Processor Capacity Reserves
for
Multimedia Operating Systems

Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda

May 1993

CMU-CS-93-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Multimedia applications have timing requirements that cannot generally be satisfied using time-sharing scheduling algorithms and system structures. To effectively support these types of programs, operating systems must support processor capacity reservation. A capacity reservation and enforcement mechanism isolates programs from the timing and execution characteristics of other programs in the same way that a memory protection system isolates programs from memory access by other programs. In this paper, we characterize the timing requirements and processor capacity reservation requirements for multimedia applications, we describe a scheduling framework to support reservation and admission control, and we introduce a novel *reserve* abstraction, specifically designed for the microkernel architecture, for controlling processor usage.

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.

Keywords: multimedia, operating systems, reservation, admission control, scheduling, performance measurement

1. Introduction

Many personal workstations provide audio and video devices for multimedia applications. To support these applications, the operating system must be responsive to the strenuous timing constraints of real-time digital audio and video data streams (*continuous media* [2]). Unfortunately, the time-sharing scheduling policy found in most workstation operating systems is an historical artifact from a time when many users with interactive and batch computing requirements shared a single processor. A personal workstation, which is typically dedicated to a single user, should schedule processes based on the user's focus or based on the preferences of the user. This user-centric approach to scheduling is even more important if the personal workstation supports real-time multimedia computing.

We have designed a *processor capacity reservation* mechanism which allows the user to control the allocation of processor cycles among programs. Our design treats continuous media applications and non-time-constrained applications in an integrated way. Reservations are requested at the time each application is created, and once a reservation has been granted by the scheduler, the program is assured of the availability of processor capacity (henceforth just "capacity"). We use a kernel abstraction called a *reserve* which keeps track of the reservation and measures the processor usage of each program. Microkernel operating systems employ separately scheduled servers to provide various system services [4], and consequently, the true processor usage of a program includes the processor usage of the servers invoked by that program. Unfortunately, some microkernels measure user time and system time for each process, thus the usage statistics do not reflect the true computation time of programs. Our reserve mechanism accurately accounts for computation time done in one program on behalf of another and charges that time against the appropriate reserve.

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 from interrupt processing. Additionally, it can be difficult to schedule two or more continuous media applications using only a priority assignment, even if their total average utilization does not saturate the processor. If two programs happen to require the processor at the same time in order to meet their timing constraints, the contention may cause both of them to miss their deadlines. In such a case where executing two programs is infeasible, the system should recognize this before attempting to schedule both programs. Having programs make reservations for their computational requirements allows the scheduler to decide 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.

The possibility of processor overload presents additional problems for continuous media 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 may 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.

In discrete resource allocation (such as memory allocation), a request for more of a resource than is currently available fails, and the program that made the request must either do without the resource or request less of the resource. In processor scheduling, a program that requires more of the processor than is available takes a little bit of the processor from each of the other programs. As long as the established programs can afford to give up the extra processing time, this policy works, but continuous media applications have certain minimum resource requirements below which it does not make sense to run the program. The solution is a capacity reservation strategy which can reserve processor time, measure usage, and verify that the reservations are not exceeded. Programs which require more processing power than is available can be accommodated only after they reduce their computational requirements. Processor capacity reservation should be more like discrete resource allocation.

In the rest of the paper, we describe our reservation strategy and discuss related work. We give the requirements for a reservation strategy in Section 2, and in Section 3 we explore two approaches to scheduling with reservations, examining the advantages and disadvantages of each. Section 4 discusses implementation issues, and Section 5 describes the new reserve abstraction and discusses usage measurement and reservation enforcement. In Section 6, we discuss related work, and we make a few concluding observations in Section 7.

2. Requirements for a reservation strategy

The problems described in the previous section motivate the design of a processor capacity reservation strategy for continuous media operating systems. The reservation strategy must

1. provide some means for application programs to specify their processor requirements,
2. accurately measure the computation time consumed by each program,
3. evaluate the processor requirement of new programs decide whether to admit them or not, and
4. be able to control the execution of programs which attempt to overrun their capacity reservation.

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. Percentage of the CPU provides a straightforward measure to describe the processor requirement of both of these kinds of programs. CPU percentage is the CPU time required by a program during an interval divided by the real time of the interval, and the CPU percentage consumed by a program over time defines its rate of progress. Periodic programs (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 time, 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 the program. Figure 1(a) illustrates the computational requirement of a periodic program τ_P and shows how the period may be split in half, along with the computation time, to reduce the maximum delay for the smaller computations. A non-periodic program appears in Figure 1(b) which shows how the long computation can be executed in periodic bursts. With this kind of framework, we can use scheduling algorithms which schedule programs according to their allocated rate.

The second requirement, that the scheduler accurately measure the computation time consumed by each program, 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 [12], but this information is not very precise over short intervals. Furthermore, the execution behavior of the monitored program must be independent of the sampling period. A more precise mechanism would measure durations between context switches and would account for interrupt processing time and other "system overhead."

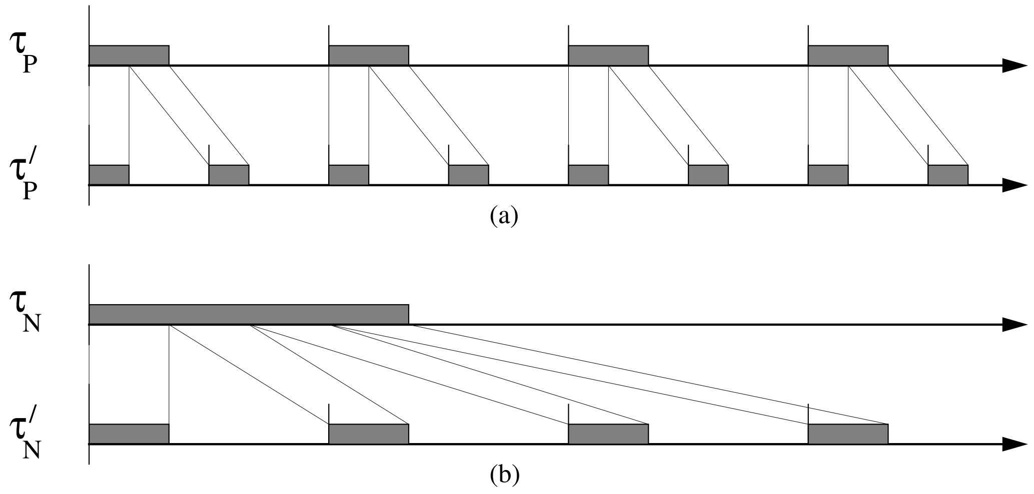


Figure 1: Periodic Framework

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 [12]. For monolithic operating systems, this approach is sufficient, but for microkernel systems where operating system services are offered by different user-level servers, the usage statistics of an activity cannot be found in the usage statistics of a single process. An activity may request services of several different operating system servers such as a name server, a file server, or an I/O server. To maintain an accurate picture of the activity's capacity consumption, the cost of these services must be charged to the activity itself rather than to the individual servers. Thus capacity reservations must be associated with reserves which are then used to charge processing time at different processes. Ideally, reserves would even be associated with incoming network packets so that the receiver of the packets could be charged for their protocol processing. Otherwise an activity which communicates over the network could exceed its capacity reservation.

Figure 2 illustrates how to account for computation time in a server. The client *C* starts with no charges against its reserve as indicated by the empty oval in the leftmost frame of the figure; the server *S* has no reserve of its own. When the client invokes the server, the charge for the invocation, as indicated by the shaded part of the oval in the next frame, is applied to the reserve, and the reserve is passed to the server. The server uses the reserve to charge the computation time required for service, and the server passes the invocation result and the reserve back to the client. The cost of the invocation is applied in full to the client's reserve.

The third 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. Figure 3 illustrates the flavor of the two reservation strategies. Part (a) shows how the processor can be divided into arbitrary size slices where the sum of the slices is 100%, and part (b) represents a method where reservations must "fit together" in some way which may not allow 100% of the processor to be reserved.

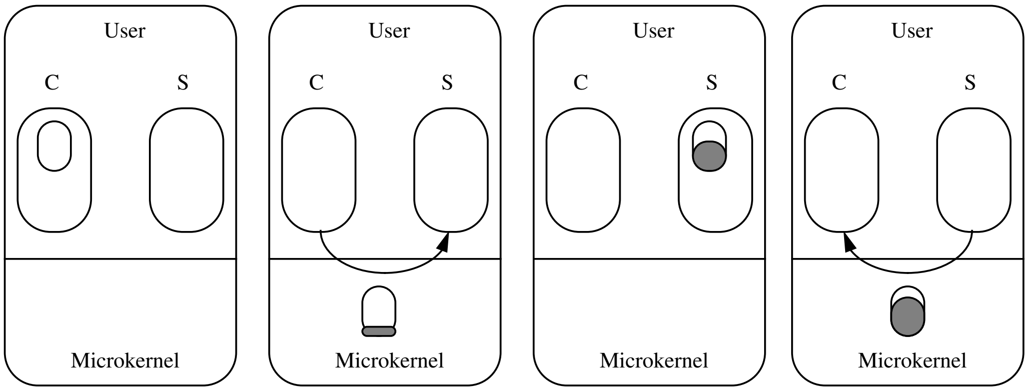


Figure 2: Simple Microkernel Accounting

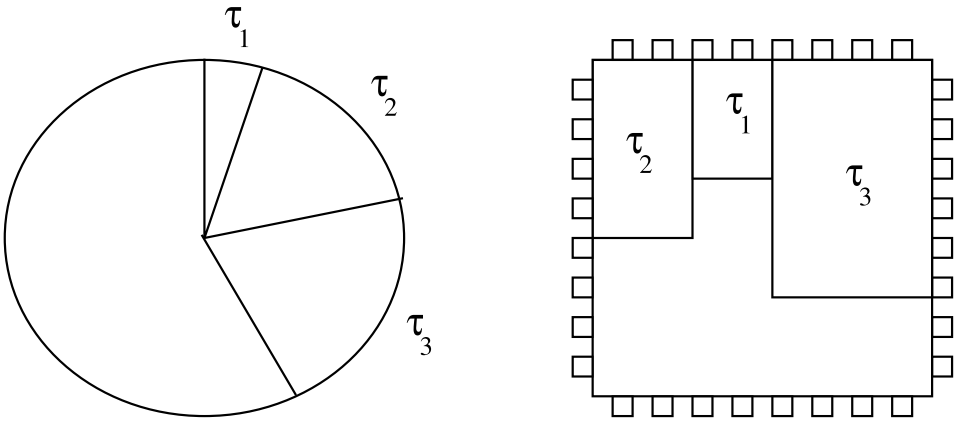


Figure 3: Processor Capacity Reservation

The fourth requirement, that the scheduler be able to control the execution of programs which attempt to overrun their capacity reservation, means that the system must accurately measure the processor usage of each program and that a mechanism must exist to notify the scheduler when a program exceeds its computation time for the current period. Whenever the scheduler dispatches a thread, it computes the maximum duration of time the thread could execute based on its allocation and its usage so far, and it sets a timer to expire after this duration.

3. Approaches to reservation

A scheduling framework based on 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. We shall compare the advantages and disadvantages of these two approaches.

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 [14] does just that. Under this regime, programs are ordered from highest rate to lowest rate and then priorities are assigned with the highest priority assigned to the highest rate program and the lowest priority assigned to the lowest rate program.

The rate monotonic scheduling algorithm was analyzed under simplifying conditions, and the results of this analysis provide a basis for processor reservation strategies. Liu and Layland [14] 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.

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. They 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.

A reservation strategy 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 its rate and the total previous reservation is less than .69. Such a policy would leave a lot of "idle" 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.* [13] 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. 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 "idle" computation time is only 10-12% rather than 31%.

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 [14] 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 reservation strategy 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 1, and, if so, the program is admitted. If not, the reservation cannot be granted.

3.3. Overload control

The two policies perform differently under overload conditions. A good processor reservation implementation should not allow long-term overload, but because of unpredictable synchronization and communication, some degree of transient overload is likely to occur. Rate monotonic scheduling, as a fixed priority policy, maintains relatively stable performance under overload. In particular, the programs with higher priorities will execute successfully at the expense of the programs with lower priorities which may miss some of their deadlines. As long as the higher priority programs are the "most important," RM should provide good performance under overload. If a program is identified as "important" but has a low priority due to loose time constraints, the period may be artificially shortened (e.g. halved) and the computation time broken up accordingly so that it gets a higher priority.

The ED algorithm fails under overload. A transient overload will adversely affect all programs, even the "important" ones, as the algorithm attempts to service the programs in deadline order without regard to whether the service time could be better spent elsewhere. ED tries to give each program its fair share, but in the overload, none of the programs get the service they were guaranteed, and all of them suffer. This is exactly the property of time-sharing algorithms that we are trying to avoid in introducing the reservation policy. Rather than try to partially satisfy the requirements of all programs, we want to fully satisfy the requirements of a subset of programs.

3.4. Discussion

We note that both of these analyses depend on restrictive theoretical models which cannot be achieved in practice. Real programs may not be independent, and they are generally not completely preemptive, especially if the kernel is non-preemptive. When preemption is allowed, the context switch time has a cost. Thus, systems will not be able to adhere to the reservations as strictly as the models predict, but with accurate execution monitoring, the scheduler can detect transient reservation violations and take corrective action on a fine-grain time scale. Some programming rules, such as "use critical regions sparingly and make them as short as possible," also help to minimize timing problems.

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 strategy and due to the effects of critical regions and other synchronization and communication among programs. So in terms of the reservation bound, the two scheduling algorithms may not be as disparate as they first appear.

ED and RM differ in a few details of their implementations. In particular, fixed priority scheduling algorithms usually use a small integer to represent priority. In dynamic priority scheduling, a timestamp is usually involved in priority decisions; for earliest deadline scheduling, the deadline time is used in making scheduling decisions. The timestamp takes more space and more time for comparison, and in a distributed system, there may not be an accurate global clock to support timestamps.

4. Implementation issues

The periodic scheduling framework described in Sections 2 and 3 and the rate monotonic and earliest deadline scheduling algorithms provide a theoretical basis for a processor reservation strategy, but we must address many practical details in the actual implementation of processor reservation. Estimating the computation time of a program is one area of difficulty, and program start-up is another problem area.

4.1. Specifying computation time

Computation time plays a major role in the scheduling framework; an accurate estimate of computation is essential for an accurate measure of computation rate in periodic programs. The other parameters of the framework are usually easier to determine. For example, period for continuous media applications is typically defined by the data type and by the resolution, although the issue of delay arises in choosing a period.

Unfortunately, the computation time is unlike the period in that the computation time comes out of the code rather than being designed into the application. As such, the programmer has less control of the exact length of the computation time except by complicated coding tricks. On the other hand, the programmer could just write the code and then measure it to try to determine its timing characteristics, perhaps in terms of a statistical distribution or a worst case measurement. Rather than complicate the programmers job with all of these details, we propose to use an estimate of the computation time for the purposes of processor reservation, and then depend on the system to make sure that the program does not overrun the estimate. This is in contrast to the approach used in many real-time systems where worst case timing measurements are required to statically determine the scheduling properties of the system. Our approach puts the job of computation time measurement and control on the system instead of on the programmer, and then as the programmer gains more experience with the timing characteristics of his program, he will be in a position to make better estimates. One can develop a program and iteratively arrive at a good estimate without interfering with previously established programs with processor reservations.

The philosophy of this approach is similar to that of the early virtual memory systems where the operating system first provided memory protection and automatic memory management. These features made it possible to write programs which were insulated from other programs, and they obviated the need for *ad hoc* memory management schemes for each new application. We seek to protect programs from the timing effects of other programs and to abstract away some of the details of timing in much the same way as automatic memory systems protect and manage memory.

In contrast to periodic programs, we note that non-periodic programs have no predetermined units of computation time, and consequently, there is no need to find computation time estimates. The computation time can be broken up by the enforcement mechanism as necessary to maintain the desired computation rate of the program.

4.2. Specifying target rate

The target rate is the computation rate at which the program is intended to execute. The specification of rate, as described in Section 3, consists of two of the three parameters: rate, computation time, and period. Short-term target rate violations may occur due, for example, to critical regions which cannot be preempted and which therefore cause a computation to overrun its processor allocation for a period. Critical regions which are large with respect to the computation time will tend to exacerbate this problem, but as long as critical region length is minimized, the target rate can be accurately enforced.

Computing rates are generally not set in stone. Continuous media applications are typically designed to run at various rates. A video application may process 30 frames/sec or just 10 frames/sec; an audio application may support 44k samples/sec or just 8k samples/sec. Even if the processing rate of a program is fixed, the decomposition of rate into computation time and period depends on the amount of delay the that can be tolerated.

In Section 1 we divided continuous media programs into two types: periodic and non-periodic. We further subdivide periodic programs as follows.

1. Fixed basic computation time – These programs have a basic computation, such as "process a fixed-size buffer of audio samples" or "process a video frame" which are atomic and cannot easily be divided. The total computation time for a period can be a multiple of this basic computation time.

2. Tunable computation time – These programs have a basic computation, the duration of which can

be tuned. By "tuned" we mean that the duration can be adjusted, but adjustments should be rare and the duration should be constant between adjustments. Examples might be "process a variable-size buffer of audio samples" or "process a video frame of adjustable resolution."

These two kinds of periodic program structure give us two possibilities for automatic adjustment by the scheduler: fixed-period rate adjustment and fixed-rate period adjustment. We will find that both automatic adjustment techniques can be used with non-periodic programs since they have a more flexible structure than periodic programs.

Fixed-period rate adjustment implies that the period remains constant while the rate is changed, and this requires a change in the processor reservation. If the rate is increased, a new reservation is required at the higher level. If the rate is decreased, the reservation should also decrease to make the additional capacity available to other applications. The scheduler cannot arbitrarily change the rate, however, since the periodic computation of a program may be indivisible, making reduction of the rate impossible. Also, when increasing the rate, it may not be possible to set the computation time at an arbitrary value. Computing one and a half video frames during each period may be difficult if the code structure is set up for one frame per period or if the timing constraints dictate one frame per period. For the adjustment to be automatic, periodic programs must have a basic computation which is made known to the scheduler; the scheduler can then set the computation time and period to a multiple of that basic computation and the associated period.

Non-periodic programs, since they do not have this additional periodic structure, are handled by arbitrarily setting the computation time and period to yield the intended rate. The program can be interrupted at will, and this makes rate adjustment much easier than in the case of periodic programs.

Each program, periodic or non-periodic, specifies a range of computation rates that suit the application as well as a target rate which is the preferred value. The scheduler attempts to execute the program at the target rate and then decreases the target or increases the rate as the load on the system allows. In any case, a program which has begun execution by successfully reserving at least its minimum rate will always be guaranteed that the reservation will not fall below the minimum.

Fixed-rate period adjustment, as the name implies, involves holding the rate constant while changing the period to achieve better performance. When the rate is fixed, a change in computation time must accompany a change in period. There are two reasons for changing the period of a program. One reason to change the period is to make scheduling easier. For example, suppose we double the period and double the computation time of a program, leaving the rate the same. We must complete twice as much processing in each double length period, but we are free to do that processing at any time during the double length period. This is in contrast to the case of the single length period where the computation must be split over two of the single length periods. Increasing (or decreasing) the period can also bring the period into a harmonic relationship with the other periods in the program mix; harmonic periods are generally easier to schedule than non-harmonic periods (see Section 3). One effect of doubling the period in this way is that delay for the computation may be increased. For example, the double length computation may not execute until the very end of the double length period which means that the first part of the computation finishes much later than in the single length case.

Another reason to change the period is related to the delay. The scheduler can achieve lower delay for subparts of the computation time by reducing the period and splitting up the computation time accordingly. For example, suppose we have a program which generates and transmits two audio packets during its computation time; it may be that both packets are transmitted at the very end of the period, and the delay is on the order of the length of the period. We can cut the maximum delay for each packet roughly in half by halving the computation time and halving the period. Then a packet leaves the program after half as much time. Neither of these cases of fixed-rate period adjustment involves a change in the processor reservation. Both involve adjustments of timing parameters which are local to the program.

Specifying a range of acceptable delay enables the scheduler to perform automatic fixed-rate period adjustment. The range includes a minimum delay and a maximum acceptable delay along with a target delay which serves as the starting point for the scheduler. With this information, the scheduler can increase the period to ease scheduling as long as the delay does not become too large. The scheduler can also decrease the period to get lower latency when the load is light.

We note that a more complicated interface between the application and the system would allow for more detailed negotiation of program structure and timing constraints, but we defer the detailed design and implementation of such an interface.

4.3. Program start-up

The initial start-up phase of continuous media programs pose special problems for the reservation strategy. After a target rate has been reserved and the computation time and period have been fixed, the program begins execution. But since the computation time is only an estimate, there is a chance that the program requires more time than has been reserved. In this case, the enforcement mechanism will limit the program to the reserved rate, and the program will not be able to meet all of its deadlines. This is a problem only during the initial development phase of the application. After the programmer has more experience with the timing constraints of the application and produces better estimates, the start-up phase becomes smooth.

Another problem during start-up occurs when the system is loaded and reservations become tight. If the target rate cannot be supported with the other reservations, the scheduler must scale down the rate, possibly to the minimum. The problem is finding a rate that satisfies the constraints imposed by the basic computation time and that is larger than the minimum rate. Thus, the overhead at creation time may be larger with heavier load.

5. 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.

5.1. Measurement and control accuracy

Our enforcement mechanism monitors processor usage by measuring the duration of 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 programs in time-sharing mode, but if there is unreserved processor time available, programs in time-sharing mode 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 take this action 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.

Other operating systems [12] use sampling methods to measure the processor time consumed by different programs. Sampling periods are typically on the order of 10 ms [12], so the accuracy of the measurement is low. The coarse grain of such built-in performance measurement mechanisms yields coarse grain control of the computation, and represents a "best effort" kind of system design. The ubiquitous multi-level feedback queue found in many time-sharing systems uses coarse grain feedback to adjust priorities [12]. The Fair Share Scheduler (FSS) [7] extends this model with reservations in the same spirit as ours except that FSS is a best-effort scheduler which does not accurately enforce its reservations. FSS reacts dynamically to conditions and uses simple heuristics to try to balance the processor usage over the long term, but it allows unstable short-term behavior which we desire to avoid in our reservation and enforcement scheme.

Although our scheme is intended to be accurate, we do not require that program usage characteristics be consistent over different executions. This is in contrast to the approach taken in Cal [11] where many design decisions were influenced by the requirement that program execution should cost the same every time. This approach, as noted by Lampson and Sturgis, leads to much larger overhead charges. For example, a program that requires a particular text page cannot use a copy that was paged in for another program; it must pay the cost of a page fault itself. Otherwise, its total cost would be different for different executions depending on how many pages were already available in memory.

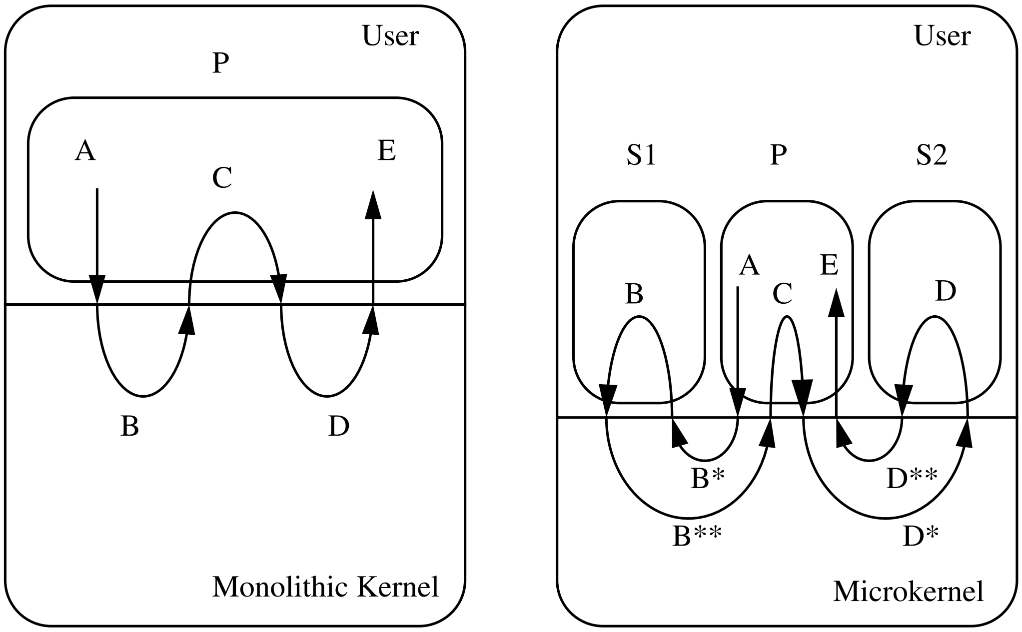
5.2. Microkernel accounting

In many operating systems, the processor usage for each process is recorded in a per process logical clock, and the logical clock is usually divided between user time and system time. 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 4 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 4 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.

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 their 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



(a) (b)

Figure 4: Microkernel Accounting

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 throughout the system by the client thread.

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.

6. Related work and future directions

Many researchers consider resource reservation desirable if not absolutely necessary for continuous media operating systems. Herrtwich [8] gives an argument for resource reservation and careful scheduling in these systems. Rangan and Vin [16] describe a file system with an admission control algorithm that is designed to support the real-time constraints of requests. 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. The authors 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 how the reservation scheme works 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.

Govindan and Anderson [5] propose deadline/workahead scheduling and indicate that this method can be used for reservation systems or for systems without reservations. Their reservation method is based on earliest deadline scheduling as in the DASH system. 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.

"Fair share" schedulers [7, 9, 18] ensure that users who pay more for the compute time actually get better service than others who pay less. The scheduler reserves capacity for each user or group of users and uses usage measurements to try to match usage with reservation in the long term. This work is similar in spirit to ours, but their reservations are statically defined by system administrators, and none of these schedulers provides accurate short-term reservation enforcement.

Lampson proposed a fixed time scheduling approach [10] for scheduling periodic processes and conjectured that a suitable analysis would predict the performance characteristics of processes. Our strategy uses Liu and Layland's results for periodic processes along with a reservation enforcement mechanism to implement predictable service for periodic processes.

Performance measurement and accounting techniques have a long history in computer systems [15]. Most systems use sampling techniques or complete traces. Our approach is to use a short-term trace where we measure the duration of processor time each program gets, but we summarize this information in an accumulator for each program rather than keep the detailed trace information around. We get the information we need without retaining the detailed log, and this gives us a better measure than sampling techniques which depend on the law of large numbers for accuracy. Self-tuning mechanisms have been proposed and implemented for time-sharing systems. The idea is to keep load under control by rejecting new logins when the load is above a certain level [17].

Harter and Geihs describe an accounting strategy for distributed systems in which accounts are known throughout the system, and clients can charge services at various nodes to their accounts [6]. This is similar to our microkernel accounting strategy in that our reserves can be passed around the system. Our reservation strategy requires additional accuracy and control to be incorporated into our accounting scheme.

Our accounting mechanism extends easily to the distributed environment where servers can charge their service time to reserves associated with remote clients, although distributed processor capacity reservation requires a protocol to reserve resources at various nodes in the network. We plan to explore these issues in future work, after completion of our prototype uniprocessor implementation.

7. Conclusion

In this paper, we have motivated the design of a processor reservation strategy in the context of a continuous media operating system. Our scheduling framework, based on computation rates expressed as computation time per duration of time, 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.

Acknowledgements

The authors would like to express their appreciation to the following people for their comments and suggestions: Brian Bershad, Ragunathan Rajkumar, 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. Govindan and D. P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 68–80, October 1991.

[6] G. Harter and K. Geihs. An Accounting Service for Heterogeneous Distributed Environments. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[7] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1858, October 1984.

[8] 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.

[9] J. Kay and P. Lauder. A Fair Share Scheduler. *CACM*, 31(1):44–55, January 1988.

[10] B. W. Lampson. A Scheduling Philosophy for Multiprocessing Systems. *CACM*, 11(5):347–360, May 1968.

[11] B. W. Lampson and H. E. Sturgis. Reflections on an Operating System Design. *CACM*, 19(5):251–265, May 1976.

[12] 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.

[13] 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.

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

[15] G. J. Nutt. Tutorial: Computer System Monitors. *IEEE Computer*, 8(11):51–61, November 1975.

[16] P. V. Rangan and H. M. Vin. Designing File Systems for Digital Video and Audio. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 81–94, October 1991.

[17] M. V. Wilkes. Automatic Load Adjustment in Time-Sharing Systems. In *ACM SIGOPS Workshop on System Performance Evaluation*, pages 308–320, April 1971.

[18] C. M. Woodside. Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers. *IEEE Transaction on Software Engineering*, SE-12(10):1041–1048, October 1986.