The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

A Soft Real-time Scheduling Server on the Windows NT

Chih-han Lin, Hao-hua Chu, Klara Nahrstedt
University of Illinois at Urbana Champaign

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Soft Real-time Scheduling Server on the Windows NT

Chih-han Lin, Hao-hua Chu, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana Champaign
clin2, h-chu3, klara@cs.uiuc.edu

Abstract

We present the design and implementation of a soft real time CPU server for the time-sensitive multimedia applications in the Windows NT environment. The server is a *user-level* daemon process from which multimedia applications can request and acquire periodic processing time in the well-known form of (*processing time* per *period*). Our server is based on a careful manipulation of the real time(RT) priority class, and it does not require any modifications to the kernel. It provides (1) the rate monotonic scheduling algorithm, (2) support for multiple processors (SMP model), (3) limited overrun protection among real-time(RT) processes, (4) fair allocation between the RT and time sharing (TS) processes so that TS processes are not starved for processing time, (5) accessibility by a normal user privilege, and (6) an efficient implementation. We have implemented the CPU scheduling server on top of the Windows NT 4.0 operating system with dual Pentium processors, and we have shown through experiments that our CPU scheduling server provides good soft real time support for the multimedia applications.

1. Introduction

Continuous media processing, such as video/audio compression/decompression, and 3-D rendering and animation, are becoming widely-used applications on computers nowadays. To preserve their temporal behavior, multimedia applications require that the underlying systems provide sufficient and periodic processing time and enforce quality guarantees to the users (e.g. a fixed video playback rate). However, in the Windows NT multi-process and time-sharing environment, these multimedia applications do not perform well when they are scheduled concurrently with the traditional TS applications such as text editors, compilers, web browsers, or computation-intensive jobs. Oftentimes, the problem lies in untimely scheduling of the processes rather than insufficient processor capacity. This paper addresses this problem and presents a user-level middleware solution on top of the Windows NT operating system with multiple processors.

There have been several research results that address the issues of accommodating scheduling of soft RT processes in general purpose operating system environment. They are the Constant Utilization Servers[2], the Processor Reserve of the RT Mach[8,6], the Hierarchical CPU scheduler[4], the User-level Real Time Scheduler[7], the Real Time Upcall[3], the Rate-Controlled Scheduling[11], the Soft RT scheduling Server[9], the Rialto operating system[5], the Nemesis[1], and the SMART system[10]. All except [10] are based on the general concepts of *reservation, resource allocation, and scheduling.* The RT process first sends a reservation request, which specifies its resource demand, e.g., RT Mach convention of (requested CPU usage time, period), to the resource manager. Then the resource manager performs an admission control to determine if there is enough available resource to allocate for this request. If the admission control test succeeds, the RT process is scheduled according to the reservation contract.

Our scheduling mechanism is based on the *user-level RT scheduler(URSched)* proposed by Kamada[7] in UNIX. The URSched mechanism is based on the POSIX.4 fixed priority extension and its priority scheduling rule. The user-level scheduler is implemented on top of the kernel scheduler, and it runs at the highest possible fixed-priority. The RT process waits its turn at the lowest possible priority (called the waiting priority), and the active RT processes run at the 2nd highest fixed priority. The user-level scheduler wakes up periodically to dispatch the RT processes by moving them between the waiting and the running priority; during the other time, it just sleeps. When the scheduler sleeps, the RT process executes at the running priority. When no RT processes are dispatched, the TS processes with dynamic priorities are scheduled by the underlying kernel scheduler. This approach has shown to have many desirable properties:

- It requires no modification to the kernels. The RT scheduler is implemented as a user-level process.
- It has low computation overhead.
- It provides the flexibility to implement any scheduling algorithms in the user-level scheduler, e.g., rate monotonic, earliest deadline first, or a hierarchical scheduler.

The above discussed scheduling mechanism was used and further expanded by additional mechanisms, algorithms, and policies in our QoS-aware resource management middleware [9], called QualMan. In this context, the middleware is understood as a system software between operating system and applications that provides access to extended and flexible OS services, e.g. real time support, to applications without any modifications of the operating systems. Our middleware provides the following services :

- Rate Monotonic Scheduling algorithm.
- Overrun Protection among RT processes.
- Fair allocation between the RT and TS processes.
- Access to system services with normal user privileges.

Based on lessons learned from the soft RT scheduling server in the UNIX environment, we design, implement, and test the soft real-time scheduling server in the Windows NT environment. In addition, we provide support for scheduling of RT and TS processes on multiple processors. The paper is organized as follows: section 2 explains the scheduling server architecture; section 3 describes the implementation on the Windows NT platform, and discusses the differences between UNIX and the Windows NT operating system; section 4 shows the experimental results; section 5 presents the concluding remarks.

2. Server Architecture and Design

Our server architecture contains three major components—the broker, the dispatcher, and the dispatch tables as shown in Figure 1. A RT Client is an external component representing an application which requests the scheduling services from the scheduling server.

Before we describe each component in detail, we discuss the priority levels in the Windows NT system because they play an important role in our architecture and design. Each NT process or its main thread has a scheduling priority. Each thread's priority is determined by the priority class of its process and the priority level of the thread within the priority class of its process. Note that our scheduler is a *process* scheduler and it does not schedule the various threads in each RT process. There are four possible priority classes for a process:

1	IDLE_PRIORITY_CLASS
2	NORMAL_PRIORITY_CLASS
3	HIGH_PRIORITY_CLASS
4	REALTIME_PRIORITY_CLASS

There are seven possible priority levels within each priority class:

1	THREAD_PRIORITY_IDLE
2	THREAD_PRIORITY_LOWEST
3	THREAD_PRIORITY_BELOW_NORMAL
4	THREAD_PRIORITY_NORMAL
5	THREAD_PRIORITY_ABOVE_NORMAL
6	THREAD_PRIORITY_HIGHEST
7	THREAD_PRIORITY_TIME_CRITICAL

2.1 Client RT Process

A RT process can reserve a certain amount of CPU time from the real-time scheduling server. At a later time, it can also free or modify a reservation through an application program interface (API) defined in Table 1.

The client's reservation request contains the process ID and its resource specification in the form: period=P(ms), and CPU usage in percentage=U. The amount of CPU time per period, denoted E, is computed as $E = U*P$. For example, if a RT process requests a reservation for CPU usage = 30%, period = 100ms, and the request is accepted by the scheduling server, then the scheduling server will guarantee that the RT process is scheduled for 30ms of CPU time every 100ms, given that the RT process is runnable.

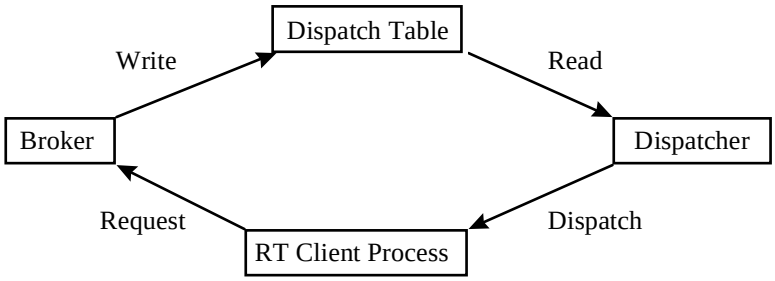

Figure 1: The Soft RT Server Architecture

Table 1: Application program interface

int Cpu::reserve(int *pid*, double *util*, int *period*)	Reserve CPU time corresponding to *util* over the time *period* for the process *pid*. Return 1/0 for resource admission success/failure.
int Cpu::freeReserve(int *pid*)	Free the reservation hold by process *pid*. Return 1/0 for success/failure.
double Cpu::getAvailResource()	Return the amount of available CPU resource in the system.
int Cpu::modifyReserve(int *pid*, double *newUtil*, int *newPeriod*)	Modify the reservation held by process *pid*. Return 1/0 for resource admission success/failure.

The users start the RT process at the NORMAL_PRIORITY_CLASS which is like any other Time Sharing (TS) processes. When the RT process calls the CPU reserve() API, it opens an inter-process communication (IPC) with the broker through which the reserve parameters and the acceptance result are exchanged. To address the security problem which one process can modify or free another process's resource reservation through the API, the broker performs an ownership check to make sure that the calling process can be permitted to change the resource reservation of the specified RT process. Then the broker and the dispatcher schedule the RT process by manipulating its priority, the mechanism is described below.

2.2 Broker and Dispatch Table

The broker receives requests from the client RT processes. It performs an admission control test for the non-preemptive rate monotonic scheduling algorithm [12] to determine whether the new client RT process can be scheduled. Note that all equations must be satisfied for the admission test to succeed. Given a total of m periodic RT processes and a single processor system (N=1), let the RT processes be sorted according to sizes of their periods. Let e_i and p_i be the execution time and the period of the i-th client RT process, e_m and p_m be the execution time and period of the RT process with the smallest period, and r be the overall percentage of processor capacity allocated to the RT processes.

$$\sum_{i=1}^{m} \frac{e_i}{p_i} \le r\,N \quad \textbf{(eq. 1)}$$

$$p_m \ge e_m + max_{(1 \le i < m)} e_i \quad \textbf{(eq. 2)}$$

$$p_i \ge e_i + max_{(1 \le j \le m, j \ne i)} e_j + \sum_{j=i+1}^{m} e_j F(p_i - e_j, p_j) \quad \textbf{(eq. 3)}$$

$$where F(x, y) = ceil(\frac{x}{y}) + 1$$

Given an N processors system, the broker needs to decide how to place the multiple RT processes into the multiple processors. The broker maintains a set of RT processes that are admitted and are assigned to run on each of the processors. The set of processes for each processor must satisfy both **eq. 2** and **3**. When a new RT process request arrives, the broker tries to place the new RT process in each of the processors *(1..N)* by performing the above one-processor admission trial on the processor. During the one-processor admission trial, the new process is inserted into the existing set of processes corresponding to that processor, and **eq. 2** and **3** are checked. If the trial succeeds, the broker will admit the new RT process which is assigned to that processor. If the trial fails, the broker will try to place the new RT process in the next processor. If the broker cannot find any processor that can accommo-

date the request, the RT process is rejected and not schedulable.

If it is schedulable, the broker will put the RT process into the waiting RT process pool by changing its priority to the waiting priority at IDLE_PRIORITY_CLASS level.

The broker is a daemon process running at a normal priority (The server's priority class is NORMAL_PRIORITY_CLASS and the priority of its primary thread is THREAD_PRIORITY_NORMAL). It can be started at the system boot time. It will wake up when a new client RT process request arrives. The broker needs to be run with the privilege of LocalSystem (equivalent to the root privilege in UNIX) so that it can start a RT process. The broker process does not perform the actual dispatching of the RT processes and does not enforce reservations. However, it needs to start (at the startup time) the dispatching process. The reason for separation between the broker process and the dispatcher process is that the admission and schedulability test in the broker may have variable computation time, hence it may affect the timing of dispatching. The other reason is that the admission and schedulability tests do not need to be done in real time. As a result, the broker runs at a normal priority and the dispatcher at a higher RT priority.

The broker computes its schedule for the RT processes in its dispatch table using the non-preemptive rate monotonic algorithm. The dispatch table is a shared memory object where the broker writes the computed schedule into it and the dispatcher reads from it. The dispatch table contains a repeatable time frame of slots, each slot corresponds to a time slice of a processor. Each slot can be assigned to a RT process PID, or is free which means yielding the control to the NT kernel scheduler to schedule TS processes. Note that Windows NT allows the system to have multiple processors in a symmetric multiprocessing model (SMP). Given N processors, we can run N processes concurrently. Therefore the dispatch table has N columns of repeatable time slots, where the i-th column corresponds to the schedule on the i-th processor. We show a sample dispatch table for a dual processors system in Table 2. Note that it is possible that the broker may make a massive change to the dispatch table when accepting a new RT process, while the dispatcher is dispatching time slots located at middle of a time frame. This massive change may cause undesirable shifts in the rate monotonic schedule and may disrupt the guaranteed time slots assigned to some RT processes. As a result, the dispatcher will keep a separate private copy of the dispatch table which it uses for dispatching. This private copy of the dispatch table is only updated with the broker when the dispatcher reaches the end of its time frame.

Table 2: A sample dispatch table for a dual processors system.

Slot Number	Time	Process PID	Process PID
0	0-20ms	100	102
1	20-40ms	101	103
2	40-60ms	100	102
3	60-80ms	free	free
4	80-100ms	100	102
5	100-120ms	free	103
6	120-140ms	100	102
7	140-160ms	free	free

The repeatable frame in Table 2 has a length of 160ms, and it contains 8 time slots of 20ms each. The sample dispatch table is a result of a rate monotonic schedule, where process 100 is assigned to run on processor #1 and according to the contract (period=40ms, execution time=20ms), process 101 is assigned to run on processor #1 and according to the contract (period=160ms, execution time=20ms), process 102 is assigned to run on processor #2 and according to the contract (period=40ms, execution time=20ms), and processor 103 is assigned to run on processor #2 and according to the contract (period=80ms, execution time=20ms). There are 4 free slots where TS processes can run. The minimum number of free slots is maintained by the broker to provide a fair share of the CPU time to the TS processes. In the above dispatch table, a total of 31.25% (100ms out of possible 320ms) of processing capacity is guaranteed to the TS processes. The site administrator can adjust this TS allocation value, which is (1-r) in the admission control equations, to be what is considered as a fair allocation between the TS and RT processes. For example, if the computer is used heavily for RT applications, the TS allocation value can be set to a small percentage number.

2.3 Dispatcher

The dispatcher is a periodic process running at the highest possible fixed priority with (REALTIME_PRIORITY_CLASS +THREAD_PRIORITY_TIME_CRITICAL). The

dispatcher maps the dispatch table into its address space, and its job is to dispatch the RT process recorded at the time slot for all the processors. The dispatcher contains the next slot number. At the beginning of each dispatch slot, a periodic RT timer signals the dispatcher to schedule the next RT process. The length of time to switch from the end of one time slot to the beginning of the next time slot is called the dispatch latency. The dispatch latency is our scheduling overhead and it should be kept at a minimal value.

Consider the sample dispatch table in Table 2 at 20 ms when the next slot number is 1. The following steps are taken by the dispatcher:

1. The periodic timer wakes up the dispatcher process at 20ms. The dispatcher preempts RT process 100(or 102) if it is running on processor 1(or 2).

2. The dispatcher sets the RT processes 100 and 102 to the waiting priority (IDLE_PRIORITY_CLASS) using the system call SetPriorityClass().

3. The dispatcher promotes the RT processes 101 and 103 to the running priority (REALTIME_PRIORITY_CLASS). It binds the RT processes 101/102 to processors #1/#2 using the system call SetProcessAffinityMask().

4. The dispatcher puts itself to sleep. As a result, the RT processes 101/102 are scheduled on processors #1/#2 for 20ms until the timer wakes up the dispatcher again.

We have encountered a problem in the Windows NT that the periodic timer may fail to wake up the dispatcher when the RT process overruns its assigned slot. Overrun is defined as an additional time to the reserved processing time of an admitted RT process. Take the example of process 101 in Table 2 and it has reserved 20ms out of every 160ms. It would be overrunning when it uses more than 20 ms of processing time (say 30ms) to complete one iteration run. When an overrun occurs, the invocation of the dispatcher will be delayed till the RT process finishes its overrun due to the timer problem. This means that one process overrun can delay the dispatching of the process in the next time slot. Hence, until we resolve the preemption timer problem in Windows NT, we assume well-behaved RT processes. In the meantime, the dispatcher takes two actions to control the overrun. The first one is that the dispatcher will monitor for any *misbehaving* RT processes that are overrunning on a regular basis and it will remove them. We currently define a misbehaving process as one that is overrunning for more than 20% of its reservation and for more than 3 iterations during the most recent 10 iterations. Take the example of process 101 in Table 2. It would be misbehaving if during the 10 most recent iteration runs, it uses more than 24ms (which is 20% more than the 20ms reservation) of processing time for more than 3 times. The parameters that define a misbehaving process can be set by the system administration to be either more strict or lax. The second action is that the dispatcher will use some of the TS allocation, by temporarily assigning the free slots to the RT process whose time slots are taken by overrunning RT processes.

3. Implementation

We have implemented our server architecture on a HP Vectra Xu system which has two Intel Pentium 200 processors and 96M of memory. It runs Windows NT 4.0 operating system. The dispatch latency consists of 4 SetPriorityClass() system calls, 2 SetProcessAffinityMask() system calls, and 3 context switches. The average dispatch latency, over 10,000 runs, is measured as 0.64 ms. The time slot is set to be 20ms, and the overhead comes to be 3.2%.

4. Experimental Result

The experiment consists of the following load of applications running concurrently:

- The MPEG player, written by MSSG (MPEG Software Simulation Group), plays a 320x240 MPEG-1 file “twister.mpg” that contains a clip of the movie Twister at 20 frames per second (FPS).
- The same MPEG player plays a second 320x240 MPEG-1 file “lecture.mpg” that contains a speaker giving a lecture, at 20 frames per second.
- The Microsoft Visual C++ Compiler compiles the MSSG MPEG player.
- Four compute programs calculate the sin and cos tables using the infinite series formula.

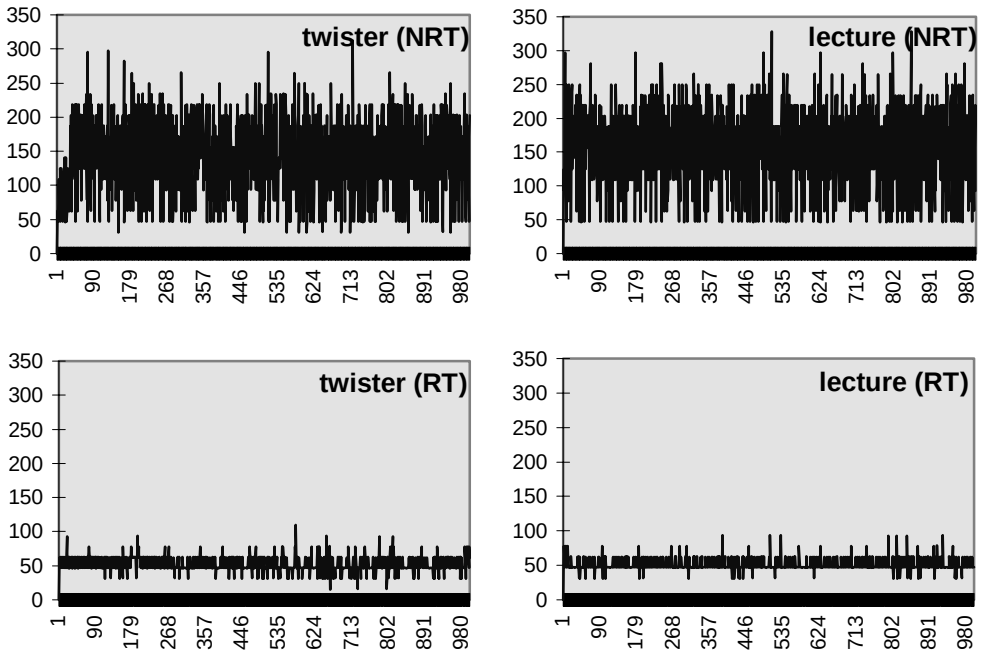

Figure 2: The inter-frame time for the MPEG player that plays the "twister.mpg" and "lecture.mpg" files at 20 frames per second. The y axis measures the inter-frame time in ms, and x axis shows the frame numbers. The top two graphs show the result for the Windows NT kernel scheduler, and the bottom two graphs show the result for our scheduling server with processor reservation for the "twister.mpg" at (100%, 50ms) and "lecture.mpg" at (80%, 50ms).

The graphs in Figure 2 show the measurement of inter-frame time on the MPEG player under the above specified load. The top two graphs show the result for the "twister.mpg" (upper left) and "lecture.mpg" (upper right) under the Windows NT scheduler without our scheduling server. The bottom two graphs show the result for the 20 FPS "twister.mpg" (lower left) with 100% processor reserve every 50 ms, and the 20 FPS "lecture.mpg" (lower right) with 80% processor reserve every 50ms. The "twister.mpg" playback requires more processor time because it contains lots of action with rapid moving background, whereas the "lecture.mpg" playback contains slow moving action with almost no changes in background. Using the Windows NT scheduler, jitter over 200ms (equivalent to 4 frames time) occurs frequently for both MPEG video. Using our scheduling server, jitter over 200ms does not occur at all.

5. Conclusion

Our experiments validate the design and implementation of the soft real-time scheduling server for continuous media processing. Our contribution is that under the control of our scheduling server within the Windows NT environment, (1) RT processes obtain a desired amount of CPU time to satisfy soft real-time requirements; (2) RT processes are monitored and protected against overruns of other RT processes; (3) RT processes have access to timing QoS guarantees and systems services with normal user privileges; and (4) TS processes get a minimum amount of CPU time and do not starve.

References

1. Richard Black, Paul Barham, Austin Donnelly, Neil Stratford. Protocol Implementation in a Vertically Structured Operating System. IEEE LCN, Nov. 1997.

2. Z. Deng, J.W.-S. Liu, J. Sun. Dynamic Scheduling of Hard Real-Time Applications in Open System Environment. *Technical Report* No. UIUCDCS-R-96-1981, Department of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1996.

3. R. Gopalakrishnan. Efficient Quality of Service Support Within Endsystems for High Speed Multimedia Networking. PhD Thesis, Washington University. Dec. 96.

4. Pawan Goyal, Xingang Guo, Harrick Vin. " A Hierarchical CPU Scheduler for Multimedia Operating System". *The proceedings of Second Usenix Symposium on Operating System Design and Implementation*, Seattle WA, Oct 1996.

5. Michael B. Jones, Daniela Rosu, Marcel-Catalin Rosu. "CPU Reservations and Time Constraints: Efficient, predictable Scheduling of Independent Activities". *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, St. Malo, France, Oct. 1997.

6. Chen Lee, Ragunathan Rajkumar, Cliff Mercer. Experience with Processor Reservation and Dynamic QoS in Real-Time Mach. *Multimedia Japan*, 1996.

7. Jun Kamada, Masanobu Yuhara, Etsuo Ono. "User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler". *Multimedia Japan*, March 1996.

8. Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications". *IEEE International Conference on Multimedia Computing and Systems*. May 1994.

9. Klara Nahrstedt, Hao-hua Chu, Srinivas Narayan. QoS-Aware Resource Management for Distributed Multimedia Application. *Technical Report* No. UI-UCDCS-R-97-2030, Department of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1996.

10. Jason Nieh, Monica Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, Oct. 1997.

11. David K.Y. Yau and Simon S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia Applications. *ACM Multimedia Conference*, Boston, MA, Nov. 1996.

12. R. Nagarajan and C. Vogt. Guaranteed-Performance Transport of Multimedia Traffic over the Token Ring. *Technical Report 43.9201*, IBM European Networking Center, IBM Heidelberg, Germany, 1992.