Processor Capacity Reserves:
Operating System Support for Multimedia Applications

Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Multimedia applications have timing requirements that cannot generally be satisfied using the time-sharing scheduling algorithms of general purpose operating systems. Our approach is to provide the predictability of real-time systems while retaining the flexibility of a time-sharing system. We designed a processor capacity reservation mechanism that isolates programs from the timing and execution characteristics of other programs in the same way that a memory protection system isolates them from outside memory accesses. In this paper, we describe a scheduling framework that supports reservation and admission control, and we introduce a novel *reserve* abstraction, specifically designed for the microkernel architecture, for measuring and controlling processor usage. We have implemented processor capacity reserves in Real-Time Mach, and we describe the performance of our system on several types of applications.

1 Introduction

Multimedia applications require that operating systems support time-constrained data types such as digital audio and video in a responsive and predictable way. The general purpose resource management policies found in most operating systems are incompatible with these strenuous timing constraints, and traditional real-time systems are also poorly matched to the multimedia application environment since user demands dictate a dynamically changing mix of both real-time and non-real-time activities. Moreover, supporting commercial "shrink-wrapped" real-time multimedia software has the additional requirement that the operating system must provide a sensible framework for controlling and communicating resource usage among independent real-time activities.

1.1 Our solution

We have designed a *processor capacity reservation* mechanism which allows the user to control the allocation of processor cycles among programs. The model supports both real-time and non-real-time activities. Applications request processor capacity reservations, and once a reservation has been granted by the scheduler, the application is assured of the availability of processor capacity. Applications are also free to increase their reservations at any time during execution (subject to the availability of additional resources), and they are always free to decrease their reservations. This design combines the predictability of reservations with the flexibility of dynamically adjusting reservation levels to accommodate a changing application mix or changing timing requirements within applications.

A new kernel abstraction, called a *reserve*, tracks the reservation and measures the processor usage of each program. The scheduler utilizes these usage measurements to enforce reservations, ensuring that programs cannot monopolize computational resources. Additionally, reserves may be passed across protection boundaries during IPC calls. This is especially important in microkernel systems which employ separately scheduled servers to provide various system services, and consequently, the true processor usage of a program includes the processor usage of the servers invoked by that program.

Our reservation system was designed to support higher-level resource management policies. For example, a quality of service (QOS) manager could use the reservation system as a mechanism for controlling the resources allocated to various applications. The QOS manager would translate the QOS parameters of applications to system resource requirements (including processor requirements), possibly with the cooperation of the application itself. The manager would then be able to reserve the resources for each pro-

This work was supported in part by a National Science Foundation Graduate Fellowship, by Bellcore, and by the U.S. Naval Ocean Systems Center under contract number N00014-91-J-4061. 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 NSF, Bellcore, NOSC, or the U.S. Government.

gram, and it could use performance feedback or interactive user input to change the reservations for various applications under its control.

Thus we have introduced a useful structure that divides the problem of scheduling based on QOS requirements into two parts: a policy for allocating system resources based on application-level QOS requirements, and an abstraction and mechanism for scheduling and controlling those resources. Our model of processor capacity reserves provides this abstraction and mechanism.

1.2 Current state of the art

Operating system support for digital audio and video typically includes only primitive scheduling support for continuous media programs. This support usually comes in the form of a fixed-priority extension to a time-sharing scheduler. Continuous media applications can then run at the highest priority without interference from non-time-constrained applications. However, such an arrangement does not protect high-priority applications from interference due to interrupt processing or from interference due to other high-priority applications. Having programs make reservations for their computational requirements allows the scheduler to decide whether they can be scheduled successfully and thus whether a new reservation request can be accepted. A program which has its reservation request refused is free to modify its timing constraints and request service at a lower rate or to request (through some higher-level server) an adjustment of capacity usage by other programs.

The possibility of persistent processor overload presents additional problems for current operating systems. Simple fixed-priority scheduling does not help to detect or prevent potential overload conditions. Attempting to execute two continuous media applications which overload the system results in neither of the programs being able to meet its timing constraints. Under a reservation scheme, the system would admit only one of the programs, and the other program could change its timing parameters and request service under the new parameters. A dynamic reservation strategy prevents overload by refusing to admit new programs which would result in an overloaded processor.

1.3 The rest of this paper

In the rest of the paper, we describe our model of processor capacity reserves more detail. We discuss our reservation strategy in Section 2, and in Section 3 we explore issues in admission control and scheduling with reservations. Section 4 describes the new reserve abstraction and discusses usage measurement and reservation enforcement. In Section 5, we present a performance evaluation which illustrates the behavior of several kinds of programs that use reservations. Sections 6 and 7 discuss future work and related work, and we make a few concluding remarks in Section 8.

2 Our reservation strategy

The previous section motivated the design of a processor capacity reservation system. Here we examine the mechanisms which are necessary to enable this type of resource management. The reservation strategy must:

1. provide some means for application programs to specify their processor requirements,

2. evaluate the processor requirements of new programs to decide whether to admit them or not,

3. schedule programs consistently with the admission control policy, and

4. accurately measure the computation time consumed by each program to ensure that programs do not overrun their reservations.

In the following sections, we example each of these points in more detail.

2.1 Capacity specification

The first requirement depends on a consistent scheduling model that can accommodate different kinds of program timing requirements. For example, an audio application might be scheduled every 50 ms to generate an audio buffer. Many programs have no time constraints at all and run as fast as possible for as long as the computation takes. Processor percentage provides a straightforward measure to describe the processor requirement of both of these kinds of programs. Processor percentage is the processor time required by a program during an interval divided by the real time of the interval, and the processor percentage consumed by a program over time defines its rate of progress.

Periodic programs (which execute repeatedly at a fixed interval) have a natural rate described by their period and the computation time required during each period, assuming the computation time is fairly constant. When the computation time is not constant, a conservative worst case estimate reserves the necessary processor capacity, and the unused time is available for background processing. Programs that are not periodic have no natural computing rate, but we can assign them a rate. This rate will determine the duration of time until the program completes, and the rate must be reserved based on the delay requirements of

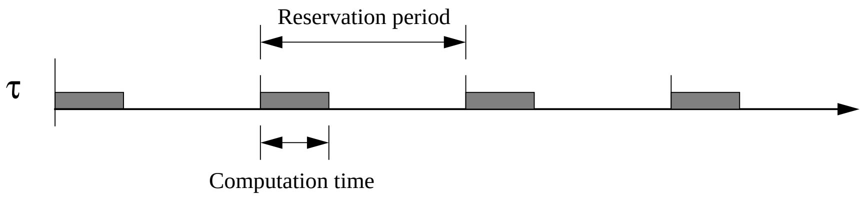


Figure 1: Periodic Framework

the program. Figure 1 illustrates the computational requirement of a periodic program τ. With this kind of framework, we can use scheduling algorithms which schedule programs according to their allocated rate.

2.2 Admission control

The second requirement, that the scheduler can evaluate the timing constraints of new programs against the available capacity, calls for a scheduling framework that translates the processor requirements specified by individual programs into utilization measures which can be used in an admission control policy. We shall see that simply summing the individual utilizations of all executing programs will work under a dynamic priority scheduling discipline, but there are drawbacks to this approach. Another approach is fixed priority scheduling with an appropriate priority assignment, but this method does not, in general, allow us to reserve the full 100% of the processor.

2.3 Scheduling

The third requirement, that the scheduling policy schedule programs in a way that is consistent with the admission control policy, reflects the fact that the scheduling framework must be consistent across all of the resource management policies in the system. If the scheduling policy does not support the assumptions made by the admission control policy about how programs are ordered for execution, the reservation system will fail to operate properly.

2.4 Reservation enforcement

The fourth requirement, that the scheduler accurately measure usage and enforce reservations, demands precise performance monitoring software which typical operating systems do not provide. Operating systems usually accumulate usage statistics for each process by sampling during regular clock interrupts [7], but this information is imprecise over short intervals. Furthermore, the execution behavior of the monitored program must be independent of the sampling period. A more precise mechanism measures durations between context switches and accounts for interrupt processing time and other system overhead.

Even if the system can accurately measure capacity consumption on a per-process basis, other problems arise. Usage statistics in traditional operating systems consist of system-level usage time and user-level time for each process. For monolithic operating systems, this approach is sufficient, but for microkernel systems where operating system services are offered by different user-level servers [4], the usage statistics of an activity cannot be found in the usage statistics of a single process. An activity may invoke separate operating system servers to perform filesystem access, networking, naming, etc. To maintain an accurate picture of an activity's capacity consumption, the cost of these services must be charged to the activity itself rather than to the individual servers. Thus, capacity reserves must be maintained independently from any particular thread of execution so that work done by any process or thread on behalf of the reserved activity can be charged to that activity. This is accomplished simply by creating an independent reserve abstraction which may be bound dynamically to different threads of execution.

With accurate usage measurements and a mechanism to generate a scheduling event when a program attempts to overrun its capacity reservation, the scheduler can control the execution of programs and prevent them for interfering with other reservations.

3 Admission control and scheduling

A scheduling framework based on the rate of program progress provides an effective environment for implementing processor reservation. We can associate rates with periodic and non-periodic programs as described in the previous section. The rate of a periodic program can be determined from the period that the programmer has in mind and the computation time during that period. For non-periodic programs, the rate arises from delay requirements. In either case, the rate alone does not fully specify the timing attributes of a program; the computation time and period are essential.

For example, a program could specify that it requires 30% of the processor time to run successfully on a given machine (processor reservation is unavoidably machine-dependent). But now the question is how to measure the 30%. Does the program require 30 milliseconds (ms) out of 100 ms? Or would 300 ms out of 1000 ms be sufficient? These two possibilities are very different: a program that is designed to output a video stream at 10 frames per second needs 30 ms out of 100 ms, and 300 ms at the end of each 1000 ms period is not sufficient to meet the timing requirements of each 30 ms burst of computation. On the other hand, a 60-second program compilation that requests 30% of the processor does not need to have its computation time spread out with 30 ms every 100 ms, incurring much context-switching overhead; in this case 300 ms every 1000 ms would suffice.

Thus, we have three values which describe the processor requirement for a program, and two of these are required to specify a processor percentage. Let ρ be the processor percentage, C be the computation time, and T be the period of real time over which the computation time is to be consumed. Then we have

$$\rho = \frac{C}{T}.$$

We generally specify processor requirements using ρ and T since ρ is such a natural expression of processor reservation and since we think of most periodic activities in terms of the period T.

The computation time C of a periodic activity is difficult for the programmer to measure accurately, so our approach is to have the programmer estimate the computation time and then depend on the system to measure the computation time and provide feedback so that the estimate can be adjusted if necessary. For non-periodic activities that are to be limited by a processor percentage, C does not correspond to the code structure, so specifying ρ and T defines how far the computation can proceed before consuming its share of the processor for each interval. This is in contrast to periodic activities which would generally associate a code block with the computation that is to repeat during each period.

Delay for a non-periodic program which executes at a given rate can be calculated from the rate of execution and the total execution time. A program that runs at rate ρ with total computation time (service time) S will take

$$D = \frac{S}{\rho}$$

time to complete execution. This equation can also be used to derive a suitable rate given the total computation time and largest acceptable delay. Using the largest acceptable delay yields the smallest acceptable rate of execution.

We now consider how to go about scheduling programs assuming we can determine the scheduling parameters ρ, C, and T that specify the timing requirements of each program. Fixed priority scheduling is a practical policy which provides a method of assigning priorities that supports processor reservation and admission control. Dynamic priority scheduling, such as earliest deadline scheduling, is another practical method that also supports reservation and admission control.

3.1 Admission control under fixed priority scheduling

Using fixed priority scheduling in our framework requires a method of assigning priorities to programs which ensures that each program will progress at its assigned rate. The rate monotonic (RM) priority assignment of Liu and Layland [9] does just that. Under this regime, the highest priority is assigned to the highest frequency program and the lowest priority is assigned to the lowest frequency program. The rate monotonic scheduling analysis also gives us a basis for a processor reservation admission policy.

Let n be the number of periodic programs and denote the computation time and period of program i by C_i and T_i, respectively. Liu and Layland proved that all of the programs will successfully meet their deadlines and compute at their associated rates if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1).$$

When n is large, $n(2^{1/n} - 1) = \ln 2 \simeq .69$. This bound is pessimistic: it is possible for programs which do not satisfy the inequality to successfully meet their deadlines, but we cannot determine this from the Liu and Layland analysis.

An admission control policy follows naturally from this analysis. To keep track of the current reservations, we must remember the rates of the programs which have reserved computation time, and the total reservation is the sum of these rates. A simple admission control policy is to admit a new program if the sum of its rate and the total previous reservation is less than .69. Such a policy would leave a lot of computation time which could not be reserved. One possibility is to use that time for unreserved background computations. Another possibility is to use the exact analysis of Lehoczky *et al.* [8] to determine whether a specific collection of programs can be scheduled successfully, although the exact analysis is more expensive than the simple, pessimistic analysis above. In their work, Lehoczky *et al.* also gave an average case analysis showing that on average, task sets can be scheduled up to 88% utilization. So in most cases, this unreservable computation time is only 10-12% rather than 31%.

We note that the rate monotonic scheduling algorithm was analyzed under simplifying conditions. Liu and Layland [9] made the following assumptions to enable their analysis:

1. programs are periodic, and the computation during one period must finish by the end of the period (its deadline) to allow the next computation to start,
2. the computation time of each program during each period is constant,
3. programs are preemptive with zero context switch time, and
4. programs are independent, i.e. programs do not synchronize or communicate with other programs.

Subsequent work focused on ways to relax these assumptions [10, 11].

3.2 Admission control under dynamic priority scheduling

The earliest deadline (ED) scheduling policy, a dynamic priority policy, is effective for scheduling periodic programs such as our continuous media programs. We define the deadline of a computation to be the end of the period in which it started, and the earliest deadline policy chooses, at a given point in time, the program which has the smallest deadline value. Liu and Layland [9] showed that, under the same assumptions outlined in the section on rate monotonic scheduling, all programs will successfully meet their deadlines under earliest deadline scheduling if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

The admission control policy that arises from this analysis is similar to the RM strategy. We record the reserved rates of programs that have been admitted, and the total reservation is the sum of these rates. Admission control uses this sum to determine whether adding the rate of the new program would result in a sum less than 100%, and, if so, the program is admitted. If not, the reservation cannot be granted.

3.3 Discussion

Earliest deadline scheduling seems preferable to the rate monotonic scheduling since ED allows the admission control policy to reserve up to 100% of the processor whereas RM can only guarantee reservations up to 69%. As mentioned previously, the 69% bound for RM is pessimistic, and in most cases, 88% is a more realistic bound. In fact, the reservation bound of rate monotonic is 100% for the special case where all periods are harmonic, i.e. each period is an even multiple of every period of smaller duration. Additionally, an amount of unreserved computation time of perhaps 5-10% may be necessary to avoid scheduling failures due to inaccuracy in the computation time measurement and enforcement mechanisms and due to the effects of critical regions and other synchronization and communication among programs. So either of the scheduling algorithms would be appropriate for reservation scheduling.

4 Reservation enforcement and reserves

Our reservation scheme depends on an enforcement mechanism to make sure that programs do not exceed their processor reservations. The main goal is to ensure short-term adherence to the reservation with the realization that perfect enforcement is impossible due to synchronization and communication among programs. We also require that the computation time of operating system services provided by user-level servers in a microkernel architecture be accounted for and included in the reservation consumption.

4.1 Measurement and control accuracy

Our enforcement mechanism monitors processor usage by measuring the time each program is executing on the processor, and it charges this computation time against the reserve associated with the program. The reserve contains the duration of computation time accumulated in the current period, and the scheduler puts the program in a time-sharing mode when its reservation has been consumed. Programs which have not yet consumed their reservation take precedence over unreserved programs, but if there is unreserved processor time available, unreserved programs can take advantage of the extra processor time. Even though the timestamp monitoring method yields an accurate measure of the processor usage, it is not always possible for the scheduler to preempt a running program at an arbitrary point in time. For example, if a computation is in the (non-preemptive) kernel or in a critical region when it overruns its reservation, it cannot be summarily truncated, although the scheduler can easily penalize the program in its next period based on the duration of the violation.

4.2 Microkernel accounting

In many operating systems, the processor usage for each process is recorded in a per process logical clock, and the

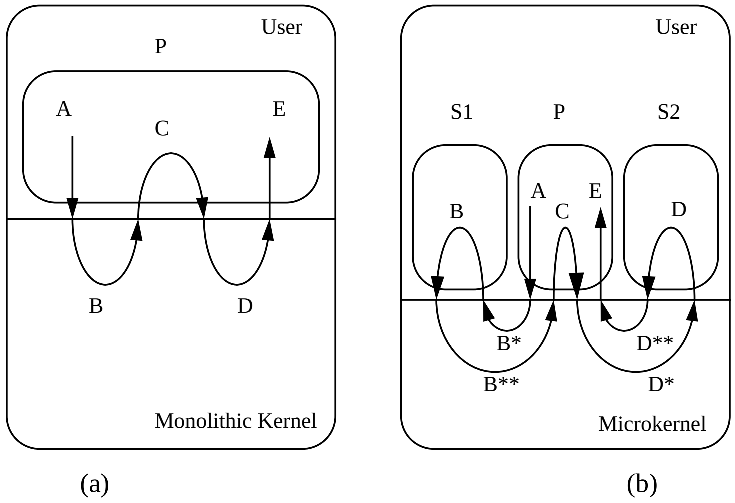


Figure 2: Microkernel Accounting

logical clock is usually divided between user time and system time [7]. The combination of these two values is an accurate long-term measure of the computation time used by the process. In a monolithic system, this is the end of the story, but in a multi-threaded microkernel architecture, per thread user and system times are meaningless. If a thread invokes various user-level servers for operating system services, the computation time consumed by the overall activity is not reflected in the user and system time of the single client thread. The computation time for that activity has been charged to the client thread and to the server threads as well. Thus, we require an accounting mechanism that accounts for the computation time of an activity being spread over a collection of threads.

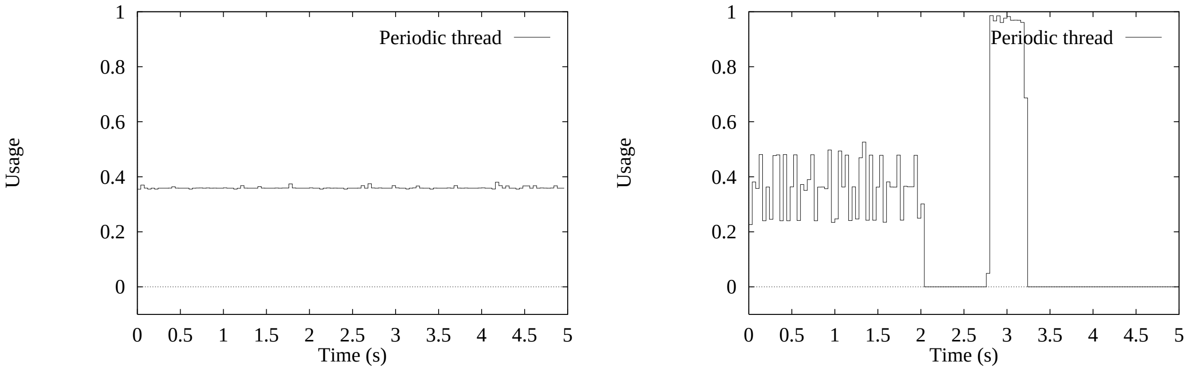
Figure 2 illustrates the difference between accounting in monolithic operating systems and the accounting we require for microkernels. Part (a) shows a program in a monolithic system which does some processing of its own and makes two system calls. The arrows represent the control paths, and each label represents the computation time required in the associated control path. In this case, the total usage of the program is $P = A + B + C + D + E$ and this time is divided into a user time of $P_U = A + C + E$ and a system time of $P_S = B + D$.

Part (b) of Figure 2 illustrates the control path for a similar program in a microkernel where the services represented by B and D are performed by separate servers. The B^* and B^{**} labels correspond to the computation time for the control necessary to issue the server requests, and likewise for D^* and D^{**}. In the monolithic system accounting scheme, the total program usage is measured by the system to be $P = A + B^* + C + D^* + E$ where the user usage is $P_U = A + C + E$ and the system usage is $P_S = B^* + D^*$. The actual service time is charged to the servers $S1$ and $S2$ and is never accounted for in the client's usage measurements. The same control path measured by our microkernel accounting scheme finds $P = A + B + B^* + B^{**} + C + D + D^* + D^{**} + E$, and this is the accurate measure of processor usage for program P.

4.3 Reserve abstraction

Our reserve abstraction serves the purpose of accurately accounting for an activity which invokes user-level services offered by the operating system. Reserves are associated with processor reservations and serve the dual purpose of organizing the reservation parameters and facilitating the enforcement of reservations by measuring usage against the reservation. Each thread has an associated reserve to which computation time is charged. One or more threads may charge computation time to the same reserve. The most useful configuration is to associate a reserve, along with its processor reservation, with each "client" thread in the system. As this client invokes operations on various servers, the IPC mechanism forwards the client's reserve to be used by the server thread to charge its computation time on behalf of the client. The result is that an accurate accounting is made of the resources consumed by the client thread throughout the system.

In addition to reserves, we allow threads to run unreserved in time-sharing mode. This makes it possible to use the unreserved capacity or unused reservations for time-sharing or background processing.



(a) No competition

(b) With competition

Figure 3: Unreserved periodic thread without and with competition

5 Performance evaluation

We have implemented processor capacity reserves in Real-Time Mach [12], and in this section, we illustrate the reservation system's behavior in various situations by showing processor usage measurements for several different task sets. The test cases ran on a Gateway2000 33MHz 486-based machine with 16MB of RAM and an Alpha Logic STAT! timer board for accurate timing. We implemented reserves on version MK78 of Real-Time Mach, and we use CMU UNIX server version UX39.

5.1 Unreserved periodic thread

The first test case, shown in Figure 3(a), shows the behavior of a single periodic thread executing with no competition for the processor. The processor usage is fairly constant over time in this case. The thread runs its computation of 16 ms during each period of 40 ms (40% utilization), and we take the processor usage measurement for each period at the end of the period. The reservation of 40% is actually slightly higher than the actual computation time for the thread; this is done to accommodate slight variations in computation time. The processor usage for a reservation period is the processor time consumed during the period divided by the length of the period, and we plot a horizontal line (at the appropriate usage level) from the beginning to the end of the period. Thus, the graph is a sequence of flat usage levels.

Figure 3(b) illustrates the behavior of the periodic thread when it competes with other threads for the processor. There are five other threads in this case, but for simplicity they are not shown in the figure. All of the threads are scheduled using the Mach time-sharing policy, including the periodic thread. The thread's processor usage is naturally quite unpredictable. We note that after the periodic thread has been prevented from executing for some time, as is the case from time 2 to time 2.7 or so, the thread attempts to catch up on all of the missed computation time. Thus, we have the surge of "catch-up" activity from time 2.7 to time 3.2. This is clearly not the behavior we desire for periodic threads.

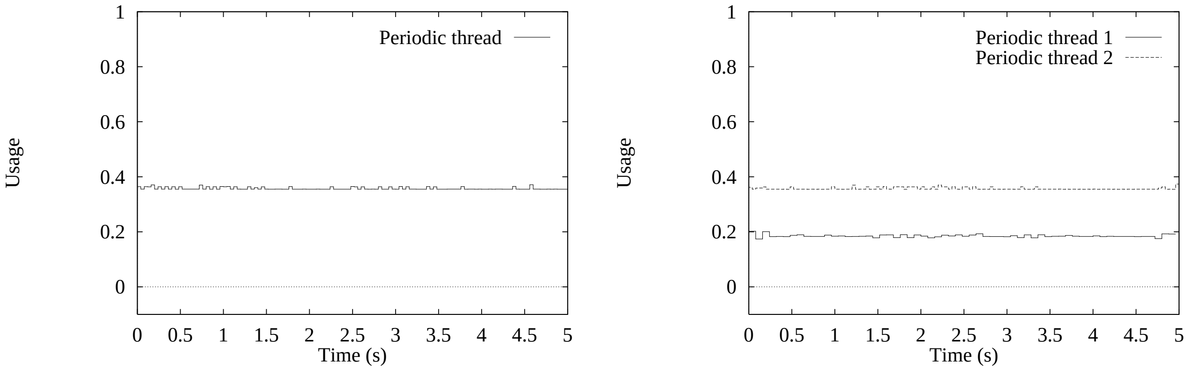
5.2 Reserved periodic threads

In the next test case, the periodic thread reserves the processor capacity it needs in advance. So in Figure 4(a), we see that the periodic thread behaves as if there were no competition, even though there are five additional unreserved threads (not shown) competing for the processor. The periodic thread executes in reserved mode, and therefore enjoys the processor capacity that it needs to run.

Figure 4(b) shows the processor usage of two reserved threads which are competing with five unreserved threads. Periodic thread 1 reserves 16 ms of computation time every 80 ms (20% utilization), and Periodic thread 2 reserves 16 ms every 40 ms (40% utilization). Both reserved threads can successfully achieve their desired computation rates.

5.3 Server invocation with no reservation

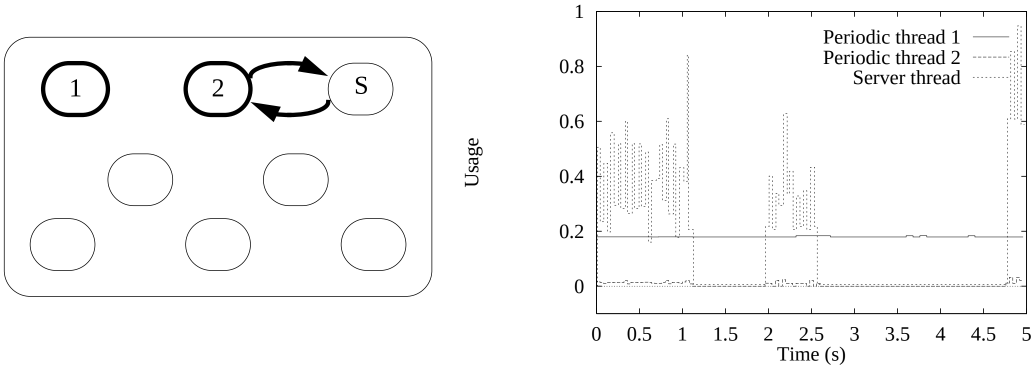
In Figure 5, we show what happens if the reservation system does not coordinate reservations during remote procedure call (RPC) types of communication. The structure of the task set is shown in Figure 5(a); reserved threads are drawn with bold lines in the figure to distinguish them from



(a) One reserved thread

(b) Two reserved threads

Figure 4: Reserved periodic thread(s) with competition from unreserved threads



(a) Task set structure

(b) Measured usage behavior

Figure 5: Uncoordinated client/server reserves

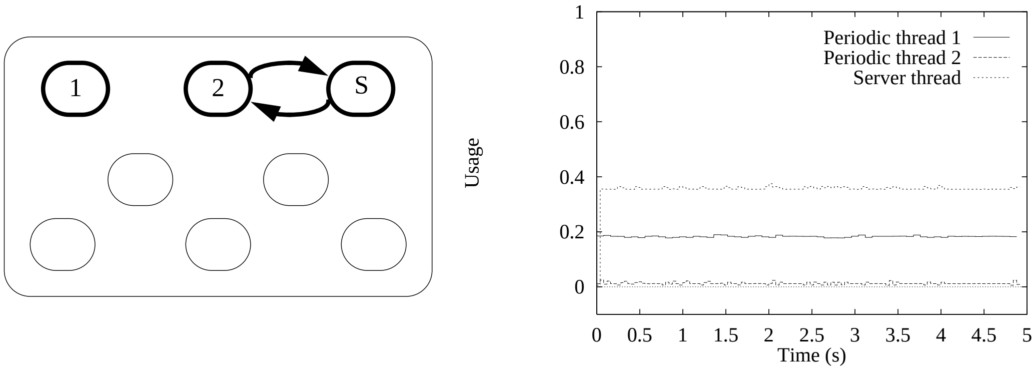
unreserved threads. Periodic thread 1 reserves 16 ms every 80 ms (20% utilization). Periodic thread 2 reserves 16 ms every 40 ms (40% utilization), and it invokes a server via an RPC to perform the actual computation. The server thread is not reserved. In addition to these three threads, there are five unreserved threads competing for the processor.

Figure 5(b) shows the usage of these threads. Periodic thread 1 executes at its reserved rate as shown by its usage line which is fairly constant at nearly 20%. The combined activity of Periodic thread 2 and its server execute unpredictably. Periodic thread 2 always has a small usage since it is merely invoking the server to do its work, and the server typically has a large usage since it is doing lots of computation on behalf of its client. The usage of these two is always in the same proportion. Since the server is unreserved, it competes with all of the other unreserved threads for processor cycles, so the combined client/server activity is essentially at the mercy of the time-sharing scheduler. Thus, the combined activity sometimes exceeds its intended reservation, and sometimes it cannot make progress at all.

5.4 Server invocation with reservation

Figure 6 illustrates the proper behavior for reserved clients and servers with integrated reserve handling. Figure 6(a) shows the same task structure as the previous example except that Periodic thread 2's reserve is passed during the server invocation, and the server runs as a reserved thread while it is performing the service. The bold lines of the server indicate that it executes as a reserved thread.

The usage for this task set is shown in Figure 6(b). As before, Periodic thread 1 executes at a fairly constant 20%



(a) Task set structure

(b) Measured usage behavior

Figure 6: Integrated client/server reserves

utilization. And now the combined activity of Periodic thread 2 and its server execute at a constant 40% combined usage. The line around 2-3% gives the usage of the client, and the line around 36% gives the usage of the server. Thus, by managing the reservations appropriately across thread interactions, we are able to preserve the isolation provided by reserves for system activities which cross thread boundaries.

6 Future work

The processor reservation mechanism described in this paper forms a basis for more sophisticated management of system resources. For example, we envision a reserved window system with a window manager that controls resource reservations for applications running under different windows. The user interface would be used to dynamically change the reservations for applications, depending on the user's preference or focus.

Toward this end, we plan to provide reservation support for system resources other that the processor, notably pages, paging activity, network buffers, and protocol processing activity. Our position is that most other resources are discrete and are therefore easier to allocate and reserve than processor cycles, so we have concentrated mainly on the harder problem. We do recognize, however, that some discrete resources, such as message buffers, are difficult to reserve in advance.

Processor capacity reserves can support reservation in distributed multimedia systems by having each reserve contain reservations for various resources around the distributed system. Then messages containing requests for remote service will contain these "sub-reserves" which can be used to charge the remote service. Another aspect of reservation in distributed systems concerns the reservation of protocol processing on each of the hosts. We are planning to use the notion of a "suspense reserve" to charge the protocol processing time associated with bringing packets from the network device to the end point of the communication session. When the destination program of the packet is determined, the usage associated with that packet can be charged to the receiving program.

7 Related work

Many researchers consider resource reservation desirable if not absolutely necessary for continuous media operating systems. Herrtwich [5] gives an argument for resource reservation and careful scheduling in these systems. Anderson *et al.* [2] give additional arguments for introducing more sophisticated timing and scheduling features into continuous media operating systems, and their DASH system design supports reservation and uses earliest deadline scheduling for real-time traffic. They use earliest deadline because it is optimal in the sense that if a collection of tasks with deadlines can be scheduled by any algorithm, it can be scheduled by the earliest deadline algorithm. They do not explicitly describe exactly how processor reservation is integrated with network reservation or why earliest deadline is well suited to reservation strategies. We demonstrate why rate monotonic scheduling and earliest deadline scheduling are suitable for processor reservation, and we compare these two approaches. In addition, we explain how reservations can be enforced and how non-real-time programs can be integrated with real-time programs in the scheduling framework.

Jeffay *et al.* implemented a programming model specifically designed for guaranteed real-time scheduling [6]. This involves a restricted programming model and an off-line analysis. In contrast, our approach is to provide optimistic predictability using a more traditional programming model and a very fast on-line analysis.

Other work, particularly work related to network communication, relies on reservation in network nodes (gateways and hosts) to support bandwidth reservation and rate-based protocols [1, 3]. Our work provides a basis for software implementation of these kinds of protocols in the context of general purpose operating systems.

8 Conclusion

In this paper, we have motivated the design of a processor reservation strategy for supporting continuous media applications. Our scheduling framework, based on computation rates expressed as computation time per duration, provides an effective way to specify processor requirements. Two scheduling algorithms with slightly different properties are suitable for implementing reservation in this framework. The design addresses practical issues in implementation of the scheduling framework, and it depends on a novel reserve abstraction for accurate computation time measurement and reservation enforcement. This accounting mechanism is well suited to the microkernel architecture; it tracks processor time used by individual threads which call on user-level servers to perform system services.

Our prototype implementation using Real-Time Mach 3.0 demonstrates the feasibility of our approach and shows that applications can achieve predictable real-time performance using our reservation mechanism.

Acknowledgements

The authors would like to express their appreciation to the following people for their comments and suggestions: Brian Bershad, Ragunathan Rajkumar, Jim Zelenka, Raj Vaswani, John Zahorjan, and the members of the ART group and Mach group at CMU.

References

[1] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet. Technical Report TR-90-006, International Computer Science Institute, February 1990.

[2] D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for Continuous Media in the DASH System. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 54–61, May 1990.

[3] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communication*, 8(3):368–379, April 1990.

[4] D. Golub, R. W. Dean, A. Forin, and R. F. Rashid. Unix as an Application Program. In *Proceedings of Summer 1990 USENIX Conference*, June 1990.

[5] R. G. Herrtwich. The Role of Performance, Scheduling, and Resource Reservation in Multimedia Systems. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science, pages 279–284. Springer-Verlag, 1991.

[6] K. Jeffay, D. L. Stone, and F. D. Smith. Kernel Support for Live Digital Audio and Video. *Computer Communications (UK)*, 15(6):388–395, July-August 1992.

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

[8] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[9] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, 20(1):46–61, 1973.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[11] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *The Journal of Real-Time Systems*, 1(1):27–60, June 1989.

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