The following paper was originally published in the
Proceedings of the USENIX Windows NT Workshop
Seattle, Washington, August 1997

Dreams in a Nutshell

Steven Sommer
Microsoft Research Institute and Department of Computing,
School of Mathematics, Physics, Computing and Electronics,
Macquarie University, Australia

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

Dreams in a Nutshell

Steven Sommer
Microsoft Research Institute and Department of Computing,
School of Mathematics, Physics, Computing and Electronics,
Macquarie University,
NSW 2109, Australia

steve@mpce.mq.edu.au

Abstract

The Dreams extensions have been developed in order to support distributed real-time applications within the conventional operating system paradigm. To demonstrate the viability of the extensions, they have been implemented within Windows NT. This paper introduces the important components of the Dreams extensions, provides an overview of the implementation, and highlights some of the experiences gained from the implementation.

1. Introduction

The aim of the Dreams (Distributed Real-Time Extensions with Application to Multimedia Systems) project is to provide a complete set of extensions for conventional operating systems, so that they may support real-time and distributed real-time processes within the conventional operating system paradigm. To demonstrate the viability of our extensions we have partially implemented our extensions within the Windows NT™ [2] operating system. This paper introduces and motivates the inclusion of each of the key components of the extensions, provides a brief overview of their implementation, and highlights some of the experiences gained from the implementation.

A fundamental component of the conventional operating system paradigm is the ability to run independent applications simultaneously while protecting these applications from interfering with one another. This is quite different from the paradigm of real-time systems, where all tasks work together with a common aim. The Dreams extensions allow real-time applications to be protected from one another.

Unlike traditional real-time systems [1, 4, 6], conventional operating systems do not have *a priori* knowledge of the arrival times and behaviors of real-time tasks; the schedulability test and scheduling algorithm must be performed on-line. Conventional operating system also allow: interrupts which run at a higher priority than both non-real-time and real-time applications; subsystems which execute system calls but which are otherwise indistinguishable from user applications; and dynamic distribution of applications.

To maximize the ease of adoption of these extensions, they have been developed to have a minimal impact on the code and conventional behavior of the operating system and on the programming model used to develop real-time applications. The Dreams extensions have been designed to be independent of any particular operating system.

In [8], we identified a new real-time process abstraction, called the *transient periodic process*, which is better suited to real-time applications running on a conventional operating system. Transient periodic processes have two distinguishing features.

- They act as periodic processes while running but, unlike traditional periodic processes, an entire process may start and complete at any time.
- The starting time of the first invocation of the transient periodic process is not constrained by the process.

An overview of the Dreams model, the advantages of the transient periodic process abstraction, and a general comparison with other similar models, can also be found in [8].

2. Protection

There are two major areas in which a conventional operating system must be enhanced to allow it to protect real-time applications from one another.

Firstly, the operating system must offer a new form of protection, called *temporal protection*. This encompasses the requirement that the timing behavior of one task should not be able to affect the ability of another, independent task to meet its deadline. To achieve this, the scheduling method needs to be altered, the timing behavior of each task must be monitored and enforced, and a method for effectively dealing with overrun tasks must be developed. Details of the Dreams approach to temporal protection can be found in [9].

Secondly, the operating system's support for resource sharing must be extended to be consistent with the requirements of real-time scheduling. The areas of resource sharing that must be considered are the concurrency support primitives (for example, a mutex) and the subsystem call mechanism. There are two predominant alterations required in this area. The first is to ensure that if the real-time task that should be running is blocked on another task, then the other task is immediately executed. In [7] we detail our approach for achieving this, that is, an extended form of priority inheritance [5], and argue for its inclusion within all conventional operating systems. We have also developed an extension to priority inheritance, called quantum inheritance, which also improves the conventional functioning of the operating system. The second resource sharing alteration consists of ensuring that a real-time task's blocking time is bounded and that the blocking can be effectively modeled within the schedulability test.

3. Modeling the Operating System

A fundamental element of the Dreams model is the guarantee that an application that does not exceed its reserved time will always meet its deadlines. The Dreams model ensures that the system as a whole has sufficient capacity to satisfy all of the active real-time applications by using an admission mechanism and by constraining and modeling particular parts of the operating system.

Many aspects of the system are outside of the control of the Dreams scheduler. The schedulability test must include the interference of the system as well as the system's own resource requirements. Elements of the system which impact real-time threads include hardware and software interrupts, caching, deferred procedure calls, parts of the conventional system, and the Dreams scheduler itself. Our schedulability test also models the impact of interrupts on effective enforcement and preemption.

We have formally extended Liu and Layland's earliest deadline first (EDF) schedulability test [3] to include clocked, interspaced sporadic, and bursty sporadic real-time interrupts. We have further extended the tests to include priority inheritance and critical sections, both with and without enforcement mechanisms. Our schedulability results can be found in [10].

Some modifications to the conventional operating system are necessary to bound the system's behavior; for example, interrupts and deferred procedure calls must have a maximum duration and frequency, or they must tolerate occasional suspension. Particular system calls must have a bounded timing behavior so that they can be used by real-time tasks. Our work in this area is not yet complete.

4. Distribution

The Dreams model allows a transient periodic process to be distributed to networked machines when there are insufficient resources to run the process on the local machine. The distribution component is responsible for the selection of the networked machines and the distribution of the transient periodic process to those machines. The distribution component of Dreams has not yet been integrated into our implementation.

Most of the difficulties involved in supporting distributed real-time processes are also key concerns in the research areas of real-time communication, process distribution, and load balancing. Although we have briefly examined the important issues required for supporting distributed real-time processes, we have chosen to concentrate primarily on those issues which should be addressed differently in the Dreams context.

We have developed a distributed placement algorithm for selecting the remote machines on which the transient periodic processes should execute. The

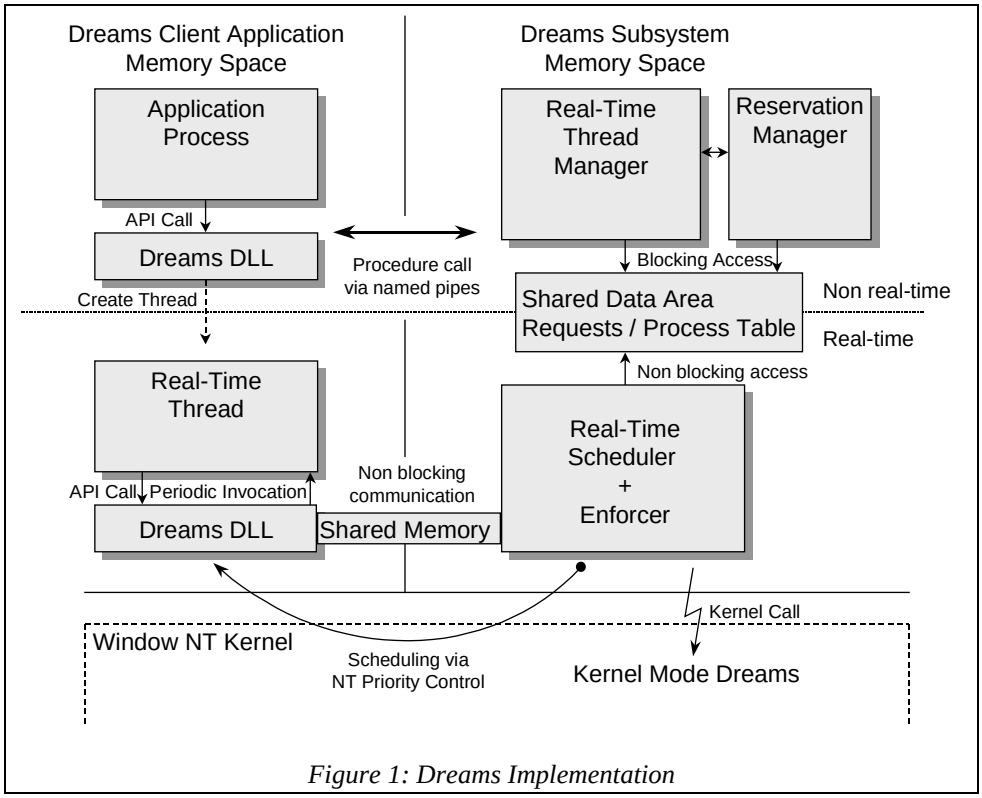

Figure 1: Dreams Implementation

algorithm was developed for optimal performance within the Dreams framework. Currently, we have only analyzed the algorithm within a simulation environment.

The inclusion of distribution within the Dreams model has had a significant influence over the design of many of the other components of the model. For example, one of the most compelling reasons for strictly preventing system overload comes from distribution. Distribution requires that network card interrupts be serviced in a real-time manner; sporadic interrupts must be enabled and modeled within the schedulability test. The system must be able to distribute each real-time entity separately; the real-time entity must, therefore, be implemented as a thread or process, not as a block of code. One of the requirements for the transient periodic process abstraction was that it allowed distribution to be performed in a manner consistent with the functionality of an operating system containing the real-time Dreams extensions.

5. Implementation Design

We have partially implemented the Dreams extensions within version 3.51 of the Windows NT operating system. A real-time[†] application is implemented as a Win32™ application with additional real-time facilities. Most of the new facilities are provided by a user-mode process called the *Dreams subsystem*. To use these facilities the application need only include an additional library and header file.

The transient periodic process of our model has been implemented within Windows NT as a Win32 thread. An application creates a real-time thread in the same manner as creating a conventional thread but with additional parameters specifying the real-time properties of the thread. These include its start time, reserved time, deadline, period, and additional flags.

Figure 1 provides an overview of the implementation. The following paragraphs highlight the important

[†] Our use of the term *real-time* is unrelated to the "real-time" priority class of Windows NT.

components in the figure, by tracing though the creation and initial execution of a real-time thread.

When an application makes an API (Application Programming Interface) call to create a real-time thread, the Dreams DLL (Dynamic Link Library) sends the request to the real-time thread manager, which runs as part of the Dreams subsystem. If the new task set passes the schedulability test performed by the reservation manager, the real-time thread manager approves the request. If not, the real-time thread manager may attempt to distribute the task by communicating with other thread managers. If the task can be run locally, the DLL creates a new thread and an associated shared memory segment. The new thread performs its initialization and then waits for its first invocation. The DLL then passes handles and control of the thread to the subsystem. The real-time thread manager performs the setup required for the thread in the subsystem and places a token for the thread in the real-time scheduler's shared data area. The scheduler takes control of the new real-time thread when it has spare time.

The scheduler executes at the highest Windows NT priority. It allocates CPU time to a real-time thread by making it the second highest priority thread. The scheduler and real-time thread communicate using the shared memory area. The periodicity of the real-time thread is implemented within the Dreams DLL by having a single Win32 thread call the specified entrance point of the real-time thread at the beginning of each invocation. Each invocation completes by making an API call which signals the completion to the scheduler.

6. Windows NT Implementation Experiences

In this section, we highlight some of the experiences gained from the implementation of the Dreams extensions within Windows NT.

An important issue in the design of the implementation was the choice of placing the extensions in a subsystem, or placing them in the Windows NT executive and kernel. We decided to place the extensions in a subsystem and to move components into the kernel as it became necessary. Having the extensions in a subsystem was consistent with our goals of minimizing the effect on the conventional functioning of the operating system and minimizing the alterations to the code of the existing operating system. Adopting this approach allowed the extensions to be developed, modified, tested, debugged, displayed, and understood, far more easily than if they were placed in the kernel.

The Dreams scheduler was the one component that could suffer from being placed in a subsystem. Placing the Dreams scheduler in the subsystem introduces a performance overhead due to additional context switches, and has the potential for duplicating kernel scheduling information. The overhead of invoking the Dreams scheduler is equivalent to a single subsystem call. This additional overhead is of no consequence to the other components of the model. In our current implementation, no data is shared between the two schedulers.

The one problem that we found in implementing the functionality of the scheduler in the subsystem was that the Dreams scheduler was occasionally being invoked late. This was seen in two forms. Firstly, the scheduler could be invoked a full millisecond late. It could detect this at the time it was invoked. Secondly, the scheduler could be invoked near the end of a one millisecond interval instead of at a one millisecond boundary. It could detect this by noting that the time had changed during the interval within which it had been executing. The scheduler was invoked late for a number of reasons. The scheduler used a Win32 multimedia timer to control the time that it was next invoked. Often the timer and the scheduler were delayed by the execution of system functions; these should not have taken a full millisecond. Unfortunately, the time at which the timer should fire is specified as a number of milliseconds from the current time. The current time could change while the call to set the timer was being made, causing the timer to fire one millisecond late. Finally, the time set by the clock interrupt, the time used by the Win32 timers, and the time obtained from interrogating the hardware clock were all slightly out of phase; this could also cause the Win32 timer to fire one millisecond late. To rectify this problem, we decided to invoke the Dreams scheduler off the clock interrupt in a similar manner to that which occurs at a quantum end.

Placing the Dreams scheduler in the subsystem led us to develop a two tiered scheduling approach that, with the implementation of priority inheritance within the priority scheduler, turned out to be remarkably advantageous. The priority scheduler, with priority inheritance, ensures that the correct thread is

scheduled each time the real-time thread blocks or makes a subsystem call. The Dreams scheduler need only be concerned with implementing the real-time scheduling algorithm. When it decides to schedule a different real-time thread, it simply drops the priority of the real-time thread being preempted and raises the priority of the next thread to be scheduled. The priority inheritance protocol will then transparently and precisely implement the desired scheduling behavior, *even if a different thread was actually executing*: for example, if the executing thread was a non-real-time thread completing a critical section, so as to unblock a subsystem thread which was performing a system call on behalf of the scheduled real-time thread. If, in this example, the real-time scheduler later decides to schedule the preempted real-time thread, then the non-real-time thread that had actually been preempted will automatically be executed to release its critical section and allow the system call to complete. The complexity of this scheduling can be hidden from the Dreams scheduler, the real-time task computational usage tracking, the enforcement mechanism, and the schedulability model.

We chose to implement priority inheritance within the QLPC (Quick Local Procedure Call) mechanism as this mechanism appeared to be the most frequent cause of priority inversion [5]. The implementation consisted of: removing the existing scheme for partially overcoming some of the effects of priority inversion; adding a smaller amount of code to implement priority (and quantum) inheritance; and lowering the priority of the Win32 subsystem servicing threads to the idle priority. Significantly more effort would have been required to implement priority inheritance in a single tiered scheduler in which one of the priority queues was being used to schedule threads in an EDF manner.

The benefits from the implementation of priority inheritance were not limited to the Dreams scheduler. After implementing priority inheritance, we found that a number of very poor scheduling behaviors were removed from the system. As a result of the implementation, an optimization used within Windows NT for particular combinations of priorities was usable for threads of all priorities. This yielded a performance increase for GDI calls belonging to threads at particular priorities. We ran two video display applications at the same priority. On the system containing our modifications, the videos played more smoothly; they appeared to execute at the same time, rather than one after the other. Finally, we found that a lower priority video player no longer interfered with the execution of a higher priority editor.

7. Conclusion

This paper has introduced the key components of the Dreams extensions. The extensions are required to allow a conventional operating system to support competing real-time applications within the conventional operating system paradigm. Most of the extensions described in this paper have been successfully applied in our Windows NT implementation. This paper has also provided an overview of the implementation and has highlighted some of the experiences gained from the implementation.

Acknowledgments

I would like to thank John Potter, Mark Dras, and Yan Han for their assistance with this paper. This work was supported by a Microsoft Research Institute Fellowship and an Australian Postgraduate Research Award.

References

[1] Burns, A. and Wellings, A. *Real-Time Systems and their Programming Languages*, Second Edition, Addison-Wesley, 1996.

[2] Custer, H. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[3] Liu, C.L. and Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM* 20(1):44-61, January 1993.

[4] Ramamritham, K. and Stankovic, J. Scheduling Algorithms and Operating Systems Support for Real-Time Systems, *Proceedings of the IEEE* 82(1):55-67, January 1994.

[5] Sha, L., Rajkumar, R. and Lehoczky, J. Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers* 36 (9):1175-1185, 1990.

[6] Shin, K. and Ramanatham P. Real-Time Computing: A New Discipline of Computer Science and Engineering, *Proceedings of the IEEE* 82(1):6-24, 1994.

[7] Sommer, S. Removing Priority Inversion from an Operating System. In *Proceedings of the Nineteenth Australasian Computer Science Conference*, 131-139, January 1996.

[8] Sommer, S. and Potter, J. Operating System Extensions for Dynamic Real-Time Applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[9] Sommer, S. Temporal Protection in Dreams. In *Proceedings of the Twentieth Australasian Computer Science Conference*, 56-64, February 1997.

[10] Sommer, S. and Potter, J. *Admissibility Tests for Interrupted Earliest Deadline First Scheduling with Priority Inheritance*, Technical Report C/TR97-10, MRI, MPCE, Macquarie University, 1997.
