gathered enough information to apply CPU reservations by observing its run-time behavior using an instrumented Windows 2000 kernel.

9. Conclusions

We have performed two case studies, each of which involved applying CPU reservations to an existing real-time application under Windows 2000. The first application is a low-level system service that provides signal-processing for a software modem. The second is a high-level GUI-based application that is the default player for streaming audio and video in Windows 2000 (however, the most hard real-time task involved in playing audio, the kernel mixer, is again a low-level system service).

We found two main benefits to studying real applications. First, there was no need to ground-truth a simulation or second-guess application characteristics: we used the same complex applications that currently run on millions of consumer systems. Second, we were able to use real performance metrics like modem throughput and perceived audio quality rather than synthetic metrics such as number of missed deadlines or mean lateness that, in a soft real-time environment, may not be strongly correlated with the value generated by applications.

In conclusion, we believe that in an open system, it is important to be able to specify application requirements in absolute units (such as amounts and periods) rather than in relative units such as priorities or shares, for which optimal values cannot be calculated without global system knowledge. The relative ease with which we applied CPU reservations, which allow requirements to be specified in absolute units, to these very different applications speaks well for Rialto/NT and for the overall applicability of CPU reservations to periodic tasks in general-purpose operating systems.

Acknowledgments

The authors would like to thank Patricia Jones for her helpful comments on drafts of this paper.

References

[1] Erik Cota-Robles and James P. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, pages 159-172, February 1999.

[2] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pages 198-211, October 1997.

[3] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[4] Michael B. Jones and John Regehr. CPU Reservations and Time Constraints: Implementation Experience on Windows NT. Michael B. Jones and John Regehr. In *Proceedings of the Third USENIX Windows NT Symposium*, Seattle, WA, pages 93-102, July 1999.

[5] Michael B. Jones and Stefan Saroiu. Predictability Requirements of a Soft Modem. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.

[6] *WDM Audio Drivers for Windows 2000*. Microsoft Corporation, 1999. http://www.microsoft.com/hwdev/devdes/wdmaudio.htm.

[7] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerald Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*. Lancaster, U.K., November 1993.

[8] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, 2000.

[9] Mike Tramontano. *The DSL Market is Going Soft*. Inter@ctive Week Online, Ziff Davis, July 17, 2000. http://www.zdnet.com/intweek/stories/news/0,4164,2604854,00.html.