Real-Time Mach Timers: Exporting Time to the User

Stefan Savage and Hideyuki Tokuda

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
{*savage,hxt*}@*cs.cmu.edu*

Abstract

The current CMU Mach 3.0 microkernel exports simple timestamp and delay abstractions through `host_get_time()` and a timeout parameter to `mach_msg()`. While this is sufficient for many purposes, it does not provide the precision or generality required for a variety of real-time applications. In this paper we describe extensions to CMU's Mach 3.0 which provide users with flexible time-based synchronization and timestamp services. Additionally, we will describe how timing and scheduling services are integrated to allow real-time applications to handle *timing faults*.

1 Introduction

Modern operating systems are expected to provide services to allow application programs to synchronize with the passage of time. The BSD UNIX interface includes `gettimeofday()`, `sleep()`, `signal()` and `select()` to provide this functionality[7]. Similarly, Mach 3.0 exports `host_get_time()` and a timeout parameter to `mach_msg()`[8]. While these timing services are sufficient for a broad range of general purpose applications, real-time applications frequently require greater precision and flexibility. Real-time applications may be loosely defined as applications in which the relationship between execution behavior and the passage of time directly impacts program correctness. Included in this class of software are applications for factory automation, military command and control, robotics, and the rapidly growning field of continuous media[6]. Timestamps, exact delays, periodic signals and timeouts are among the services which may be demanded by these types of applications.

There are several requirements for providing this kind of support. First, it is important to be able to measure time with a high degree of precision. Second, users must be able to synchronize with this high precision time source. Third and last, it is necessary to integrate scheduling and synchronization so that computation can be modified in the presence of *timing faults*, or errors in temporal correctness.

This research was supported in part by the U.S. NRaD under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, and by the Federal Systems Division of IBM Corporation under University Agreement YA-278067. 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 NRaD, ONR, DARPA, IBM, or the U.S. Government.

We have developed this necessary functionality in Real-Time Mach, an extension to CMU's Mach 3.0 which provides predictable scheduling and priority consistent synchronization[15, 14]. Time services are exported to the user in the form of two abstractions: *clocks* and *timers*. Timers are active objects which allow users to synchronize with time in a variety of ways. Clocks are devices which measure the passage of time and support the use of timers to a particular degree of accuracy. In the next sections we will examine these two abstractions more closely and the motivation for their design.

2 Clocks

In first implementing our time services we were faced with two problems. The first was our desire to allow the use of additional time measurement hardware to be integrated seamlessly for those applications which required high precision. Second, we encountered a problem with the lack of resolution stability in Mach 3.0's representation of the host time. In order to support synchronized time in a distributed environment while still preserving monotonicity[12], Mach 3.0 provides the `host_adjust_time()` interface. This interface allows the host time to be skewed at resolutions under one scheduling clock tick. Consequently, the time between two scheduling clock ticks is not always measured as such. The solution we reached to these problems was to create the abstraction of clocks, which measure the passage of time. Clocks can be used for accurate timestamps, measurements of CPU usage, or for basing representations of the time of day. Timers can be bound to different clocks depending on application need which allows the simple introduction of high resolution timing hardware. Additionally the problem of the host time representation disappears since clocks represent a local, device specific time. "Host time" is a system personality specific notion and belongs outside the kernel along with operations on its skew.

2.1 Functionality

In essence, a clock is simply a piece of hardware which measures the passage of time. It was logical therefore to represent clocks as Mach devices. This limits the number of additional abstractions and interfaces in the kernel and facilitates the addition of specialized time measurement hardware.

Clocks are manipulated using the standard Mach 3.0 device interfaces. As with any device, a clock must first be opened using device_open() with the name of the clock device. The names for these devices are machine dependent so we have provided a universal macro, `CLOCK_REALTIME`, which maps to a baseline clock device on all platforms. Clocks are generally manipulated through the `device_get_status()`, `device_set_status()`, and `device_map()` interfaces although the particular functionality allowed is hardware dependent. Typical operations include getting the time, getting the resolution, setting the resolution, or mapping the clock into a user's address space.

2.2 Implementation

For all system architectures which we have supported to this point (i486 AT, DECstation, and SUN3) the baseline clock has been based on the scheduling clock tick. In the case of the i486 AT and the DECstation where the clock resolution may be altered, it was necessary to multiplex the kernel `clock_interrupt()` routine on top of this baseline clock device. For instance, when the baseline clock on the i486 AT is directed to increase its resolution to 1ms, the scheduling clock tick frequency increases by 10. To preserve the illusion of the standard 10ms scheduling clock tick, the `clock_interrupt()` routine is executed only at every tenth interrupt. When the interrupt frequency is not an integral divisor of the scheduling clock tick frequency, the scheduling clock tick will be temporarily skewed over the period of the modulo of the two. This seemed reasonable since the short term inaccuracy of this number has relatively minor effects, and quantum based scheduling is a poor choice for real-time programs anyway.

In general, a clock device contains an interrupt handler, a representation of the current time and a timer queue. Pending timers are inserted into this queue in time order. When a clock device receives an interrupt it checks the head of the queue to see if any timers have expired. If so it sets an AST to process the timers at a safe point. Additionally, clock devices may provide services to map representations of the current time into a user's address space, to change clock resolution, or to obtain higher resolution timestamps than are available via the mapped device interface.

There is a great variance in the timing hardware, especially with respect to achievable interrupt resolutions. The MC146818 clock chip on the DECstation, for example, will only interrupt at frequencies which are powers of 2. Alternatively, the I8254 clock chip on the i486 AT will interrupt at any multiple of 838ns up to .055 seconds. To retain machine independence in our interface yet preserve the power of the underlying hardware we let the user request clock resolution changes in terms of a desired resolution and an allowable skew from that resolution. If the clock driver cannot be set to a resolution within (desired_resolution - skew) then it will return failure. The correct resolution may be queried later if this information is important to the user.

3 Timers

Clock devices are sufficient for high resolution timestamps but to provide blocking synchronization requires a stateful object. This prompted the design of a timer abstraction. Timers are kernel-exported objects with three properties, an expiration time, a synchronization action to be taken, and an activity state. An active timer will perform its synchronization action when its expiration time is reached.

A timer is created using the `timer_create()` call. It is bound to a task and to a clock device which will measure the passage of time for it. The task binding is useful for ownership and proper tear-down of timer resources. If a task is terminated, then the timer resources owned by it are terminated as well. It is also possible to explicitly terminate a timer using the `timer_terminate()` call.

Timers allow synchronization primarily via two interfaces. `timer_sleep()` is a synchronous call similar to the POSIX 1003.4 `nanosleep()`[13]. The caller specifies a time to wakeup and indicates whether that time is relative or absolute. Relative times are less useful for real-time software since program-wide accuracy may be skewed by preemption[4]. For example, a thread which samples data and then sleeps for five seconds may be preempted between sampling and sleeping, causing a steadily increasing skew from the correct time base. Timers which are terminated or canceled while sleeping return an error to the user.

`timer_arm()` provides an asynchronous interface to timers. Through it the user specifies an expiration time, an optional period, and a port which will receive expiration notification messages. When the expiration time is reached, the kernel sends an asynchronous message containing the current time to this port. If the timer is specified as periodic then it will atomically re-arm itself with the specified period relative to the last expiration time.

Lastly, `timer_cancel()` allows the user to cancel the expiration of a pending timer. For periodic timers, the user has the option of canceling only the current expiration, or all forthcoming expiration.

These last facilities, periodic timers and partial cancellation, are important because they allow an efficient and correct implementation of periodic computation. This permits a user level implementation of Real-Time Mach's periodic threads.

4 Timing Faults

The original motivation for this work came from a limitation in Real-Time Mach's thread package. While users could specify a deadline for periodic threads, this deadline would only be checked at the end of a thread's periodic computation. Effectively this meant that the user was only notified at the start of the next periodic computation. Moreover, aperiodic deadlines were largely ignored. It was clear that we needed some sort of asynchronous notification mechanism to allow such timing faults to be acted upon aggressively. We looked first into the use of the exception mechanism, but its current design left open the possibility of extended priority inversion in the kernel. This is because the exception mechanism *knows* that the exception was caused by the current thread. Therefore a timing fault registered by a low priority thread could execute in the context of a high priority thread, and keep it blocked in the kernel indefinitely. Instead of radically changing the exception code we designed a new asynchronous mechanism which led to the creation of a generalized timer service.

The timer interface supports timing faults via a flag in `timer_arm()` which indicates that the calling thread should be suspended when timer expiration occurs. The interface is important because it allows timing fault handling to export exception-like semantics. When a timing fault occurs, the offending thread is suspended and a message is sent to its deadline handler which will then take corrective action and possibly resume the faulting thread. This action is left to the user, although in practice the most frequent actions include a change of scheduling parameters, user notification, or for hard real-time programs, thread termination. To simplify programming we have added library support for deadline handling in the context of our *rt_thread* package[15]. The following library routine:

```
thread_deadline_handler(mythread, mythread_attr, entry, arg)
```

is sufficient to insure that when `mythread` misses a deadline the handler `(*entry)(arg)` will be invoked.

5 Performance

All benchmarks were performed on a Gateway 2000 486DX2 66Mhz system with 16 megabytes of 70ns ram, and 64 kilobytes of 25ns secondary cache. Both primary and secondary caches were warm. The system was runing Real-Time Mach version MK78 with CMU Unix server version UX39 running in single user mode. Additionally, the network was disconnected and the benchmarks were scheduled with the highest thread priority. We used a STAT![1] timer board to take measurements accurate to 1μsec.

5.1 Timer operations

We measured a series of microbenchmarks to evaluate the execution cost of each new timer operation. These measurements were repeated 1000 times and the average taken (controlling for clock and watchdog interrupts and their effect on the cache, the variance was very small)

Table 1 summarizes the latency of the timer operations. The measurements of `timer_arm()` are best case numbers in which there are no pending timers earlier in the clock queue. When other timers intervene, the cost increases linearly by a factor of 600ns per timer. The periodic form of `timer_arm()` is slightly longer that the one-shot but there is significant amortized savings since

operation	time in μsecs
timer_arm() [one-shot]	33
timer_arm() [periodic]	37
timer_cancel()	19
timer_create()	220
timer_terminate()	119
mach_thread_self()	20
null trap	7

Table 1: *Latency of timer operations*

expiration action	time in μsecs
unblock thread	106
send message	162

Table 2: *Latency of thread execution from time of interrupt*

timer reset costs are 1μsec (again subject to the worst case *O(n)* insertion time for the clock queue). These numbers could be improved using tree-based or timing wheel algorithms[16].

The `timer_create()` and `timer_terminate()` are substantially slower than other primitives largely because they are involved and allocating memory and port structures. Additionally both of these operations are implemented using IPC while the rest of the timer operations are invoked directly through traps. The times for the `mach_thread_self()` and null traps are given for reference.

5.2 Execution latency

In table 2 we see the latency between the time a clock device interrupt occurs and a thread waiting on an expired timer starts to execute. The two cases represent two different timer expiration actions: threads which are directly blocked on a timer using `timer_sleep()`, and threads which are waiting for a timer expiration message. MIG stub, copyout and general IPC overhead all contribute to the 56μsec difference. We expect that a handcoded IPC stub in the kernel and general IPC improvements should make these numbers comparable. Also, we believe that the scheduling overhead component of these numbers can be reduced by at least 40μsecs.

5.3 Clock operations

The latency of clock device operations is bounded by the performance of the device server in the kernel. Table 3 shows a value of 85μsecs for timestamps via IPC, and this is typical for all clock

operation	time in μsecs
timestamp (using device_get_status)	85
timestamp (using mapped page/io port)	<2

Table 3: *Latency of clock timestamp operations*

operations. However, for some clock devices, changes in resolution may be postponed from the time of invocation for the duration of one clock device tick.

These numbers suggest that all timestamps should be obtained via mapped I/O, but this is not necessary true. For some clock devices the interrupt resolution differs by several orders of magnitude from the resolution observable from supervisor state. For example, i486 AT systems and some SPARC systems are able to obtain timestamps of accurate to 1-4μsecs in the kernel, but periodic interrupts only occur every 1-10ms. In these cases, application programmers may wish to pay the extra overhead of invoking the kernel in exchange for increased precision.

5.4 Overhead

Clock device queue processing costs 1μsec at interrupt level. Changing the clock granularity increases the overhead of this routine proportionally. It has an adverse effect (sometimes disastrous) effect on locality of reference in the cache. For reference, the regular `clock_interrupt()` routine costs 9μsecs before the addition of this code.

6 Related Work

POSIX 1003.4 defines both timer and clock interfaces which provided much of the inspiration for our own time interfaces in Real-Time Mach. Real-Time Mach's clocks and timers provide a superset of POSIX functionality, including the ability to change clock granularity dynamically and to effect scheduling decisions via the timing fault mechanism. Otherwise the interfaces are very similar.

Microsoft's NT executive provides *timer objects* for the purposes of time based synchronization [5, 11]. In addition to blocking time synchronization, timer objects may be associated with APC's (Asynchronous Procedure Calls) which allow the kernel to asynchronously set a thread's context to execute a particular function when a timer expires. Within the kernel and device drivers, timer objects may also be associated with DPC's (Deferred Procedure Calls) which perform a similar function but are invoked in the current execution context. NT timer objects invoke expiration actions using an AST-like mechanism much like Real-Time Mach does. Unlike our system however, NT does not provide the user with the ability to bind these objects to different clock devices.

The OSF RI has designed and implemented a similar set of interfaces for their microkernel called "clocks" [10, 9]. The OSF interfaces export services for getting timestamps, setting clock resolution, exact delays, and setting alarm messages. Alarms which are found to have expired at interrupt time are copied to a special alarm queue. A high priority kernel thread is unblocked and it sends alarm messages on behalf of the kernel. While there is substantial similarity between our work and that of OSF, we feel that the interfaces and mechanisms produced at CMU are smaller, more consistent with Mach 3.0, and have superior functionality. Briefly, CMU timers support periodic reset, time faults, and timer cancellation functionality which is not present in the OSF interface. CMU clocks are standard Mach 3.0 devices while OSF names their clocks via "`clock_id`'s" which are raw integers interpreted by the kernel. Lastly, CMU's implementation uses an AST instead of a kernel thread to dispatch timer messages, thereby avoiding the cost of additional queueing and context switch operations.

7 Current Status and Future Work

The previously mentioned time services have been integrated successfully into Real-Time Mach across a range of hardware. Specifically, we have supported the MC146818 clock chip on the

DECstation, the I8254 and MC146818 clock chips as well as the STAT! timer board for the i486 AT, and the standard SUN3 clock chip. The vast majority of this code is machine independent and adding new clock devices extremely simple.

We are working closely with the CMU Mach project to integrate this code along with the base Real-Time Mach code into the mainline CMU Mach 3.0 distribution. As a part of this merge some of the interfaces and mechanisms may change. With respect to this paper, we are investigating ways to allow timing faults to be handled within the exception mechanisms. Additionally, we hope that all kernel timing, such as device timeouts and pc sampling, can be subsumed by this single mechanism. In addition to removing duplication of effort this will simplify machine independent support for particular timing functionality, such as BSD's relatively prime profiling timer[7]. Lastly, we are examining ways in which the timer mechanisms can be integrated with first class user level threads[6, 2, 3] to provide low overhead real-time threads.

8 Conclusion

We have shown that there are three requirements for providing time services for real-time programs. These are, high accuracy time measurement, synchronization to these time sources, and integration with scheduling. We have implemented solutions to all of these problems in an interface which is clean, efficient and consistent with the Mach 3.0 design philosophy.

References

[1] Alpha Logic Technolgies. *STAT! System Timing Analysis Tool User Guide*, 1992.

[2] T.E. Anderson, B. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Mangement of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, October 1991.

[3] P. Barton-Davis, D. McNamee, R. Vaswani, and E.D. Lazowska. Adding Scheduler Activations to Mach 3.0. Technical Report 92-08-03, University of Washington, October 1992.

[4] V. Blazquez, L. Redono, and J.L. Freniche. Experiences with “Delay Until” for Avionics Computers. *Ada Letters*, 12(1):65–72, January 1992.

[5] H. Custer. *Inside Windows NT*. Microsoft Press, 1992.

[6] R. Govindan and D.P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, October 1991.

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

[8] K. Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie-Mellon University, January 1992.

[9] K. Loepere. *OSF Mach Draft Kernel Interfaces*. Open Software Foundation and Carnegie-Mellon University, October 1992.

[10] K. Loepere. *OSF Mach Draft Kernel Principles*. Open Software Foundation and Carnegie-Mellon University, October 1992.

[11] Microsoft Corporation. *Preliminary Windows NT Device Driver Kit*, 1992.

[12] D. Mills. Experiments in Network Clock Synchronization. DARPA Network Working Group Report RFC-957, August 1985.

[13] Realtime Extension for Portable Operating Systems. Proposed Standard IEEE P1003.4/D12, February 1992.

[14] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[15] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of the USENIX Mach Workshop*, October 1990.

[16] G. Varghese and T. Lauck. Hashed and Hierarchical timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, November 1987.