Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems

Raj Rajkumar, Kanaka Juvva, Anastasio Molano and Shuichi Oikawa
Real-Time and Multimedia Laboratory[1]
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{raj+, kjuvva, amolano, shui}@cs.cmu.edu

Abstract

We consider the problem of OS resource management for real-time and multimedia systems where multiple activities with different timing constraints must be scheduled concurrently. Time on a particular resource is shared among its users and must be globally managed in real-time and multimedia systems. A resource kernel is meant for use in such systems and is defined to be one which provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

We identify the specific goals of a resource kernel: applications must be able to explicitly state their timeliness requirements; the kernel must enforce maximum resource usage by applications; the kernel must support high utilization of system resources; and an application must be able to access different system resources simultaneously. Since the same application consumes a different amount of time on different platforms, the resource kernel must allow such resource consumption times to be portable across platforms, and to be automatically calibrated. Our resource management scheme is based on resource reservation [25] and satisfies these goals. The scheme is not only simple but captures a wide range of solutions developed by the real-time systems community over several years.

One potentially serious problem that any resource management scheme must address is that of allowing access to multiple resources simultaneously and in timely fashion, a problem which is known to be NP-complete [5]. We show that this problem of simultaneous access to multiple resources can be practically addressed by resource decoupling and resolving critical resource dependencies immediately. Finally, we demonstrate our resource kernel's functionality and flexibility in the context of multimedia applications which need processor cycles and/or disk bandwidth.

1. Motivation for Resource Kernels

Example real-time systems include aircraft fighters such as F-22 and the Joint Strike fighter [19], beverage bottling plants, autonomous vehicles, live monitoring systems, etc. These systems are typically built using timeline based approaches, production/consumption rates [9] or priority-based schemes, where the resource demands are mapped to specific time slots or priority levels, often in ad hoc fashion. This mapping of resources to currently available scheduling mechanisms introduces many problems. Assumptions go undocumented, and violations go undetected with the end result that the system can become fragile and fail in unexpected ways. We argue for a resource-centric approach where the scheduling policies are completed subsumed by the kernel, and applications need only specify their resource and timing requirements. The kernel will then make internal scheduling decisions such that these requirements are guaranteed to be satisfied.

Various timing constraints also arise in desktop and networked multimedia systems. Multi-party video conferencing, mute but live news windows, recording of live video/audio feeds, playback of local audio/video streams to remote participants etc. can go on concurrently with normal computing activities such as compilation, editing and browsing. A range of implicit timeliness constraints need to be satisfied in this scenario. For example, audio has stringent jitter requirements, and video has high bandwidth requirements [8]. Disk accesses for compilation should take lower precedence over disk accesses for recording a live telecast.

Two points argue in favor of resource-centric kernels we call "resource kernels":

- Firstly, operating system kernels (including microkernels) are intended to manage resources such that application programs can assume in practice that system resources are made available to them as they need them. In real-time systems, system resources such as the disk, the network, communication buffers, the protocol stack and most obviously the processor are shared. If one applica-

[1]This research was supported by the Defense Advanced Research Projects Agency in part under agreement E30602-97-2-0287 and in part under agreement F30602-96-1-0160. Mr. Molano was funded by a research grant from the Community of Madrid and by the National R&D Program of Spain under contracts TIC96-0982 and TIC97-0438.

tion is using a large portion of the system resources, then it implies that other applications get a less portion of the system resources and consequently can take longer to execute. In other words, their timing behavior is adversely affected. Letting kernels take explicit control over resource usage is therefore a logical thing to do to prevent such unexpected side effects.

- Secondly, our resource model captures the essential requirements of many resource management policies in a simple, efficient yet general form. The implementation of the model can actually be done using any one of many popular resource management schemes (both classical and recent) without exposing the actual underlying resource management scheme chosen. User-level schemes can be used to dynamically downgrade (upgrade) application quality when new (current) resource demands arrive (leave). This feature of the resource model leads to minimal changes from existing infrastructure while retaining flexibility and offering many benefits.

Other alternatives to resource kernels include user-centric and application-centric kernels:

- A user-centric kernel can emphasize multi-user capabilities, and also track and facilitate the needs of specific users. Unix in general and Unix filesystems in particular can be viewed as providing such support. At the same time, Unix attempted to present and manipulate all system entities as files. In resource kernels, we adopt a similar approach and attempt to present all system resources using a uniform model for guaranteed access. Our implementation of the resource kernel is orthogonal to user-centricity, but tighter integration between the two may be possible. Currently, specific user-level requirements must be translated by intermediate layers into resource demands at the resource kernel interface.

- Application-centric kernels are typically custom executives with built-in support for the applications they are intended to serve. As an example, kernels used in telecommunication switches are application-centric and deal explicitly with the high-performance, upgrading, availability, billing and auditing requirements of telecommunication paths. Conceptually, the notions of resource kernels to guarantee timely access to resources can be applied to such kernels as well. For example, consider a postscript printer. It has an executive running a postscript interpreter and control of the physical printing operations. Precisely timed control and concurrency management of downloading new print tasks in such executives can also benefit from the support available in resource kernels.

1.1. Comparison with Related Work

A wealth of resource management schemes and scheduling algorithms exist from which one can draw. Our resource management work builds on and significantly extends established real-time scheduling theory and derived processor reservation work reported in [25]. The work in [25] did not deal with the management of multiple resource types, concurrent accesses to different resources, explicit timeliness control, feedback about resource usage, behavioral control on resource overruns, management of interactions between resource users, and considerations of portability, compatibility and automation. In brief, our resource management scheme supports the abstractions behind real-time priority-based scheduling for periodic activities, and service schemes for aperiodics in that framework.

Some of our goals (such as resource centricity and portability) are similar to those of Microsoft Research's Rialto kernel among others. The reservation model also has its counterparts in network reservation protocols as used in ATM and RSVP. However, the operating system problem seems more complex in one sense since inherently different resource types must be dealt with, while networks essentially deal with one type (namely network packets). In another sense, network reservations must be homogeneous, scalable and efficient, making its realization harder in a different sense.

Despite its origins in real-time scheduling theory, we expect our resource management model to be compatible with resource management schemes with their origins in networks such as proportional fair-sharing schemes such as PGPS, WF^2Q, virtual clocks and lottery scheduling. The notion of fairness has for long been deemed to be anti-thetic to real-time systems and the management of timeliness [38]. Weighted allocation schemes such as proportional fair-sharing, however, can still be applied to the real-time model. This can be done by dynamically recomputing the weights so as *not* to be proportional or fair, but instead to obtain a fixed share of a resource when new requests arrive or current requests complete. Our scheme employs a different period for each real-time activity, and guarantees a "share" of that period to the activity. As a result, the dynamic recomputing of weights in a proportional fair-share scheme can be viewed as a special case of our model as having a single (small) fixed period for all resource allocations. The primary difference that we see is that our work advances system capabilities to include non-traditional resources such as disk bandwidth that can be used in conjunction with processor scheduling.

Finally, Blazewicz et al. [5] have shown that the problem of scheduling activities which need multiple resources simultaneously and have timeliness constraints is NP-complete. In our work, we therefore strive for practical and acceptable alternatives which can guarantee access to different resource types.

1.2. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we present the goals to be satisfied in designing a resource kernel, and based on well-established principles of real-time resource management, defines a resource reservation model and its parameters. In Section 3, we describe the implementation of our resource reservation model in the context of processor scheduling, and evaluate it. In Section 4, we detail the implementation of the resource reservation model in the context of disk scheduling, and evaluate those schemes. In Section 5, we address other issues that arise in the context of using the resource kernel in practice including calibrating an application's resource demands automatically and in portable ways. Finally, in Section 6, we conclude with some remarks outlining our research contributions and future work.

2. Designing a Resource Kernel

The challenges for a resource kernel are many. We characterize these challenges below as a set of goals that resource-centric kernels should aim to satisfy.

2.1. Design Goals of a Resource Kernel

G1. Timeliness of resource usage. An application using the resource kernel must be able to request specific resource demands from the kernel. If granted, the requested amount of resources must be guaranteed to be available in timely fashion to the application. An application with existing resource grants must also be able to dynamically upgrade or downgrade its resource usage (for adaptation and graceful degradation purposes). This implies that the kernel must support an admission control policy for resource demands. Conventional real-time operating systems do not provide any such admission control mechanisms, even though an equivalent feature (without enforcement capabilities) could be built at user level.

G2. Efficient resource utilization. The resource kernel must utilize system resources efficiently. For example, a trivial and unacceptable way to satisfy G1 would be to deny all requests for guaranteed resource access. In other words, if sufficient system resources are available, the kernel must allocate those resources to a requesting application. This goal implies that the admission control policy used by the resource kernel have provably good properties. Such proof may be analytical or empirical but our version of the resource kernel provides analytically proven properties. It must be noted that this goal is subordinated to G1, in that guaranteed resource access is the primary goal, and efforts to improve efficiency and throughput cannot happen at the expense of the guarantees.[2] Traditional real-time operating systems leaves the matter completely open to the developers, each of whom must use their own schemes to obtain better utilization for their applications.

G3. Enforcement and protection. The resource kernel must enforce the usage of resources such that abuse of resources (intended or not) by one application does *not* hurt the guaranteed usage of resources granted by the kernel to other applications. Traditional real-time operating systems such as those compliant with the POSIX Real-Time Extensions [30] do *not* satisfy this goal.

[2]This emphasis on guarantees and timeliness may understandably seem to bias the resource kernel away from multimedia systems. (In real-time systems, missed deadlines may potentially lead to system failure, and possible loss of life and/or property). However, we believe that as multimedia applications on desktops and internet appliances mature, users will come to expect smooth video frame changes, jitterless audio, and synchronized audio and video. It is to be noted that VCR/TV/satellite technologies have been delivering such guaranteed timing behavior for years. It seems rather illogical to expect anything less from computers at least when a user is willing to pay for it, particularly if virtual reality environments must seem real, or for applications such as tele-medicine and tele-surgery.

G4. Access to multiple resource types. The resource kernel must provide access to multiple resource types such as processor cycles, disk bandwidth, network bandwidth, communication buffers and virtual memory. The communication protocol stack on the system may potentially be viewed as a resource type as well, but in most systems, they use the processor and hence can be managed by appropriate scheduling and allocation of processor cycles. For example, see [23]. Traditional real-time operating systems provide mechanisms that can *only* be used to guarantee access to processor cycles.

G5. Portability and Automation. The absolute resource demands needed for a given amount of work can unfortunately vary from platform to platform due to differences in machine speed. For example, a signal processing algorithm can take 10ms on a 200MHz Pentium but take 20ms on a 100MHz Pentium. Ideally, applications must have the ability to specify their resource demands in a portable way such that the same resource specification can be used on different platforms. In addition, there must exist means for the resource demands of an application to be automatically calibrated.

G6. Upward compatibility with fielded operating systems. A large host of commercial and proprietary real-time operating systems and real-time systems exist. Many of these systems employ a fixed priority scheduling policy [12] to support provide real-time behavior, and the rate-monotonic [18] or deadline-monotonic [17] priority assignment algorithm is frequently used to assign fixed priorities to tasks. Basic priority inheritance [33] is used on synchronization primitives such as mutexes and semaphores to avoid the unbounded priority inversion problem when tasks share logical resources. For example, Solaris [11], OS/2, Windows, Windows NT, AIX, HP/UX all support the fixed priority scheduling policy. The Java virtual machine specification also does. Priority inheritance on semaphores is supported in all these OSs (except Windows NT). POSIX real-time extensions, Unix-derived real-time operating systems such as QNX and LynxOS, and other proprietary real-time operating systems such as pSOS, VxWorks, VRTX, OS/9000, and iRMX support priority inheritance and fixed priority scheduling. To be accepted, the resource kernel must be upward compatible with these schemes. The priority inheritance scheme is also used or being considered for use in multimedia-oriented systems [28, 40].

Goals G1-G4 are integral to resource kernels, while goals G5 and G6 are for practicality and convenience. Goals G1, G2, G5 and G6 can be satisfied by appropriate extensions to traditional real-time operating systems which support fixed priority CPU scheduling. For example, a user-level server can perform admission control using a resource specification model similar to ours, and assign fixed priorities based on the resource parameters used by our model. However, in order to satisfy goals G3 and G4, the internals of these operating systems need to be modified in ways similar to our resource kernel design and implementation.

2.2. An Historical Perspective of our Real-Time Resource Management Model

Many deployed real-time systems have been built and analyzed using the fixed priority periodic task model first proposed by Liu and Layland [18]. This model employs two parameters, a maximum computation time *C* needed every periodic interval *T* for each activity that needs to be guaranteed. The rate-monotonic scheduling algorithm [18] assigns fixed priorities[3] based only on *T* and is an optimal fixed priority scheduling algorithm. Instead of using priorities, if the {*C*, *T*} model is directly used in a real-time system, the assumptions underlying the Liu and Layland model can be monitored and enforced at run-time. Following this strategy, the "aperiodic server" model [13, 37] uses artificially introduced *C* and *T* values for new "server tasks" which can then service aperiodic tasks within a periodic setting. This bounded periodic usage was adopted by the processor reservation work carried out in [25].

We build on this proven trend by identifying, designing, implementing and evaluating significant kernel extensions to the Liu and Layland work along multiple dimensions:

- using arbitrary deadlines [16, 17] to obtain fine-grained control timeliness of concurrent activities,
- applying the priority inheritance solutions explicitly to the unbounded priority inversion problem when activities share resources [2, 31, 34],
- dealing with new resource types such as disk scheduling, a problem which has not been studied in depth in the Liu and Layland model, and
- combining the scheduling of multiple resources into a single common framework observing that the problem of scheduling multiple resources with deadlines is known to be an NP-complete problem [5].

2.3. The Resource Reservation Model

The resource kernel gets its name from its resource-centricity and its ability of the kernel to

- apply a uniform resource model for dynamic sharing of different resource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage,

The resource kernel attains these capabilities by reserving resources for applications requesting them, and tracking outstanding reservation allocations. Based on the timeliness requirements of reservations, the resource kernel prioritizes them, and executes a higher priority reservation before a lower priority reservation if both are eligible to execute.

[3] A lower *T* yields a higher priority.

2.4. Explicit Resource Parameters

Our resource reservation model employs the following parameters: computation time *C* every *T* time-units for managing the net utilization of a resource, a deadline *D* for meeting timeliness requirements, a starting time *S* of the resource allocation, and *L*, the life-time of the resource allocation. We refer to these parameters, {*C*, *T*, *D*, *S* and *L*} as explicit parameters of our reservation model. The semantics are simple and are as follows. Each reservation will be allocated *C* units of usage time every T units of absolute time. These *C* units of usage time will be guaranteed to be available for consumption before *D* units of time after the begining of every periodic interval. The guarantees start at time *S* and terminate at time *S* + *L*.

2.5. Implicit Resource Parameter

If various reservations were strictly independent and have no interactions, then the explicit resource parameters would suffice. However, shared resources like buffers, critical sections, windowing systems, filesystems, protocol stacks, etc. are unavoidable in practical systems. When reservations interact, the possibility of "priority inversion" arises. A complete family of priority inheritance protocols [31] is known to address this problem. All these protocols share a common parameter *B* referred to as the blocking factor. It represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. If its *B* is unbounded, a reservation cannot meet its deadline. The resource kernel, therefore, implicitly derives, tracks and enforces the implicit *B* parameter for each reservation in the system. Priority (or reservation) inheritance is applied when a reservation blocks, waiting for a lower priority reservation to release (say) a lock. As we shall see in Section 4.5, this implicit parameter *B* can also be used to deliberately introduce priority inversion in a controlled fashion to achieve other optimizations.

2.6. Reservation Type

When a reservation uses up its allocation of *C* within an interval of *T*, it is said to be *depleted*. A reservation which is not depleted is said to be an *undepleted* reservation. At the end of the current interval *T*, the reservation will obtain a new quota of *C* and is said to be *replenished*. In our reservation model, the behavior of a reservation between depletion and replenishment can take one of three forms:

- *Hard reservations*: a hard reservation, on depletion, cannot be scheduled until it is replenished. While appearing constrained and very wasteful, we believe that this type of reservation can act as a powerful building block model for implementing "virtual" resources, automated calibration, etc.
- *Firm reservations*: a firm reservation, on depletion, can be scheduled for execution only if no other undepleted reservation or unreserved threads are ready to run.
- *Soft reservations*: a soft reservation, on depletion, can be scheduled for execution along with other unreserved threads and depleted reservations.

2.7. System Call Interface to Reservations

System Call	Description
`Create`	Create a reservation port
`Request`	Request resource on reservation port
`Modify`	Modify current reservation parameters
`Notify`	Set up notification ports for resource expiry messages.
`Set Attribute`	Set attributes of reservation (hard, firm or soft reservation)
`Bind`	Bind a thread to a reservation.
`GetUsage`	Get the usage on a reservation (accumulated or current) .

Table 2-1: A subset of the reservation system call interface for each resource type.

Our resource reservations are *first-class* entities in the resource kernel. Hence, operations on the reservations must be invoked using system calls. A select subset of the system call interface for the resource reservation model is given in Table 2-1. A reservation modification call allows an existing reservation to be upgraded or downgraded. If the modification fails, the previous reservation parameters will be restored. In other words, if an application cannot obtain higher resources because of system load, it will at least retain its previous allocation. A notification registration interface allows the application to register a port to which a message will be sent by the resource kernel each time the reservation is depleted. A binding interface allows a thread to be bound to a reservation. More than one thread can be bound to a reservation. Query interfaces allow an application to obtain the current list of reservations in the system, the recent usage history of those reservations (updated at their respective *T* boundaries), and the usage of a reservation so far in its current *T* interval.

3. Processor Resource Management

In this section, we shall discuss and evaluate our implementation of the resource reservation model for the processor resource.

3.1. Admission Control

Description	Overhead (µs)
Processor admission control with 1 reservation	25
Processor admission control with 10 reservations	120
Processor admission control with 20 reservations	195
Processor reservation creation (excluding admission control)	150

Figure 3-1: Processor Admission Control Policy Overhead w/ Exact Schedulability Conditions

Our processor reservation scheme employs a fixed priority scheme due to its widespread popularity (as mentioned in the description of goal G6 in Section 2.1). In other words, each reservation is assigned a fixed priority, which is equal to its period *T* or deadline *D*, depending upon whether a rate-monotonic or a deadline-monotonic scheme is used respectively. Our admission control policy does *not* employ (oft-used) static utilization bounds (e.g as given in [18]) since they can be very pessimistic in nature [14]. Instead, we use an exact schedulability condition which provides the best admission control for a given set of real-time threads. This algorithm is described in detail in the Appendix (Section 2). The algorithm, being a complex non-linear function of the thread periods, their relationships and their computation times, does not have a standard degree of complexity. However, it is easily coded and can be computed efficiently. The computational cost of the scheme for a wide range of thread counts is shown in Figure 3-1. As can be seen, the overhead is acceptable. It is also incurred only when a thread requests a new reservation (or upgrades an existing reservation).

3.2. Tracking Implicit Parameter *B*

When a lower priority reservation blocks a higher priority reservation[4], the former inherits the reservation (and therefore its priority). When the higher priority reservation finally unblocks, the inheritance is revoked. However, the duration for which the inheritance was in place is priority inversion. The resource kernel tracks and accumulates the duration of priority inversion during a reservation's *T*. If it exceeds the maximum *B* that can be tolerated by that reservation, a message is sent to the reservation's notification port.

3.3. Performance Evaluation

Reservation Type	Initial Reservation				Upgraded to				Doungraded to			
	Ci (ms)	Ti (ms)	Di (ms)	Ci/Ti	C_i (ms)	T_i (ms)	D_i ms)	C_i/T_i	C_i (ms)	T_i (ms)	D_i ms)	C_i/T_i
Hard	8	80	60	0.1	12	80	60	0.15	4	80	60	0.05
Firm	15	80	70	0.19	19	80	70	0.24	11	80	70	0.14
Soft	20	80	80	0.25	24	80	80	0.3	20	80	80	0.25

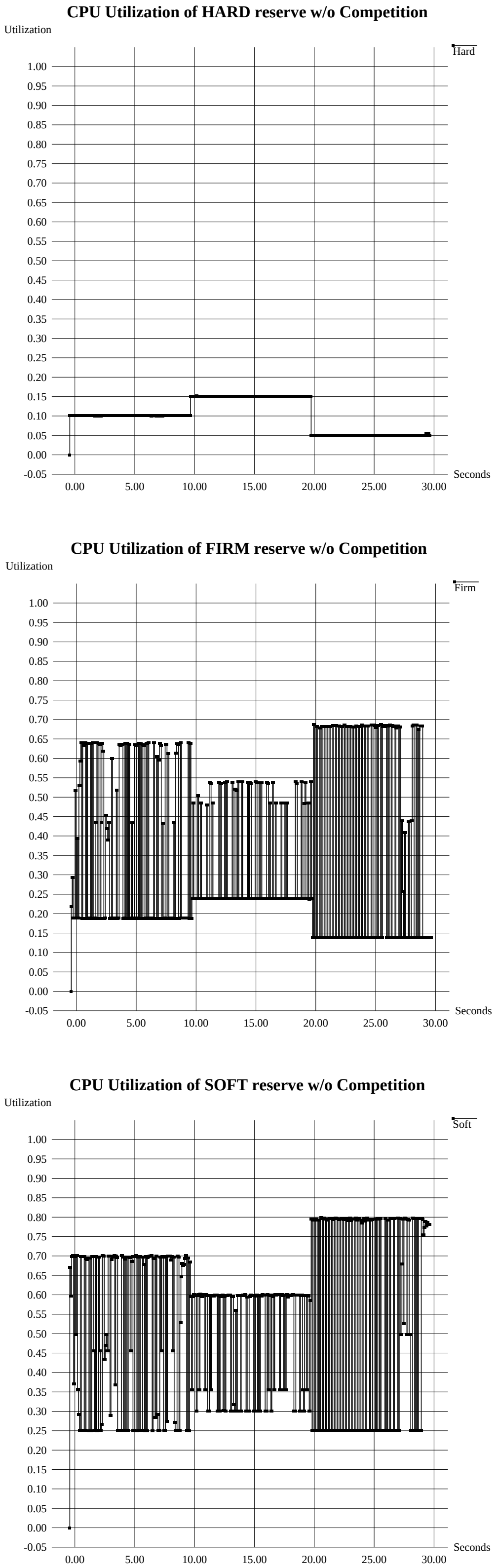
Table 3-1: The processor reservation parameters used for Figures 3-2, 3-3 and 3-4

We now evaluate the processor reservation scheme by running different workloads with and without the use of reservations. All our experiments use a PC using a 120MHz Pentium processor with a 256KB cache and 16MB of RAM. We illustrate two basic points in these experiments:

1. the nature of the three types of reservations, and
2. the flexibility to upgrade and downgrade different reservations. dynamically.

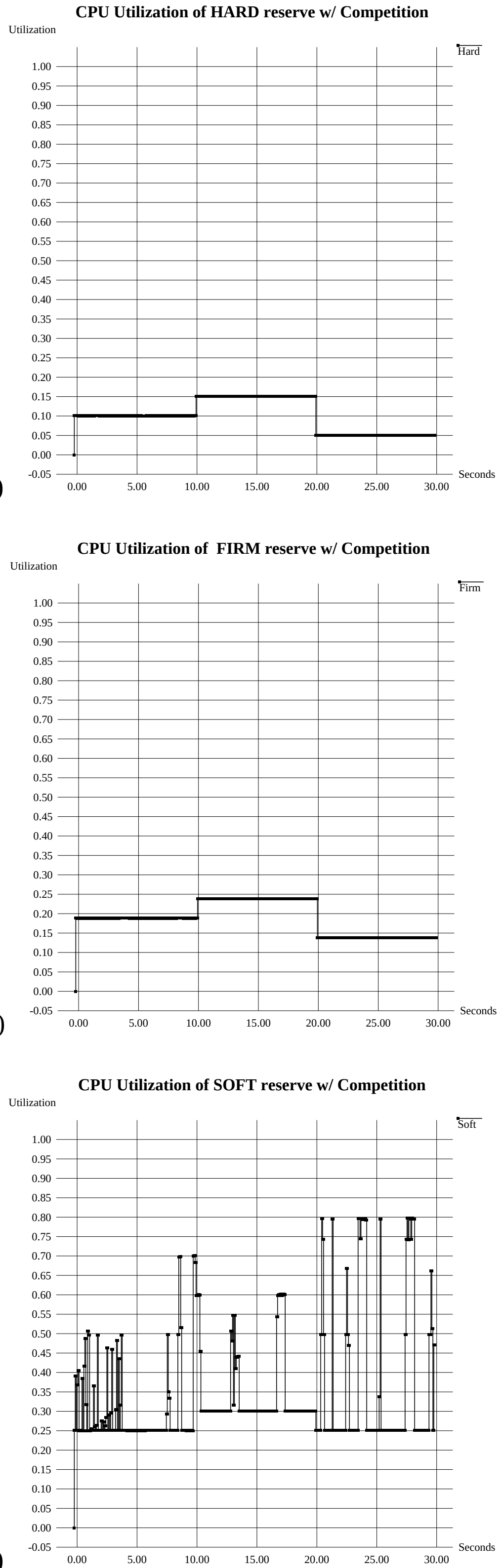
In the experiments of Figures 3-2 and 3-3, three threads running *simultaneously* in infinite loops are bound to the three reservations listed in Table 3-1. In the experiment of Figure 3-2, only these three threads are running. In contrast, in the experiment of Figure 3-3, many other unreserved threads in infinite loops are also running in the background and competing for the processor. The behavior of the three types of reservations is illustrated between these two figures.

[4]This terminology means that a thread bound to a lower priority reservation is blocking a thread bound to a higher priority reservation.



(a) (b) (c)

Figure 3-2: Behavior of *reserved* infinite loop threads *without* unreserved competition



(a) (b) (c)

Figure 3-3: Behavior of *reserved* infinite loop threads *with* unreserved competition

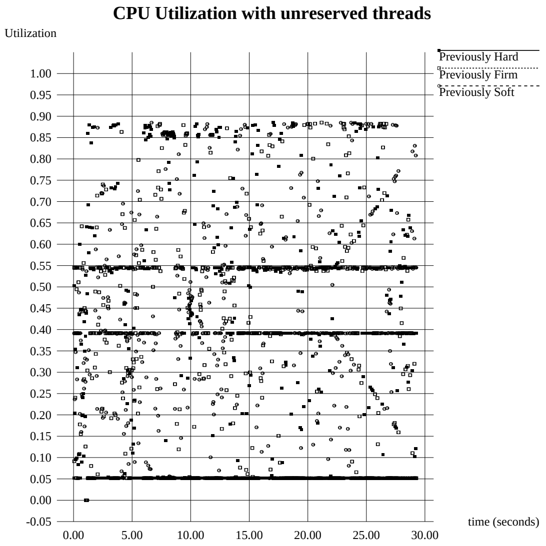


Figure 3-4: Behavior of *unreserved* infinite loop threads with unreserved competition

- The first reserved thread is bound to a hard reservation and should not consume more than its granted utilization which is initially 10%, explicitly raised to 15% at time 10, and then explicitly dropped to 5% at time 20. As can be seen in (a), this thread, despite running in an infinite loop and the presence of many competing threads, obtains exactly this specified usage in *both* Figures 3-2 and 3-3.
- The second reserved thread is bound to a firm reservation, and is allocated 19% of the CPU initially, upgraded to 24% at time 10, and then downgraded to 14% at time 20. In Figure 3-2-(b), when there is no unreserved competition, this thread obtains a minimum of its granted utilization. In addition, it obtains "spare" idle cycles from the processor since there are no unreserved competing threads. However, in Figure 3-3-(b), when there is *always* unreserved competition, this thread obtains only its granted utilization. Thus, as intended, a firm reservation behaves like a hard reservation when the processor is *not* idle, and like a soft reservation when idle processor cycles are indeed available.
- The third reserved thread is bound to a soft reservation, which is allocated 25% initially, upgraded to 30% at time 10, and then downgraded to 25% at time 20. A soft reservation can compete for cycles left behind by any threads with currently undepleted reserves. As a result, this thread obtains more cycles than its granted utilization in both Figures 3-2-(c) and 3-3-(c). It must be noted that the thread obtains a minimum of its granted utilization during all its instances. It can also be seen that this thread obtains more processor cycles in Figure 3-2 since it competes only with one thread bound to a firm reservation.

It must be recalled that the three threads of Figures 3-2 and 3-3 are running simultaneously. The completion times of this same set of threads (with the background competition of Figure 3-3) when run without using any reservations are plotted in Figure 3-4. The same threads which behave extremely predictably in Figure 3-3 now exhibit an enormous amount of seemingly random and practically unacceptable unpredictability. This demonstrates that our resource management scheme works as expected; without the scheme, resource usage is not predictable.

C_i (ms)	T_i (ms)	D_i (ms)	U_i=(C_i / T_i)
5	20	10	25%
10	40	30	25%
10	60	45	16.66%

Table 3-2: The processor reservation parameters used for the experiment of Figures 3-5 and 3-6

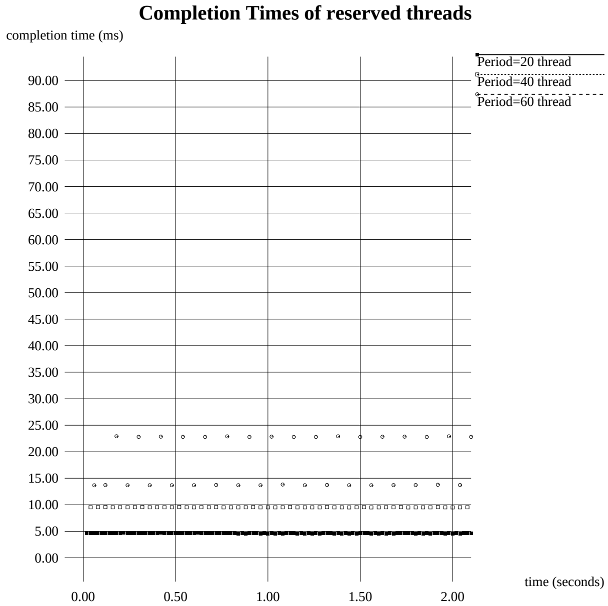


Figure 3-5: Completion times of reserved threads in the presence of competing threads

We now run another experiment where each thread imposes only finite demands, but the completion times of these demands can be predictable only with explicit resource management. The reservation parameters used for this experiment are listed in Figure 3-2. Note that the deadlines are substantially smaller than the reservation periods giving finer grained control over timeliness. One thread for each of the three reservations is created with the same period and (slightly less) computation time as its corresponding reservation. When using reservations in our resource kernel, the completion times for each of these threads as they execute with their different periods was time-stamped. The corresponding results are plotted in Figure 3-5. As can be seen, all the three threads complete their executions well ahead of their deadlines. Thread 2 also has a constant completion

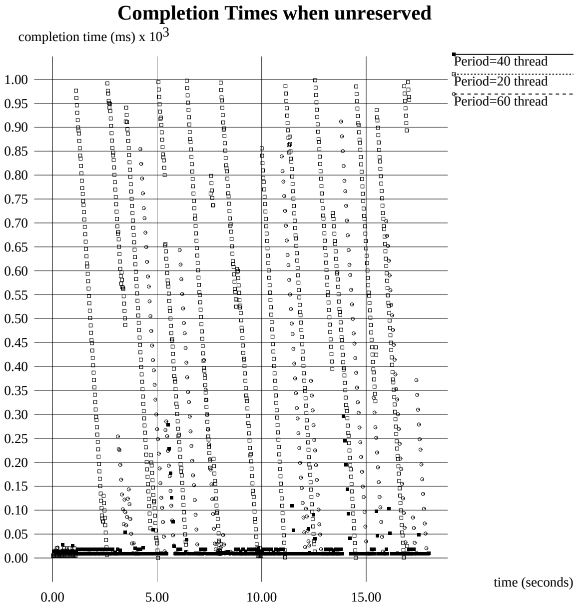


Figure 3-6: Completion times of unreserved threads with competing threads

time despite its lower priority because of its harmonicity with thread 1. The behavior of the completion times when no reservations are used is depicted in Figure 3-6. As can be seen, the same threads have widely varying completion times and also miss their deadlines rather frequently.

To summarize, the resource kernel provides a guaranteed slice of processor resources to applications independent of the behavior of other applications (including execution in infinite loops). Processor cycles that are idle can also be *selectively* allocated to running tasks.

4. Disk Bandwidth Resource Management

Traditional real-time systems have largely avoided the use of disks. This is in part because they may be relatively slow for some real-time applications. However, many real-time applications can benefit from the use of disks to store and access real-time data (such as real-time database applications). Unfortunately, the use of a disk can (a) introduce unpredictable latencies, and even worse (b) the disk access requests must now be managed in conjunction with the processor scheduling. On the processor side, fixed priority algorithms allow a mix of tasks with different periodicity, and hence the disk subsystem must do too. This problem has not been studied extensively partly because the multiple resource problem with deadlines is known to be NP-complete [5]. Some exceptions can be found in [1, 20, 21] but their resource specification models and metrics are very different from the ones we study. The closest scheduling model to ours is found in [6] but its approach is one of using fixed priority scheduling, minimizing blocking through the use of "chunking" and using a static task set. Also, only simulation studies were carried out. In contrast, we use dynamic priority scheduling, exploit blocking instead of minimizing it and evaluate an implementation within our resource framework. In addition, we deal with processor needs that must be dealt with concurrently .

Desktop multimedia systems also need to read from (or write to) disk storage relatively large volumes of video and audio data. In addition, these streams represent continuous media streams, and must therefore be processed by the disk subsystem in real-time. In other words, it would be practically very useful in practice if disk bandwidth can also be guaranteed in addition to managing processor cycles.

In this section, we present a simplistic disk scheduling algorithm based on earliest deadline scheduling. We then improve the algorithm by exploiting "slack" in the reservations to obtain a hybrid of earliest deadline scheduling and a traditional scan algorithm. Our evaluations of these schemes that guaranteed disk bandwidth reservation can be obtained at only a small loss of system throughput.

4.1. Filesystem Bandwidth Specification

Our resource specification model for disk bandwidth is identical to that of processor cycles. In other words, a disk bandwidth reservation must specify a start time S, a processing time C to be obtained in every interval T before a deadline of D. The processing time C can be specified as # of disk blocks (as a portable specification) or in absolute disk bandwidth time in native-platform specification.

4.2. Admission Control

Our simplest disk head scheduling scheme employs the earliest deadline scheduling algorithm [18]. The earliest deadline scheduling algorithm is an optimal scheduling algorithm for our scheduling model and can guarantee 100% resource utilization under ideal conditions. In other words, a higher priority reservation must be able to preempt a lower priority preemption preemptively, and $D_i = T_i$. However, instantaneous preemptions are not possible in disk scheduling. An ongoing disk block access must complete before the next highest priority disk block access request can be issued. This introduces a blocking (priority inversion) factor of a single filesystem block access (as per [35]). Also, when $D_i < T_i$, the required earlier completion time (of $T_i - D_i$, must be added to the blocking factor. A detailed discussion of this admission control policy is beyond the scope of this paper and can be found in [27].

4.3. Scheduling Policy

Instances of a disk bandwidth reservation become eligible to execute *every* T_i units (at times S_i, $S_i + T_i$, $S_i + 2T_i$, $S_i + 2T_i$, $\cdots$). Consider an instance which arrives at time $S_i + nT_i$. This instance has a deadline of $S_i + nT_i + D_i$. Similarly, all instances of all outstanding disk bandwidth reservations have corresponding deadlines. After each disk block access is completed, the disk scheduler makes another scheduling decision. It picks the next ready reservation instance with the earliest deadline and issues a disk access command corresponding to that instance's next disk access request. If there are no pending requests, the disk remains idle.

4.4. The Architecture of the Reserved Filesystem

The architecture of the reserved filesystem follows a traditional scheme. A *Real-Time File Server* running on top of our resource kernel (based on the RT-Mach microkernel) manages the reserved real-time filesystem. RT-FS has multiple worker threads which receive and process filesystem access requests from real-time clients. Each worker thread stores the incoming request it is processing into a common io-request queue. The worker thread responsible for issuing the current disk block access waits for its completion. It then awakens, and determines the next request based on the scheduling policy above. If the next request corresponds to another worker thread, that thread is signaled. Else, the worker thread continues to service its remaining disk access requests, if any.

4.5. Exploiting 'B': Just-In-Time Disk Scheduling

The earliest deadline disk scheduling described above blindly picks the next block with the earliest deadline irrespective of the current position of the disk head. Since the physical movement of the disk head and the disk's rotational latencies constitute significant durations of time, such dynamic scheduling can result in significant disk subsystem throughput reductions particularly under heavy disk traffic. The reductions can be directly attributed to the time wasted by the disk head moving from one end to another and the disk's rotational time. In summary, the deadines are preferred over a block's physical location.

Traditional scan algorithms, in contrast, re-order the disk request queue such that the block closest to the current head position (in the direction of movement) is accessed next. As a result, a disk request which just arrived can be serviced before another disk request which has been waiting for a long time just because the latter is farther away from the head position. To summarize, the physical block location is favored over timeliness.

The earliest deadline scheduling algorithm and the scan algorithm are therefore at odds with one another. Fortunately, a hybrid scheme which can obtain all the benefits of the earliest deadline scheduling algorithm and at least part of the benefits of the scan algorithm is possible. In priority-driven scheduling, higher priority activities preempt lower priority activities. Since both lower and higher priority activities must be schedulable in an admission-controlled system, higher priority activities typically complete well ahead of their deadlines. In other words, such higher priority activities have a good amount of "slack" in their completion times. Based on this observation, we present a new algorithm we call "Just-in-time" disk scheduling. This algorithm exploits the slack available to higher priority tasks to schedule accesses of other disk blocks which are closer to the current head position.[5]

A brief description of the just-in-time disk scheduling algorithm is as follows. The maximum "slack" available to each disk reservation is computed whenever a new request is admitted (or an existing reservation is deleted). At run-time, if the current slack of higher priority reservations is non-zero, another unreserved (or lower priority reserved) request can be scheduled if closer to the disk head. If slack is stolen, the slack of higher priority reservations is reduced by one. This process is then repeated. If the slack of a high priority reservation goes to zero, it will be serviced independent of its location. The detailed description of the just-in-time algorithm can be found in [27].

4.6. Performance Evaluation

The capability of the disk bandwidth reservation scheme in our resource kernel to satisfy demands on disk bandwidth is illustrated in Figure 4-1. One disk bandwidth reservation of 12 disk blocks every 250 *ms* was requested in the presence of other unreserved accesses to the disk. As can be seen from the plot, this demand is satisfied by both the earliest deadline scheme and the just-in-time scheme; in fact, both lines are flat and coincide almost completely in the plot. However, the scan algorithm attempts to optimize disk throughput and pays for it by *not* delivering the needed throughput of 12 blocks every 250 *ms*. As a matter of fact, the bandwidth consumption varies widely.[6]

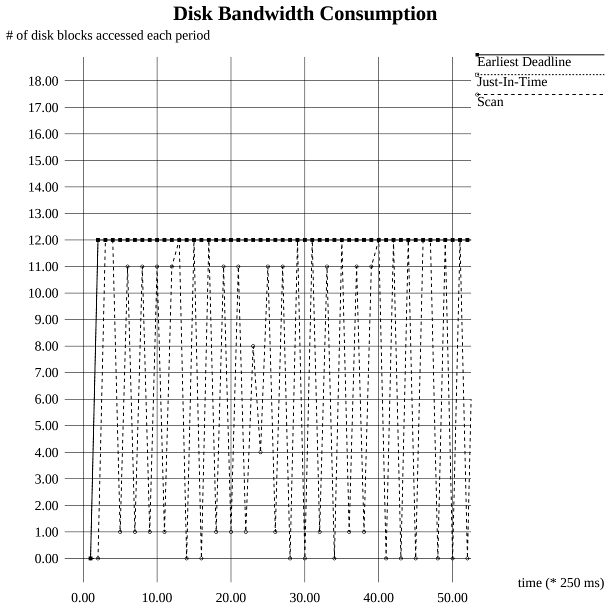


Figure 4-1: Disk Bandwidth consumed (# of disk blocks read) by reserved thread. Earliest deadline and Just-In-Time reservation schemes are flat and coincide almost completely.

We also imposed heavy disk traffic conditions and measured the throughput obtained under the scan and earliest deadline algorithms. This is shown in Table 4-1. As can be seen, the earliest deadline algorithm obtains only about 10% less throughput than the scan algorithm. This is the price to be paid for the predictable and guaranteed disk bandwidth obtained by the earliest deadline algorithm (as shown in Figure 4-1).

[5]Such "slack-stealing" has been done in the context of processor scheduling theory in order to provide better response to aperiodic activities [15]. The optimization, cost functions and implementation tradeoffs seem to be different for the processor and the disk, however.

[6]The pattern is more dramatic in a zoomed out view with the x-axis ranging upto 400 periods, but the lines/points are not clearly legible in a relatively small black-and-white graph.

Requested throughput (KB/s)	Throughput with Scan (KB/s)	Throughput for Earliest Deadline Scheduling (KB/s)	Throughput Degradation (%)
1158.6	856.36	764.88	10.68%

Table 4-1: Scan and EDF real-time filesystem throughput comparison

4.6.1. Synthetic Workload Behavior with both CPU and Disk Requirements

We next imposed a synthetic workload to determine the completion times of disk access requests, and to study the drop in disk throughput when the Scan policy is replaced with a policy which attempts to satisfy timing constraints in preference to enhancing disk throughput.

As illustrated in Figure 4-2, the real-time workload tested consists of two threads, *Thread 1* and *Thread 1b*. *Thread 1* reads periodically from disk and copies all the data into buffer A, while *Thread 1b* processes data previously stored in buffer B. At the end of the period, there is a buffer switch and the role of both buffers is interchanged. Buffers A and B have the same size. We bound disk bandwidth and CPU reserves to *Thread 1*, and a CPU reserve to *Thread 1b*, and traced the execution in terms of completion times, deadline misses and disk utilization. *Thread 1* makes use of relatively little cpu time and it sleeps till the beginning of the next period to invoke a new read operation. *Thread 1b* processes the data previously stored in the buffer. Both *Thread 1* and *Thread 1b* have a period of 250 *ms*. Also, *Thread 1* reads 44 KBytes during each of its instances, and has a deadline of 162 *ms* for completing its reads. Note that this deadline is shorter than its period of 250 *ms*, forcing a stringent test for the filesystem. *Thread 1b* is offset from *Thread 1* by 162 ms and has a deadline of 88 *ms*.

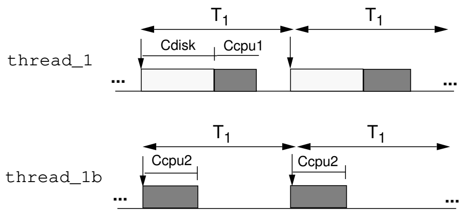
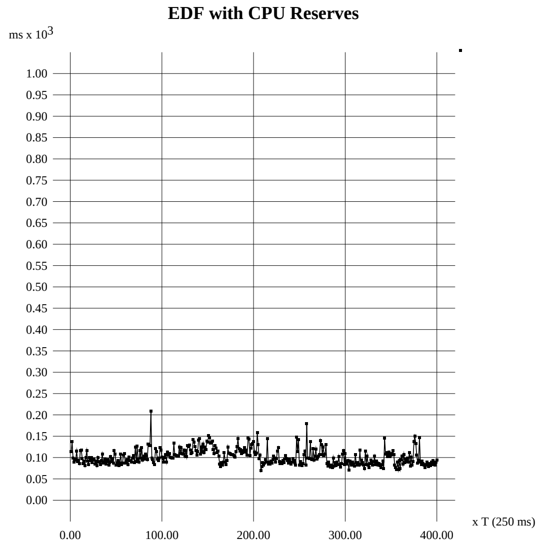


Figure 4-2: Execution patterns of *Thread 1* and *Thread1b*

Six periodic threads each with a different period (varying from 300 *ms* to 640 *ms*) and a different read-access load on the disk (varying from 8 KBytes to 200 KBytes per periodic instance of the thread) were introduced as competing threads without any reserved capacity on either the CPU or the disk bandwidth. We ran this workload for a duration of 100 seconds and measured the completion times of each periodic instance, and the total disk bandwidth consumed. The completion times are illustrated in Figure 4-3.

If we use EDF/EDF+JIT without reserving the CPU there

(a)



(b)

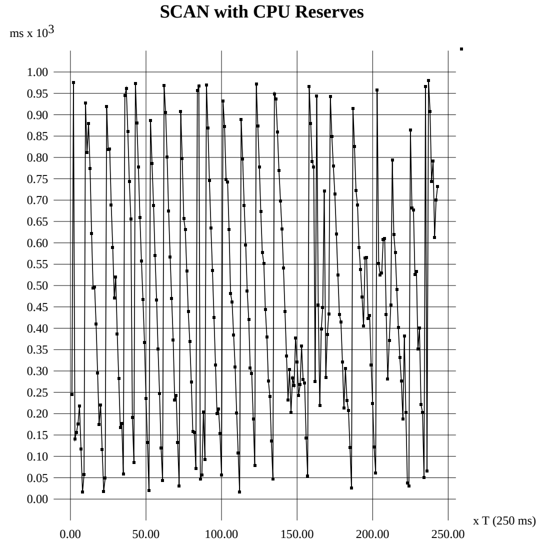


Figure 4-3: *Thread 1* Completion Times

are some deadlines misses (2 out of 400: periods 88 and 258). In these cases the task finished after 162 *ms* (but never after the period of 250 *ms*). These two deadline misses are due to the fact that our filesystem (extension of the Berkeley Fast FileSystem) does not allocate blocks contiguously on disk. So relatively high interblock seek times out of the cylinder group may happen from time to time even with requests for successive blocks within a file from the same thread. This can happen for each 1 MB of filesystem data according to the Berkeley FFS allocation algorithm and can lead to potential deadline misses. Accounting for the worst-case interblock seek times in the admission control test would avoid this problem, but can lead to extremely low guaranteed disk throughput. Thus, not withstanding our admission control test, some deadlines can be missed. However, as can be seen from our experiments, the deadline misses are rather infrequent. Conversely, in the Scan case there is no time to run the needed 400 disk accesses and only 248 accesses are completed within the experiment duration. The completion times are nearly always greater than the period itself (> 250 *ms*) and sometimes much greater. This shows that EDF w/ CPU reserves consistently meets the timeliness constraints of the real-time application accessing the disk.

Disk Throughput: The total disk bandwidth consumed in the above experiment was 16,464 KBytes with the EDF and CPU reserve policy, and 17,750 KBytes with the Scan policy. This represents only a performance throughput loss of 7.25%. In return, however, the timing constraints and periodic bandwidth requirements are satisfied with the EDF/CPU reserves policy, while they are dramatically unsatisfied with the Scan policy.

5. Practical Issues

5.1. Using Different Reservations Together

Consider a video display application which reads a video movie from the local disk and displays it on the screen. The movie is long enough that it does not fit into memory. As a result, subsequent video frames must be read from the disk while frames already read into a double buffer are being played on the screen. In this case, the video display algorithm must not only be scheduled on the CPU (where it can also do decompression or special signal processing) but also obtain guaranteed disk bandwidth to display the movie and its audio track without user-perceptible jitter.

The most straightforward way of approaching this problem is as follows: the application consists of a single thread which binds itself to a processor reservation and a disk bandwidth reservation with the same period, start time and appropriate computation times to satisfy the application’s needs. There can be other threads in the system which use other combinations of resources (such as the processor and network bandwidth). Each of these reservations need to satisfy their associated deadlines given by the parameter *D*. However, it is known that the problem of scheduling concurrent tasks on multiple resources with timeliness constraints is NP-complete [5]. As a result, one faces the dilemma of finding a practical acceptable solution, since finding an optimal solution to the problem is very impractical. We address this problem next.

5.2. Resource Decoupling

Since simultaneous access to multiple resources is the problem we face, a natural solution to the practical dilemma one faces is to try to decouple the use of different resources, which can be used independent of one another. An end-to-end timing constraint problem is normally intractable as a single big problem, and is hence solved as a series of small problems where each problem only spans a single resource. For example, in an audio-conferencing application [23], the first pipeline stage occurs in the sound card which transfers data to the processor using *periodically self-initiated* DMA or multi-master bus transactions. A 2nd pipeline stage occurs on the processor to transmit the data and the next stage occurs in the network. The end-to-end delay for audio is the sum of the delays encountered in each of the audio pipeline stages. We refer to this strategy where each of the resources involved are scheduled independent of one another as resource decoupling. When resources are decoupled, for example, the completion time test of Section 2 can be applied to each resource independently.

In the audio-conferencing application, the only coupling between the pipeline stages lies actually at the interface between stage 2 and the network (or the network and stage 4).[7] When the processor is ready to send out packets, the network must be able to transmit them. Memory buffers on the network interface card provide some decoupling by storing packets that the processor is ready to transmit, but the network is not ready to accept yet. We address this problem next.

5.3. Processor Co-Dependency

A phenomenon that we name *processor co-dependency* provides a hint to the solution. Complete resource decoupling seems possible between any two resources if neither of them is the processor. Since the processor is the brain of the system, communications between the network and the disk, for example, must go through the processor. The processor must obtain the network packets and then send them to the disk. In other words, a coupling problem which at first sight is between the disk and the network gets translated into two independent couplings between the disk and the processor, and between the processor and the network. The net result is that as soon as the disk (or the network) demand attention from the processor, the processor must be able to provide it.

5.4. "Immediate" or "System" Reservations

In our resource reservation model, we define the concept of a "system reservation" which is a highest priority reservation which does not get depleted. As a result, any thread or threads bound to a system reservation will be able to execute at the highest priority as soon as possible (subject only to other threads using a system reservation). We also sometimes refer to the system reservation as an "immediate reservation" because of the immediacy of its service. Clearly, the use of "system reserves" must be confined to trusted services only (to satisfy goal G3 of resource kernels), which must be trusted to use them only sparingly for relatively quick transfers of data. The worker threads in the reserved filesystem of Section 4.4 also fall into the category of system reservation users. It must be remembered that the usage of the system reservation will adversely affect new resource requests and must be accounted for in admission tests.

We measured the time consumed by components of a disk I/O to complete a filesystem block fetch of 4KB: time spent in core filesystem code = 532 µs, time spent in filesystem overhead (block map queries, etc.) = 171 µs, time spent in data copies = 131 µs, time spent in disk reserve overheads (scheduling, updating slack, etc.) = 230 υs, time spent in I/O = 2550 υs leading to a total elapsed time of 3082 υs. The CPU usage for the worker thread in the filesystem is there-

---

[7] If the network interface card hardware can be configured to be in auto-initiation mode as on the sound card, this coupling problem would disappear as well. This argues for better and more sophisticated support in interface cards and controllers. The trend towards MMX support and "software modems" is unfortunately in the opposite direction!

fore 532 υs out of 3082 μs = 17.26%. Since one worker thread can access the disk at any given time, this represents the worst-case processor requirement imposed by the real-time filesystem. However, due to the fact that disk seeks will not be issued continuously in a general system, this number will be lower in practice. Otherwise, for a disk-intensive context, this overhead is likely acceptable.

5.5. Calibrating an Application's Requirements

The computation time C needed for a reservation must be known in order to reserve processor time before it can be requested. It is, however, unknown practically before its actual execution since it heavily depends on a machine platform on which an application program runs. Even on machines with the same CPU and the same clock rate, the execution time may be affected by the presence of cache, the amount of memory, memory and system bus interface chip sets, and other I/O interface cards. Thus, we need to obtain C for the current platform by actually running an application on it. Obtaining C requires the kernel to support precise measurement of the processor time consumed by a certain thread. We now discuss how this can be obtained using only our resource kernel capabilities.

Our resource kernel supports hard reservations and also provides current and accumulated usage on a reservation by a program. The hard reservation ensures that any threads bound to it can only run upto its specified C. The execution time of the application program to be calibrated is then measured as follows. A new hard reservation, named (say) "calibration", is created, and the given application program is bound to it just for the purpose of measuring its execution time. The reservation will get depleted by the running of the application program, get replenished by the resource kernel, and the process will repeat until the application program completes execution. The accumulated usage on the hard reservation "calibration" now yields the execution time of the application program. An advantage of this method is that it is certain that a program can obtain its C even when the system is busy since it is guaranteed to receive a certain amount of processor time for its execution.

5.6. Portability Of Resource Specifications

As mentioned above, the absolute execution time of a program changes from platform to platform depending upon processor speed, etc. As a result, the specification of C in absolute time-units can become inherently not portable. Fortunately, portable time-units are available in the form of the number of clock ticks and the number of instructions executed for a given program segment on the processor. Of these two, the number of clock ticks is perhaps more portable since today's microprocessors contain on-chip clock counters which can not only provide high-accuracy resolution as well be inherently scalable across chips with lower or higher clock speeds. Similarly, C_i for disk bandwidth reservation can specify the number of disk blocks to be read, or better, the number of bytes to be read. The latter units will also be portable across platforms using different disk block sizes. Implementations of resource kernels must therefore provide convenience functions to translate "portable time-units" on a resource to native absolute time-units.

5.7. Adaptive QoS Management

User-level resource managers can be built on top of a resource kernel to react (or adapt to) to changes in application, system resources and the environment. In distributed real-time applications, such as video conferencing, the change in quality at one end-point typically implies that the other end-point must also adapt its quality correspondingly. Such distributed adaptations must clearly happen at a larger time-scale than single-node resource allocation changes. Similarly, we take the position that user-level application changes happen at a larger time-scale than the decisions made in the resource kernel to dynamically schedule activities on system resources. Such user-level resource managers can also potentially implement more complex resource management policies than the ones used by our resource model.

6. Concluding Remarks

We have presented a resource-centric approach to building real-time kernels, and we call the resulting kernel a resource kernel. The resource kernel provides timely, guaranteed and protected access to resources. We now compare our approach with two related approaches, and summarize our research contributions.

6.1. Resource Kernels and Related Approaches

We now compare the resource kernel notions with the approaches used by operating systems such as Nemesis [29] and Exokernel [7]. Nemesis and our resource kernel approach adopt a similar model of resource specification and allocation, based on the so-called {C, T} model originally proposed by Liu and Layland [18]. Nemesis implicitly assumes a deadline of T before which the C units of time must be available. Our resource kernel also supports a deadline shorter than T[8]. The Nemesis approach to dealing with the problem of priority inversion, a potentially significant stumbling block of multi-tasking real-time systems, is rather unclear. In our resource kernel approach, bounding priority inversion is a key principle of managing interactions between concurent real-time activities. Priority inversion, where a higher priority request is blocked by a lower priority activity, is unavoidable in the general case (such as critical sections, non-preemptible bus transactions and finite size ATM cells). However, it is imperative that unbounded priority inversion be eliminated, as in the use of semaphores in a priority-driven system [31, 35]. Such durations of priority inversion must be bounded and if possible minimized. Priority inheritance protocols have also been extended to dynamic priority algorithms [3, 9]. In resource kernels, we use priority inheritance in the form of reserve propagation [26] where a blocking thread inherits the scheduling priority of a higher priority reserved thread for the duration of the blocking.

Nemesis advocates the minimization of servers to enable correct "charging" of resource usage to applications. The Nemesis approch is to put 'server code' into client libraries, which would then use critical sections to enforce consistency requirements across multiple clients as necessary. Our

[8] A deadline longer than T is also possible.

resource kernel notions take a neutral stance on the topic of servers in that we (must) support configurations with and without servers. We do so for two fundamental reasons:

1. *Time and space are distinct*: Servers and critical sections executing in client space providing the given service are strictly analogous in a timing predictability sense, and differ only in a spatial organization sense. More precisely, the blocking (or priority version) factor is (almost) the same whether a service is implemented as a client library or within a server thread. Any difference arises only due to spatial overhead factors (primarily due to less context-switching in the case of client library implementations, for example see [24, 23]). This is hardly a fundamental question of functionality or capability. Consider a service S (such as a draw-in-window operation) executing in a real-time server like X. The server obtains requests from multiple clients. In a real-time system, the requests will be queued up in priority order *and* with support for priority inheritance to avoid unbounded priority inversion problems. If implemented as a client library, the critical section used within the library will use a mutex, which in turn will use a priority queue for waiting threads and support priority inheritance.[9]

2. *Sharing and interactions are in general unavoidable*: Concurrently running applications interact not only because they eventually share the same underlying physical resources, but also because of logical requirements above the physical layer. Shared display, shared files, concurrent access to bank accounts, shared data such as movies and databases are only some examples of these shared logical resources. As a result, critical sections which manage these shared *logical* resources are unavoidable in the context of multi-tasking and multi-threaded systems. Whether these critical sections are organized in client space or in a dedicated server is only a question of convenience and flexibility with the time/space distintions coming into play. Anyhow, critical sections can be shortened or optimized but in general cannot be eliminated.[10]

Memory implications of using a client library (with a critical section) and a server also need to be considered. When a service is implemented as a server, it is relatively easy (for example) to wire down that server memory for predictable real-time performance. However, if clients used their own libraries (with critical sections), other relatively more complex issues must be addressed. In one case, each client can have its own copy of the library leading to higher memory usage. In contrast, if shared (dynamically linked) libraries are used, memory usage is the same as a server, but one must now be able to ensure that a shared library is wireable. In other words, a finer granularity of memory control becomes necessary.

The Exo-kernel approach advocates that all policy decisions except for access protection reside in user-level programs. However, for real-time systems, the CPU scheduling policy must be centrally managed (at the "root") to ensure that an application group can satisfy its own timing constraints. This global scheduling policy *cannot* be delegated to individual applications. On the other hand, if the CPU resource management policy is deemed to be a temporal protection mechanism that resides in the exo-kernel, the resource kernel notion is actually compatible with the exo-kernel approach as well. Each application can then build its own local scheduler to use its allocated time in a way that it sees fit. However, in practice, we do not expect local schedulers in user space to provide significant added value. Instead, we propose a Quality of Service (QoS) manager running in user space (as a server) on top of the resource kernel [22, 32]. This QoS manager can arbitrate among competing requests when the maximal requests of all applications cannot be satisfied with the available resources.

6.2. Contributions

We have presented the notion of a resource kernel, which provides timely and protected access to machine resources. In this approach geared towards real-time and multimedia operating systems, guaranteed and protected access

- **Uniformity**: a single resource specification scheme can be applied to different time-shared resource types with timeliness control. The scheme can be locally optimized and applied for each resource type.

- **Resource management transparency**: the use of the exact resource management scheme is hidden from the application programs and changed transparently across different implementations. The implementation of the resource management scheme can use, among other things, fixed priority schemes such as rate-monotonic scheduling [18] and deadline-monotonic scheduling [17], dynamic priority schemes such as earliest-deadline-first [18], or processor sharing schemes such as PGPS, virtual clocks or WF^2Q [4]. We demonstrate two very different schemes for CPU and disk bandwidth management even though each uses the same resource specification model.

- **Resource composability**: We show that multiple resource types can be guaranteed at the same time with acceptable performance levels. In specific, reservations of different resource types can be independently created and then composed. We use the technique of *resource decoupling* [36] and management of *processor* co-dependency using higher priority *system reserves* to provide simultaneous access to CPU resource and another resource type simultaneously. We are unaware of other OS work where simultaneous access to two or more resources is addressed.

- *Hard* **resource reservation**: In this resource allocation scheme, the usage of a resource cannot exceed the allocated amount of the resource even if the resource is

[9] In the general case of this discussion, one should replace the notion of priority with the notion of 'scheduling attribute' which may be priorities or reserves with the basic concept remaining the same.

[10] Lock-free protocols exist but seem to be useful only under limited conditions.

idle. While this may sound draconian and wasteful, we expect that this will be a powerful building block for constructing virtual resources, which allow untrusted applications to be built and run in their own resource space with a pre-determined finite effect on other applications at all times.

- **Interactions and Disk bandwidth management**: The resource kernel is able to monitor and control priority inversion arising from the interactions between real-time tasks due to the use of common shared services. By deliberately introducing priority inversion in a controlled fashion, we demonstrate that there is no significant loss of disk subsystem throughput for acceptably substantial ranges of disk traffic while guaranteeing timely access to disk bandwidth for real-time and multimedia applications. This is achieved using a novel *just-in-time* disk scheduling scheme. Guaranteed access to disk bandwidth is obtained at the expense of a relatively small loss in throughput.
- **Flexibility of resource kernels**: Our resource kernel abstractions allow resource usage to be automatically calibrated, and to be portable across different hardware platforms.

6.3. Future Work

Our future work will include exploring network bandwidth reservation in conjunction with processor and disk reservation. Network bandwidth management has many implications in the context of a resource kernel: protocol stack overhead dominates on the CPU. As a result, network bandwidth management translates to both network reservation and CPU management. The times during which both network bandwidth and CPU cycles need to be available seem to be fairly limited, but remain to be verified.

The issue of CPU co-dependency needs to be addressed at greater length. Additional buffer space between different resource types with hardware buffers can also alleviate the problem; this is typical of today's hardware systems with self-triggered DMA on sound cards (such as the SoundBlaster 16), and bus-mastering on multi-master backplanes such as the PCI bus. Finally, distributed resource reservation in networked systems will open up another frontier of work.

Appendix: Admission Control Schemes

1. Resource Specification Notation

Let the set of n reservations requiring processor reservation be denoted as $\tau_1, \tau_2, \cdots, \tau_n$. Each reservation τ_i needs to obtain C_i units of time every T_i units of time. In addition, the C_i units of resource time must be available at or before D_i in each periodic interval separated by T_i.

2. Admission Control Using Fixed Priority Policies

The reservations are ordered in descending order of their fixed priorities such that for i = 1 to n–1, priority(τ_i) < τ_{i+1}.

In mathematical form, a necessary and sufficient condition for the schedulability of a set of periodic tasks using fixed priority scheduling is as follows [14]:

$$\forall i,\ 1 \le i \le n, \quad \min_{0 < t \le D_i} \left(\sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \le 1$$

In algorithmic form, the completion time CT_i of a reservation τ_i with a resource allocation can be computed as follows using a recurrence relation [10, 39].

1. Let $w_i^0 := C_i$.
2. Compute $w_i^{k+1} := \sum_{j=1}^{i-1} C_j \left(\left\lceil \frac{w_i^k}{T_j} \right\rceil \right)$.
3. If $w_i^{k+1} > D_i$, $CT_i := \infty$. Skip to Step 6.
4. If $w_i^{k+1} = w_i^k$, $CT_i := w^k_i$. Skip to Step 6.
5. $k := k + 1$. Go to Step 2.
6. If $CT_i \le D_i$, τ_i meets its deadline.

The completion time test is repeated for all reservations which need to be guaranteed. Even if one reservation will miss its deadline, the admission test will deny the newest incoming request.

3. Admission Control Based on Rate-Monotonic Priority Assignment

The rate-monotonic priority assignment algorithm is an optimal fixed priority algorithm when $D_i = T_i$ [18]. The reservations are ordered in descending order based on their rate-monotonic priorities (i.e., $T_i < T_{i+1}$). The admission control test use the scheme described in Section 2.

4. Admission Control Based on Deadline-Monotonic Priority Assignment

The deadline-monotonic priority assignment algorithm is an optimal fixed priority algorithm when $D_i \le T_i$ [17]. The reservations are ordered in descending order based on their deadline-monotonic priorities (i.e. $D_i < D_{i+1}$). The admission control test uses the same scheme described in Section 2.

References

[1] R. Abbott and H. Garcia-Molina. *Scheduling Real-Time Transactions with Disk Resident Data X Server*. Technical Report CS-TR-207-89, Department of Computer Science, Princeton University, February, 1989.

[2] Baker, T. P. A Stack-Based Resource Allocation Policy for Real-Time Processes. *IEEE Real-Time Systems Symposium* , Dec., 1990.

[3] Baker, T. Stack-Based Scheduling of Realtime Processes. *Journal of Real-Time Systems* 3(1):67--100, March 1991.

[4] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case Fair-Weighted Fair-Queueing. In *Proceedings of INFOCOM 96*. March, 1996.

[5] J. Blazewicz, W. Cellary, R. Slowinski and J. Weglarz. Scheduling under Resource Constraints -- Deterministic Models. In *Annals of Operations Research, Volume 7*. Baltzer Science Publishers, 1986.

[6] S. J. Daigle and J. K. Strosnider. Disk Scheduling for Multimedia Data Streams. *Proceedings of the SPIE Conference on High-Speed Networking and Multimedia Networking* , 1994.

[7] D. R. Engler, M. F. Kaashoek and J. O. Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM Symposium on Operating System Principles* , December, 1995.

[8] K. Jeffay, D. L. Stone and F. D. Smith. Kernel Support for Live Digital Audio and Video. In *Proceedings of the Second International Workshop on Network and Ope rating System Support for Digital Audio and Video*, pages 10-21. November, 1991.

[9] K. Jeffay. Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89-99. IEEE, December, 1992.

[10] Joseph, M. and Pandya. Finding Response Times in a Real-Time System. *The Computer Journal (British Computing Society)* 29(5):390-395, October, 1986.

[11] Khanna, S., Sebree, M., and Zolnowsky, J. Real-Time Scheduling in SunOS 5.0. *The Proceedings of USENIX 92 Winter* :375-390, 1992.

[12] Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993. ISBN 0-7923-9361-9.

[13] Lehoczky, J. P. and Sha, L. Performance of Real-Time Bus Scheduling Algorithms. *ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1* , May, 1986.

[14] Lehoczky, J. P., Sha, L. and Ding, Y. The Rate Monotonic Scheduling Algorithm --- Exact Characterization and Average-Case Behavior. *IEEE Real-Time Systems Symposium* , Dec, 1989.

[15] Lehoczky, J. P., Sha, L., Strosnider, J. K. and Tokuda, H. Fixed Priority Scheduling Theory for Hard Real-Time Systems. *Technical Report, Department of Statistics, Carnegie Mellon University* , 1991.

[16] Lehoczky, J. P. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proceedings of the IEEE Real-Time Systems Symposium* , Dec., 1990.

[17] Leung, J. Y., and Whitehead, J. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation* 2(4):237-250, Dec., 1982.

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

[19] Locke, C. D., Vogel, D. R., Lucas, L. Generic Avionics Software Specification. *Technical Report, Software Engineering Institute, Carnegie Mellon University* , 1990.

[20] S. Chen, J. A. Stankovic, J. F. Kurose, D. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *The Real-Time Systems Journal* 3:307-336, 1991.

[21] P. Lougher and D. Shepherd. The Design and Implementation of a Continuous Media Storage Server . *Proceedings of the 3rd International Workshop on Network and Operating System Support for Audio and Video* , November, 1992.

[22] C. Lee and R. Rajkumar and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. *In the proceedings of Multimedia Japan 96* , April, 1996.

[23] C. Lee and K. Yoshida and C. Mercer and R. Rajkumar. Predictable Communication Protocol Processing in Real-Time Mach. *In the proceedings of IEEE Real-time Technology and Applications Symposium* , June, 1996.

[24] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244-255. December, 1993.

[25] C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. May, 1994.

[26] C. W. Mercer and R. Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. May, 1995.

[27] A. Molano, K. Juvva and R. Rajkumar. Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *IEEE Real-Time Systems Symposium*. December, 1997.

[28] Needham, R. and Nakamura, A. An Approach to Real-Time Scheduling: Is it really a problem for multimedia? *The Third International Workshop on Network and Operating System Support for Multimedia* , 1992.

[29] *Nemesis, the kernel: Overview* Dickson Reed and Robin Fairbairns, Editors, May 20, 1997.

[30] *IEEE Standard P1003.4 (Real-time extensions to POSIX)* IEEE, 345 East 47th St., New York, NY 10017, 1991.

[31] Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.

[32] R. Rajkumar, C. Lee, J. P. Lehoczky and D. P. Siewiorek. A QoS-based Resource Allocation Model. *IEEE Real-Time Systems* Symposium , December, 1997.

[33] Sha, L., Rajkumar, R. and Lehoczky, J. P. Task Scheduling in Distributed Real-Time Systems. *Proceedings of IEEE Industrial Electronics Conference* , 1987.

[34] Sha, L., Rajkumar, R. and Lehoczky, J. P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *Technical Report (CMU-CS-87-181), Department of Computer Science, CMU* , 1987.

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

[36] Sha, L., Rajkumar, R. and Sathaye, S. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE (journal)* , January, 1994.

[37] Sprunt, H.M.B., Sha, L., and Lehoczky, J.P. Aperiodic Task Scheduling on Hard Real-Time Systems. *The Real-Time Systems Journal* , June, 1989.

[38] John A. Stankovic. Misconceptions about Real-Time Computing. *Computer* 21(10):10-19, Oct., 1988.

[39] Tindell, K. *An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks.* Technical Report YCS189, Department of Computer Science, University of York, December, 1992.

[40] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Operating Systems Design and Implementation*, pages 1-11. November, 1994.

Table of Contents

List of Figures

List of Tables