Real-Time Scheduling Support in Ultrix-4.2 for Multimedia Communication*

Tom Fisher
fisher@tenet.Berkeley.EDU

The Tenet Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720
and
International Computer Science Institute
1947 Center Street
Berkeley, California 94720

Abstract. Advanced workstations are now being equipped to handle various types of multimedia applications such as audio and video. The quality of these images and sounds depends highly on the timely management of multimedia data. In order to provide such management, support for real-time process scheduling must exist. In this paper we describe the fundamental mechanisms implemented in the Ultrix4.2 kernel that provide the real-time scheduling support needed for multimedia applications. Our primary goal is to reduce and bound the overall delay in the response times to both internal as well as external events. To achieve this goal we have modified the Ultrix4.2 kernel to be preemptible at highly specific locations and have added a small subset of data structure locks to maintain data consistency. Work has also been done towards reducing and bounding the time spent resolving lock conflicts. The end result is that we have a kernel capable of the real-time processing necessary for multimedia applications. Finally, we demonstrate and validate through experimentation that the above claim is true.

1 Introduction

When we talk about real-time in the Tenet Group in a broad sense we are referring to guaranteed performance, however in this paper, real-time as in real-time scheduling, refers to scheduling processes under bounded response times. In

* This research was supported by the National Science Foundation and the Defense Advanced Research Projects Agency (DARPA) under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives, by AT&T Bell Laboratories, Hitachi, Ltd., Hitachi America, Ltd., Pacific Bell, the University of California under a MICRO grant, and the International Computer Science Institute. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing official policies, either expressed or implied, of the U.S. Government or any of the sponsoring organizations.

order to achieve such bounded response times there needs to exist greater control over scheduling behavior. To achieve this greater control several areas of real-time functionality have been added to the Ultrix4.2 kernel. First, we discuss in detail the implementations in each of these areas. Then we proceed to validate the additional functionality through measurements. We note that in order to achieve valid and accurate measurements, all areas of the entire system must be vigorously and continuously exercised. It is only through this type of exhaustive measurement process do we establish guarantees about the work.

In addition, it must be understood that we cannot totally solve this problem of bounding the existing delays by software alone. We do not attempt to reduce nor bound the delays caused by interrupt processing and/or DMA processing, even though both types of high priority activities in the lower half of the kernel may produce very noticeable delays. To adequately bound such delays would require hardware support.

The rest of this paper is organized as follows: Section-2 states our objective. Section-3 describes and discusses our implementation. Section-4 describes our performance measurements. And, section-5 gives our conclusions.

2 Objective

Scheduling latency is defined in the context of this paper to be the delay between the time when an event (or request) takes place and the time when execution to handle (or respond) to that event actually occurs. The primary goal of our work is to reduce and bound the duration of this delay. In particular, the target is to reduce this delay to below *3ms* for close to %100 of the time while the machine is operating under loaded conditions.

3 The Implementation

The fundamental mechanisms that make up the real-time scheduling features added to the Ultrix4.2 kernel are included in 5 areas: real-time process establishment, preemption points, data structure locks, an implemented scheme for resolving lock conflicts, and a separate real-time callout mechanism. This combined functionality provides for the scheduling control needed to reach the above stated objective.

3.1 Real-Time Process Establishment

The notion of a real-time user process has been introduced into the Ultrix kernel. In this context, a real-time process is simply a normal user process that has explicitly been accelerated to real-time status. It is this real-time status, which any user-process in the system is equally able to gain, that we would like to describe. In order to obtain this real-time status several things must occur:

Admission control. First of all, the kernel uses a simple admission control scheme to accept or reject a process's request to obtain real-time status. The scheme, at this point, is straight forward and simply allows only 4 (user-requested) real-time processes to exist at any one time, and rejects all others. It is possible, however, for the system to accelerate specific processes, and exceed this limit. This is done to reduce and limit the duration of time spent handling lock conflicts and will be described later (see section 3.4). In the future, we expect this admission control scheme to take into account multiple user-defined real-time priorities as well as user-defined resource requirements (or requests). The user process, under this new scheme, would then negotiate and haggle with the kernel until an agreement, if any, is made about the nature of the real-time status to be allocated.

Initialization. After an approval (or agreement) has been made, the process's pages are locked in memory, an entry in the system wide real-time process table is allocated and initialized, and the user/kernel process priorities are changed to a static real-time UNIX priority (see next section). And finally, a global flag is set signaling the kernel that execution of a real-time process is now pending and that preemptions are now enabled. Once a preemption has occurred and the real-time execution is under way, the global flag is turned off, preemptions are then disabled and the kernel proceeds to operate as normal.

Assignment of Priorities. A real-time process has associated with it several priorities. These are a static UNIX priority and 3 kinds of real-time priorities.

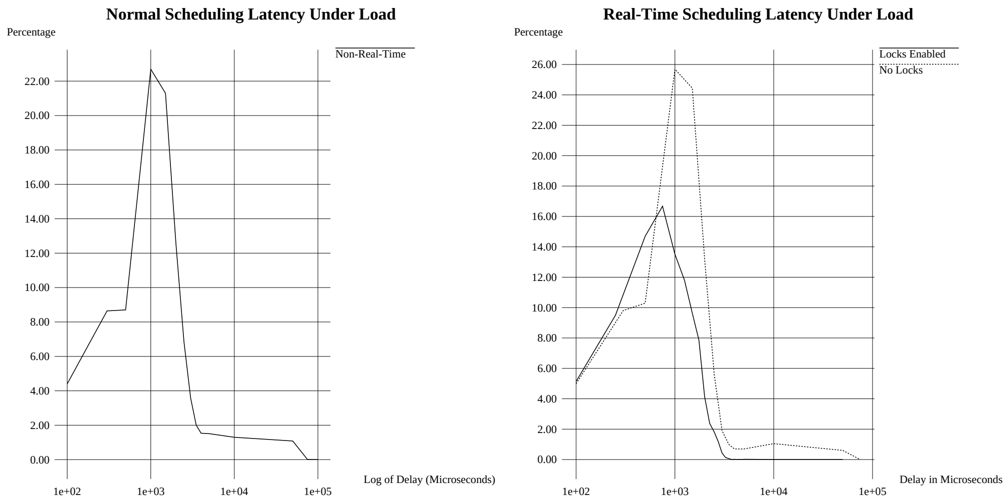
When a process is first accelerated to real-time status it is assigned a static UNIX priority. This priority, referred to as the process's real-time UNIX priority, defines the overall position of the priority of real-time processing in relation to all other high priority activities in the system.

In order to distinguish between real-time processes of various urgency we have adopted the notion of global, inherited and effective priorities from SunOS5.0 [Khanna92]. The use of these priorities is further described in section 3.4.

3.2 Preemption Points

Preemption points are highly specific locations in the kernel where a process may choose to give up its execution of the CPU in favor of a higher priority process. This approach is much simpler for us than putting locks on all data structures and making the entire kernel fully preemptible. We do not say that this is the correct approach, but only that it's the simplest way for us to achieve our needs which are to satisfy the requirements of multimedia applications. Other work has successfully been done to make the kernel fully preemptible, for example in OSF-1.0, SunOS-5.0, and HPUX.

Initially our goal was to see how far we could get (how many effective points we could add and far we could reduce and bound the scheduling delay) without having to implement locks over any data structures. As a result, the first generation of preemption points we added totaled 11 and our measurements reflected a major improvement in the overall average and maximum scheduling response times (see figs. 1, 2).



Normal Processing, Preemption Disabled   Real-Time Processing, Preemption Enabled

Fig.1. Average Delay Times Between Execution Request and Execution.

After having implemented locks over a small set of data structures (see next section) we were able to, one by one, add 31 more preemption points. It was at this time where we made our final measurements (described in section 4).

3.3 Data Structure Locks

As stated above, our goal in the beginning was to avoid locking any data structures and to reach the highest level of kernel preemptibility possible without using these them. We soon realized though that this goal was unrealistic (see fig. 1) and proceeded further with the new goal of implementing locks over as few a number of data structures as possible while still achieving our above stated target.

Our next move was to implement a mechanism in the kernel to determine exactly which data structures needed locking for any given point within the kernel. As a result, we were able to come up with a small subset of data structures, totally 27, that would need protection when all the preemption points were in place and operational. This small subset of total kernel data structures then became the set of structures to which we applied our version of real-time locks to.

The real-time locks we've implemented are, at this point, only simple mutual exclusion locks (mutexes). In the near future we intend to implement multiple readers, single writers (readers/writer) locks as well. These mutexes are primarily in place to prevent races on access to each of the shared data structures in the small subset mentioned above.

An important point is that under normal sleep situations, locks are never held. But during real-time processing when a lock conflict is discovered, the process can go to sleep, blocked waiting for the lock while holding onto other locks. This is ok in this situation. Work has been done to ensure that locks are never acquired out of order.

3.4 Priority Inheritance

To limit the duration of time spent handling lock conflicts (situations where lower priority processes hold locks needed by higher priority ones) a strategy referred to as the *basic priority inheritance protocol* was adopted from SunOS-5.0 [Khanna92]. A more elaborate description may be found in [Sha90]. In short, the priority inheritance protocol comes into play only when higher priority processes block while waiting to obtain locks. It attempts to limit the duration of this time spent blocking by having the high priority process propagate (will) its priority to all lower priority processes that currently block it. In our particular case, the lower priority process is ramped up to real-time status and the priority that it inherits is the real-time process priority of the process(es) blocked waiting for it. When lower-priority processes cease to block a high priority process, the lower priority ones revert back to their original priority, which in our particular case may mean to giving up the newly acquired real-time status. The strategy was modified and developed for use with processes instead of threads and to operate in the context of a kernel only preemptible through specific preemption points.

Under normal operations the amount of time spent blocking is potentially very long–in fact, it is unbounded since the amount of time a high priority process must wait for a lock to be freed depends not only on the duration of some critical sections, but possibly on the duration of the complete execution of some processes. However, with the use of priority inheritance this time is bounded. It is described in [Rajkumar88] how, if we have m lower priority processes holding onto k distinct locks, the maximum number of critical sections that a real-time process will have to wait is at most $MIN(m,\ k)$.

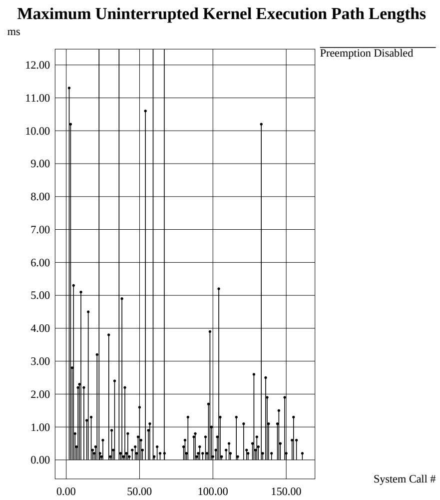
3.5 Internal Event Processing

Callout processing is the traditional UNIX way of handling internal timeout events. In a standard UNIX kernel like Ultrix4.2 this processing is all done at software interrupt level. This processing, however, can produce unexpected, variable delays even while the a real-time process is in the middle of execution. In fact, in SunOS 4.0 and 4.1 this delay has been measured to be up to *5ms*. This is unacceptable for real-time processing.

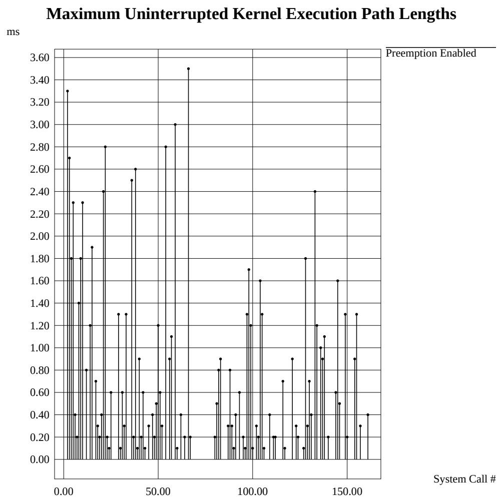
Currently, what we've done is implement a separate real-time callout queue for processing especially time-critical internal events. And, we've taken the approach of blocking out all other normal callout processing during the execution of real-time processes.

4 Measurements

As we have noted in the introduction, we need to vigorously and continuously exercise all pathways throughout the entire kernel. Otherwise, we may miss the affects of exercising certain pathways and later suffer either previously undetected race conditions and/or unexpected delays. Therefore, we have taken care to develop scripts that spawn processes of all types to load our machines in the best way we can.



Normal Processing, Preemption Disabled



Real-Time Processing, Preemption Enabled

Fig.2. System Call Uninterrupted Kernel Execution Path Lengths.

Uninterrupted kernel execution times. This experiment was designed to observe where in the kernel we would experience the greatest uninterrupted execution path lengths. The initial idea was to determine where preemption points needed to be placed and to gain a better understanding of where the greatest scheduling delays would arise from. As a result, we can see the affects the preemption points have on reducing the length of uninterrupted execution path lengths (fig 2). Note, that in the graph with preemption enabled, we can see that all system calls now have uninterrupted execution path lengths almost entirely under *3ms*.

Delay between wakeup and actual execution. This experiment was conducted to determine the exact scheduling delays that take place under actual execution between when a process wakes up and when it actually runs. Measurements were taken with a normal process and real-time processes before and after locks and many preemption points were added (fig 1). Note, that in order to have reasonably legible graphs here, we've displayed the log of the X values in both graphs. ***Also note, that after our final generation of preemption points were added (those added after locks were enabled) that our original objective of bounding this delay to under *3ms* has been achieved***.

5 Conclusions

Work has been done in the Ultrix Kernel to help reduce and bound scheduling latencies so as to better support the real-time processing requirements of multimedia applications. In particular, work has been done to add preemption points and data structure locks and a scheme has been implemented to help resolve lock conflicts in a straight forward way. In particular, we have been able to bound the scheduling latency to be almost entirely under *3ms*, which was our original target. We hope to continue this work in the area of supporting multiple, simultaneous, co-existing real-time processes.

References

[Eykholt92] . R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing: Multithreading the SunOS Kernel", *Summer 1992 USENIX,* June 8-12, 1992, San Antonio, Texas.

[Herrtwich90] . G. Herrtwich, "An Introduction to Real-Time Scheduling", *ICSI Technical Report, TR-90-035,* Jul. 1990.

[Khanna92] . Khanna, M. Sebree, and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0", *Winter 1992 USENIX,* San Francisco, California.

[Leffler89] . J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System,* Addison-Wesley, 1989

[Nakajima91] . Nakajima, M. Yazaki and H. Matsumoto, "Multimedia/Realtime Extensions for the Mach Operating System", *USENIX Proceedings Summer '91 - Nashville,* Jun. 1991, p183-198.

[Rajkumar88] . Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", *Proceedings of the Real-Time Systems Symposium,* Dec. 6-8, 1988, Huntsville, Alabama.

[Sha90] . Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers,* Vol. 39, No. 9, September 1990.

[Tokuda87] . Tokuda, J. W. Wendorf and H. Y. Wang, "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems", *Proceedings - IEEE Real-Time Systems Symposium,* Dec. 1987, p271-280.

This article was processed using the LaTeX macro package with LLNCS style