Resource Sharing in Reservation-Based Systems

Dionisio de Niz[+], Luca Abeni[*], Saowanee Saewong[+], Ragunathan (Raj) Rajkumar[+]

[+]Real-Time and Multimedia Systems Laboratory
Carnegie Mellon University, Pittsburgh, USA

[*]RETIS Lab, Scuola Superiore S. Anna, Pisa, Italy
dionisio@cs.cmu.edu, luca@sssup.it, ssaewong@cs.cmu.edu, raj@ece.cmu.edu

Abstract

The resource-sharing problem in priority-driven real-time systems has been studied at length, with the result that some effective and practical solutions are available for both fixed-priority and dynamic-priority systems. In recent years, real-time operating systems have begun to support the resource reservation paradigm, providing a "temporal isolation" abstraction. However, the problem of sharing logical resources across reserved applications has not been extensively studied. In this paper, we consider both the theoretical and practical implications of such resource-sharing in reservation-based systems. Moreover, we provide some experimental results from the implementation of our proposed schemes in Linux/RK, a "resource kernel" that supports reservations.

1. Introduction

Operating systems providing resource reservations are becoming increasingly popular in real-time and multimedia applications. These systems provide important advantages over traditional priority-driven ones. For example, admission control is performed online (forcing all applications to go through some form of schedulability analysis). The stated resource and timing requirements are also monitored and enforced by the operating system, ensuring *temporal isolation* between real-time tasks. Such temporal isolation is as powerful an abstraction as spatial protection in systems with address-space protection.

While experience with resource reservations seems to be relatively limited at this time, they have been shown to be effective in providing Quality of Service (QoS) guarantees to real-time and legacy applications. However, in the current generation of reservation-based operating systems (e.g. [11, 16, 15]), resource reservations can generally provide temporal isolation only if the application tasks are independent from one another, and many real applications do not respect this assumption. For instance, in Unix-like systems, multimedia applications use the X server to display graphical content on the screen. If the X server does not have an appropriate resource reservation, a client task can be blocked in an uncontrollable fashion waiting for the X server to complete processing its requests. A similar problem happens when multiple tasks use a mutual exclusion mechanism to access shared resources.

In this paper, we study two approaches to achieve temporal isolation when applications do share common user-level services and resources: extensions to *priority inheritance techniques* as well as *priority ceiling protocol techniques*.

1.1. Related work

Many deployed real-time systems have been built and analyzed using the periodic task model proposed by Liu and Layland [10]. This model characterizes each task with two parameters: a maximum computation time C needed every recurring interval T. The rate-monotonic policy [10] assigns fixed priorities based only on the period T; this assignment is an optimal fixed-priority scheduling algorithm.

Mercer et al. [11] propose the use of the (C, T) model directly in the OS API letting the OS perform the appropriate priority mapping. As a result, not only is the application decoupled from the priority mapping but the OS can also monitor and enforce this time-behavior at run-time. The Real-Time and Multimedia Systems Laboratory at Carnegie Mellon [15] has developed a "Resource Kernel" (RK), a kernel that provides timely, guaranteed, and enforced access to system resources. Resource reservations are used as a general framework for accessing any time-multiplexed resource. This resource reservation support was implemented first in the RT-Mach μkernel [11] and then in monolithic kernels, Linux [13] and Windows/NT. Moreover, [13] also demonstrated that the resource management subsystem can be built in a "portable" OS-independent fashion.

CPU reservation techniques have also been implemented by other researchers using the earliest-deadline-first (EDF) priority assignment [5] to take advantage of dynamic priorities [1]. Nemesis [16] is a kernel developed from scratch in order to be able to provide clean support for resource reservations to satisfy QoS requirements. Nemesis implicitly assumes a deadline of T before which the C units of time must be available to a reserved task. One of Nemesis' goals is to minimize, if not eliminate, interactions ("cross-talk") between different applications so that problems from priority inversion are avoided. In contrast, we believe that such interactions cannot be eliminated in general, and that OS mechanisms must be available to bound and minimize the impact of any resulting priority inversion.

The uncontrolled priority inversion problem [17] is well-known in the real-time systems literature, and has been successfully addressed using priority inheritance techniques [17, 14]. Initially developed on fixed priority systems, these techniques have been extended to dynamic-priority scheduling schemes such as EDF [2, 4]. In [9], a new scheme is proposed to share resources between periodic and aperiodic tasks (using an aperiodic server).

In [6], the authors propose a technique to share resources in an "Open System". Their technique uses some ideas from [7] and [9], where the Stack Resource Policy [2] is used to share resources between dynamic priority aperiodic servers. This approach has been recently used in [3] to share resources between aperiodic servers used to provide temporal protection.

The problem of these solutions is that they require an a-priori knowledge of the resource access times, and temporal protection can be broken if this knowledge is not accurate enough. Section 2 will show a similar problem when a basic priority inheritance scheme is used. A similar scheme, based on dynamic priorities, is presented in [8], where the problem of sharing resources between hard and soft real-time tasks is also addressed.

1.2. Resource Kernels

We now provide a brief overview of the Resource Kernel (RK) approach used in this work. A resource kernel is a system 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. As a result, this *resource-centric* approach can be implemented with any of several different resource management schemes. A reservation represents a portion of a shared resource like CPU, network bandwidth, or disk bandwidth. In [13], Oikawa et al. describe how the resource kernel uses resource reservations as the core abstraction to implement both access control and enforcement. Since an application is expected to use more than one of these resources, reservations on different resources such as the CPU, disk and network are aggregated into *resource sets*, which can then act as virtual machines with guaranteed and enforced timing properties. One or more application tasks can be bound to a resource set. Four mechanisms ensure both that the applications bound to a resource set receive the amount of resource specified in the reservation, and that they do not consume more than this amount: admission control, scheduling, enforcement, and accounting. **Admission control** checks the schedulability of a reserve and that of existing reserves whenever a reserve is created; **scheduling** implements the dynamic allocation of the resources to give the tasks their reserved amount; **enforcement** limits the consumption of the resources to the reserved quantity, and **accounting** supports the scheduler and enforcement mechanisms.

A reserve on any time-multiplexed resource is expressed as a tuple (C, T, D), where C represents the computation time guaranteed every recurring interval (period) T and the deadline D specifies the time, after the beginning of the period, in which the computation time C should be available to the task.

In the rest of this paper, we will use the terms "reservation" and "reserve" interchangeably. At each replenishment time of a reserve R_i, $rep_{i,j} = jT_i, j = 0, 1, 2, ...$, the available execution time is replenished and the reserve is considered eligible for execution. As an attached task executes, the reserved execution time is correspondingly decreased. When the available execution time reaches zero, the reserve is said to have been *depleted*. A reserve that is not depleted is referred to as an *undepleted reserve*.

The implicit assumption is that if the reserve is eligible to execute, task(s) associated with the reserve will execute. Therefore, if a task blocks (e.g. on a mutex), this assumption is violated and the computation time within that periodic interval cannot be guaranteed. In this case, we need a resource access protocol to coordinate the reservation mechanisms and the blocking of the tasks waiting to access resources in a mutually exclusive fashion.

The rest of this paper is organized as follows. Section 2 introduces the resource-sharing problem in reservation-based systems and presents the notion of different degrees of temporal isolation under resource-sharing conditions. Section 3 describes the inheritance and PCP protocols we study in detail, and compares them for theoretical efficacy. Section 4 presents an experimental evaluation of an implementation of our proposed solutions in Linux/RK. Finally, Section 5 presents our concluding remarks.

2. The Problem

In this section, we will characterize the requirements and objectives of resource-sharing protocols in reservation-based systems.

2.1. Resource Policy Decisions and Metrics

Consider a shared resource R accessed by multiple tasks in a reservation-based system like the Resource Kernel (RK). Since R is a non-preemptable resource, a task trying to access R may block when R is being used by another task. In priority-based systems, real-time synchronization protocols such as the basic priority inheritance protocol, priority ceiling protocol and SRP determine the scheduling priority at which tasks accessing resources get to execute. This priority assignment generally leads to bounded durations of priority inversion[1], which in turn can be used in schedulability analysis.

In a reservation-based system, three distinct decisions must be taken when R is being accessed by a task:

1. **Prioritization policy**: What (relative) priority is used by the task accessing R (and under what conditions)?
2. **Charging policy**: Which reservation(s), if any, gets charged?
3. **Enforcement policy**: What happens when the reservation being charged by the charging policy gets depleted?

It is natural to expect that the prioritization policy in reservation-based systems will be the same as that used in synchronization protocols in priority-based systems. The charging and the enforcement policies, however, offer multiple possibilities with different consequences. Then, different choices for the charging and enforcement policies will be compared using the following metrics:

- **Degree of Temporal Isolation**: What happens when a task mis-behaves and violates a timing assumption?
- **Schedulability Penalty**: What is the schedulability penalty, if any, incurred relative to a pure priority-based policy? In other words, will the new policy reject tasks as being unschedulable when they will be "admitted" in an analogous priority-based system?

An *ideal* synchronization protocol in a reservation-based system will have zero additional scheduling penalty relative to a priority-based synchronization policy and ensure that *only* the task that mis-behaves will suffer.

[1] To bound priority inversion, basic priority inheritance may require the use of a deadlock avoidance scheme (like partial ordering of resource accesses).

2.2. Degrees of Temporal Isolation

We now present some metrics and terminology, which can be used to compare and evaluate our resource-sharing schemes.

Definition 1 *A task is said to have* **strict temporal isolation** *if its timing behavior cannot be affected by the timing mis-behavior of any other task.*

As an example of temporal isolation, consider two independent tasks A and B each with its own reservation. If task A (B) mis-behaves temporally, it cannot affect task B (A).

Definition 2 *Conversely, a task is said to have* **non-strict temporal isolation** *if its timing behavior can be affected by the timing mis-behavior of another task.*

Let us now consider two tasks C and D which share a logical resource M which is accessed in mutually exclusive fashion. Suppose that task C locks M and then goes into an infinite loop and thereby never unlocks M. Naturally, when task D needs to access M, its timing behavior will be affected, perhaps significantly. In other words, when mutually exclusive and non-preemptable resource-sharing needs exist, reservation-based systems can hope to provide only non-strict temporal isolation between tasks[2].

Now, consider another task E which does not share any resources with C or D. In some schemes, the mis-behavior of task C can affect the timing behavior of E, but not in others. We next present terms to distinguish between such schemes.

Definition 3 *A task τ is said to be* **temporally contained** *if its timing behavior cannot affect tasks with which τ does not share any resources with.*

Definition 4 *A resource-management scheme Υ_1 is said to provide* **stronger temporal isolation** *than another scheme Υ_2 with respect to a task τ if τ will experience shorter delays under Υ_1 than under Υ_2 when one or more other tasks mis-behave. Correspondingly, Υ_2 is said to provide* **weaker temporal isolation** *than Υ_1.*

Based on the above definitions, we can define the following requirements for reservation-based systems[3]:

1. *Isolation Requirement*: Ideally, reservation-based systems must provide strict temporal isolation for all application tasks.

[2] In this context, it would be interesting and useful to study the temporal protection properties of lock-free protocols but the topic is beyond the scope of this paper.

[3] It is worth bearing in mind that the complexity of system implementation is not accounted for in these requirements, primarily because it cannot be quantified.

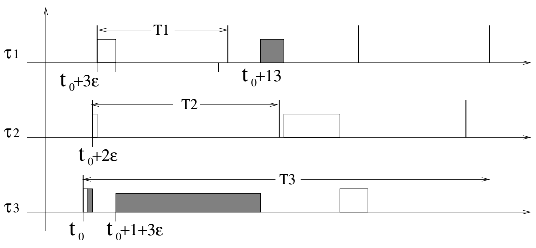

Figure 1. Lack of enforcement on priority inversion Time.

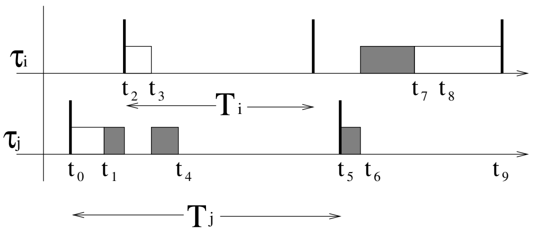

Figure 2. Enforcement during resource access.

2. *Containment Requirement*: All tasks must be temporally contained.

3. *Strength Requirement*: Stronger temporal isolation is preferable to weaker temporal isolation.

As stated earlier, the isolation requirement can only be satisfied for tasks that do not share any resources. In *all* the examples and schemes that we discuss in this paper, the containment requirement will be satisfied. We shall however discuss multiple schemes with different degrees of strength.

2.3. Priority Inheritance in Reservation Systems

We first motivate the need for using the right combination of prioritization, enforcement and charging policies. We do this by illustrating the impact of the wrong combination of prioritization and enforcement policies, each of which may look reasonable by itself. As we shall see, this combination will lead to relatively weak temporal isolation.

Let us consider a real-time synchronization protocol derived from the basic priority inheritance protocol. If a task τ_i blocks on a lower priority task τ_j, τ_j inherits τ_i's priority, otherwise it runs at its own priority corresponding to its reservation. However, τ_j's execution is always charged to τ_j's own reservation. Two effects are of particular interest to us: the **lack of enforcement** on the duration of priority inversion, and the **depletion of a lower priority task's reservation** while in a critical section.

From the temporal isolation point of view, the first problem is critical, because it can affect tasks that do *not* share any resource with τ_j. In other words, the temporal isolation property offered by reservation-based systems can become rather weak, if not completely broken.

As an example, consider the task set presented in Figure 1. This figure shows the effect of the lack of enforcement in the priority inheritance duration. In this case, τ_3 inherits the priority of τ_1, but holds the lock for a long time causing τ_1 to miss its deadline. In addition, τ_2 is also affected by the preemption of τ_3's inherited priority. On the other hand, if an enforcement occurs in the middle of the critical section, the situation illustrated in Figure 2 will occur. In this case, the blocking time of τ_i will be increased until τ_j's reserve is replenished again (at the beginning of the next period) and τ_j finishes its critical section.

These examples highlight the importance of the proper combination of the priority, enforcement and charging policies in synchronization protocols. The discussion of these issues is the focus of this paper.

2.4. Some Possible Solutions

As shown earlier, in order not to break temporal isolation, the code accessing shared resources (*critical section*) must execute attached to an appropriate reservation. Two options are possible:

- A task inherits and executes with the highest-priority non-depleted reserve of itself and the tasks it blocks. This leads to a scheme analogous to *priority inheritance*.
- A special reserve restricted only for execution in the *critical section* is used. This leads to a scheme analogous to *priority ceiling protocol emulation*.

These two approaches are explored in the next section.

3. Resource Access Protocols

In this section, we shall discuss and analyze the reservation analogues of the priority inheritance (PIP) and priority ceiling protocols (PCP). We show that multiple enforcement and charging policies are possible, and that different combinations have different impact on OS complexity, schedulability penalty and degree of temporal isolation.

We will start presenting the analogues to the priority ceiling protocol emulation (or Highest Locker Priority: HLP) given its simplicity. We discuss next the priority-inheritance like schemes that, even though less simple, they can be useful in some situations.

3.1. Reserve Analogue to PCP Emulation

Under PCP emulation, in a priority-based system, every shared resource is assigned a priority ceiling equal to the priority of the highest priority task that accesses that shared resource. A task accessing the resource will then inherit the ceiling priority during its critical section.

A simple priority ceiling protocol emulation policy can be implemented by attaching a common shared-server process to an appropriate reservation (having the priority equal to the priority ceiling of the server). Since the server executes at ceiling priority, there can be at most one outstanding client request at any given time. As a result, the priority inversion encountered by any client task is bounded by the duration of one critical section.

The critical question to be answered in our context is the following: *What should be the size of the reservation that must be assigned to the shared resource server*? The problem can be formulated as follows. Consider m periodic tasks, τ_1, τ_2, $\cdots$, τ_m. Each task τ_i has a period of T_i, $1 \leq i \leq m$, and issues requests for a common server for resource R for C_i^R units of time. The server itself is assigned a period of $T_s = T_1$ (assuming T_1 to be $\min\{T_i\}$), and a computation time of C_s. What is the appropriate value for C_s such that all requests to the common server can be completed on time (i.e. within the deadlines of tasks τ_1, $\cdots$, τ_m)?

We propose the following pessimistic solution to serve as an *initial* starting point that can be further optimized. This solution consists of making $C_s = C_1^R + C_2^R + \cdots + C_m^R$. This pessimistic resource assignment ensures that every client task's requirements are satisfied. But as can be obviously seen, since $T_s = T_1$, the utilization of the server is artificially inflated. The source of the inflation is that for task τ_i, $i > 1$, the term C_i^R/T_i is replaced by C_i^R/T_1, and we have $T_1 \leq T_i$.

The source of the pessimism of the above scheme is that we are constrained to using a single reservation, and the priority of the reservation must be the priority ceiling of the server, forcing the reservation period to be disproportionately small, relative to client task periods.

This problem can be relaxed by associating *multiple* reservations to the server process, with one reservation *per* client task. *Each* of these reservations is parameterized by a computation time of C_i^R, a period of T_i and the shortest deadline of tasks which share the same resource, T_1 in this case. The first two parameters ensure that this scheme is no different from PCP emulation from a system load point of view. The deadline parameter ensures that the server is still assigned a priority equal to the priority ceiling of the server. We refer to this scheme as **multi-reserve PCP** shown in Figure 3.

Admission control for this scheme *must* also be aware of the fact that these reservations are used for resource accesses through a common server, so that the priority inversion that is introduced by at most one of these reservations is accounted for.

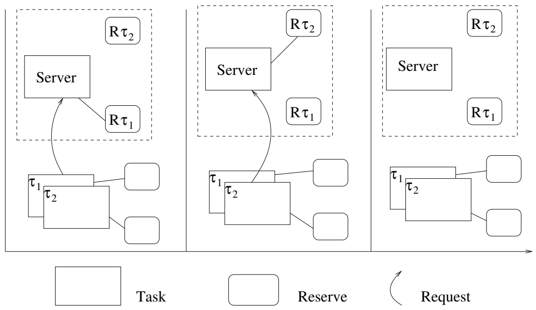

Figure 3. Multi-reserve PCP

3.2. Implications to the OS

The multi-reserve PCP approach implies several new requirements on a reservation-based OS like RK:

1. The server process must be able to switch from one reservation to another reservation depending on which client's request is currently being processed. Hence, client identification information must be communicated along with requests to the server.

2. The operating system must be able to bind new reservations to a process as new clients join the system. Similarly, reservations must be detachable from a server process as clients leave the system.

3. The enforcement of each client reservation at the server side must detect without ambiguity which client's request processing is mis-behaving.

These changes are in addition to those to the admission control scheme which must account for the blocking terms appropriately.

We implemented the "multi-reserve PCP" technique scheme in Linux/RK. Our implementation intercepts the communication between the server and its clients using the unix sockets API. This implementation is based on the "resource control filters" layer implemented by the Resource Control List (RCL) infrastructure [12] in Linux/RK. This infrastructure allows control to be passed to custom kernel modules when certain pre-defined events occur within the kernel. In our case, whenever a server receives a message from a client, it attaches itself to the corresponding "ceiling reserve" of the client. On the other hand, the reserve is detached when the server sends the response back. Three new system calls have been added:

- `rk_cpu_ceiling_reserve_create()` to create a special cpu reserve that is *not* deducted from the schedulable capacity of the processor[4].

[4]This call returns a reserve for use in the next two calls.

- `rk_ceiling_server_attach_process(rsv, server_id, client_id)` takes as its parameters a *ceiling* reserve and the pids of the {server-client} pair for which that reserve should be used. It then automatically activates the attachment and detachment of this ceiling reserve to/from the server when this client uses the server.

- `rk_ceiling_mutex_attach_process(rsv, mutex_id, client_id)` takes as its parameters a *ceiling* reserve, a mutex id and the pid of the client for which that reserve should be used. It then automatically activates the attachment and detachment of this ceiling reserve when this client uses the specified mutex.

Remark: The above scheme works in the presence of multiple servers and/or mutexes with each accessed by different and perhaps overlapping sets of clients. Each server and/or mutex will therefore get its own "ceiling" just as in the case of PCP, HLP and variants. And as in the general case of PCP and HLP, the priority ceiling values can be checked for validity at run-time.

3.3. Priority Inheritance in RK

Priority inversion in real-time systems was extensively analyzed in [17]. If a high-priority task shares resources with a low-priority task and a medium-priority task preempts the low-priority task while in the critical section, the high-priority task can introduce unbounded delays. A family of priority inheritance protocols was introduced [17, 14] to solve this problem in fixed-priority systems.

Using only priority inheritance to overcome this problem ameliorates the waiting time but temporal isolation is very weak (see Figure 1). In fact, a medium-priority task τ_k that does not access the shared resource can experience push-through blocking and can be affected by a task with which it shares no resources other than the CPU. However, under the assumption of knowing the resource access times for each task, this blocking time is bounded, and can be computed as in [14].

3.4. Basic Reserve Inheritance

Under resource reservation, tasks are scheduled according to the parameters of the reserves they are attached to. Basic reserve inheritance is analogous to priority inheritance and allows a blocking task to inherit the reservation of a task that is blocked waiting for it. Basic reserve inheritance is defined as follows:

- Each task can be attached to more than one CPU reserve.

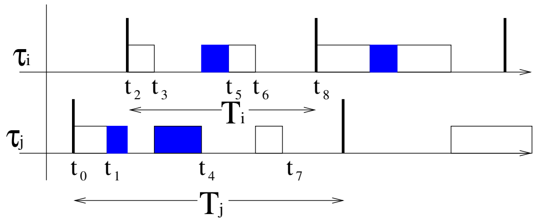

Figure 4. Accessing resources with RI.

- The task is scheduled according to the highest-priority reserve that it is attached to and that is not depleted.

- When a task blocks waiting for a shared resource, its reserve is inherited by the task owning the shared resource. The blocking task then uses the highest priority undepleted reserve for its execution.

- When a task releases a shared resource, any reserve inherited through this shared resource is returned to the original task.

In Figure 4, it is possible to see how reserve inheritance solves the problem shown in Figure 1. As in the previous example, task τ_j starts executing at time t_0 and acquires shared resource R at time t_1. At time t_2, it is preempted by the high-priority task τ_i, that tries to acquire resource R at time t_3. Since R is held by τ_j, τ_i blocks and τ_j inherits its reserve RSV_i. Hence, at time t_3, τ_j restarts executing, and its execution time is charged to RSV_i. In this way, τ_j can use RSV_i execution time (that remained unused in Figure 1) and can terminate its access to R, releasing it at time t_4. At this point, τ_i can restart, and executes until RSV_i is depleted at time t_6. Comparing Figures 1 and 4, it is easy to see how in the reserve inheritance case, τ_i can execute sooner (even if τ_j "steals" some of its reserved time), and the duration of priority inversion is decreased.

3.5. Admission Control under Basic RI

Using priority inheritance, tasks can encounter priority inversion from two sources: *direct blocking* and *push-through blocking*. Direct blocking happens when a task waits for a resource being used by a low-priority task, whereas push-through blocking is caused by preemption from a low-priority task running at a higher inherited priority. Both these times are taken into account in the admission test:[5]

$$\forall j \sum_{i \leq j} \frac{C_i}{T_i} + \frac{B_j}{T_j} \leq j(2^{1/j} - 1) \qquad (1)$$

i.e., B_j can be due to any of the two cases discussed previously.

[5] This admission test is only for illustration. Other exact tests including the fixed-point completion time test, and the 100% utilization bound for harmonic task sets can certainly be used.

In other words, under priority inheritance, the term B_j is deducted from the available processor capacity only when the admission test for task τ_j is considered.

Let us now consider how this blocking term B_j is treated during admission control in a reservation-based system. In RK, a task can be attached to at most one reserve by default. Using this model in its simplest form, under a basic reserve inheritance scheme, one can add B_j to the C_j of the reserve attached to τ_j. The maximum time that τ_j will wait for lower priority tasks will then be "charged" to τ_j's reserve, and τ_j will still be schedulable (assuming that admission control succeeded when the reserve was created). No system support other than the reserve inheritance rules already defined would then be needed. However, when B_j is added to the reserve attached to τ_j, it deducts that amount not only from the available capacity for the admission test of τ_j but also for all other tasks including those that do not suffer any *direct* or *push-through blocking* under priority inheritance. In other words, under the basic reserve inheritance protocol, the admission test changes resulting in a more pessimistic admission control test.

3.6. Eliminating the Schedulability Penalty in RI

Basic reserve inheritance introduces a schedulability penalty because the blocking time B_j is added to C_j and reserved for τ_j, whether priority inversion actually occurs or not. This term then acts as a preemption term for all the lower-priority tasks (instead of as a single instance term in the analysis of the lower-priority task). This schedulability penalty is the price that we pay to address the two problems of lack of enforcement and reservation depletion within a critical section.

These two problems can be addressed by using priority inheritance, and enhancing it with a new mechanism called *blocking-time accounting*. This will ensure that B_i is used *only* to deal with priority inversion. That is, neither the high-priority task τ_i nor the low-priority task τ_j can use this additional reserved time if no priority inversion occurs.

The goal of this new "blocking time accounting" scheme is to perform *priority inversion limit enforcement*. In order to do this, each reservation (C_i, T_i, D_i) is characterized by one more parameter: the *blocking term* B_i, which represents the maximum priority inversion time expected during its reservation period. This scheme is described as follows.

Directly account for the priority inversion time experienced by each periodic instance of a task τ_i and enforce it to be less than or equal to B_i. When task τ_j inherits τ_i's priority, τ_j's execution time is charged to RSV_j *and* to *all* the priority inversions encountered by reserves $R_{j+1}, \ldots, R_i$ of the tasks that are directly or indirectly (push-through) blocked by τ_j. When *any* one of these blocking terms is exhausted (i.e. reaches the maximum bound B_k corresponding to reservation R_k), τ_j is forced to return to its own priority (the priority of RSV_j). Moreover, τ_j will also be enforced when RSV_j is depleted. We refer to this scheme as *PI-PILE* (Priority Inheritance with Priority Inversion Limit Enforcement) and it appears to be the protocol outlined in [15].

In this scheme, when priority inversion occurs, more than one reservation must be "charged". This charging of multiple reservations has the following effects: It prevents extending the execution time of task τ_j beyond its reserved C_j time, and it also prevents τ_j from causing a priority inversion beyond what was accounted for in the admission test. Due to these reasons, the admission test is the same as the one used for basic priority inheritance. However, it is worth noting that the problem highlighted in Figure 2 is reintroduced even though the schedulability penalty is eliminated. In other words, if a B_i term is under-specified, a lower-priority task τ_j may not release a shared resource on time. Consequently, the higher-priority task τ_i may miss the deadline even though it indeed has lots of reserved time left in the current period.

This particular problem of reservation depletion within the critical section can be addressed by reverting back to the basic reserve inheritance protocol *after* priority inversion time enforcement. When the blocking time B_i is depleted, the lower priority task can inherit the CPU reserve of the higher priority one, performing reserve inheritance instead of returning to its original priority. We refer to this hybrid scheme as Priority Inheritance with Priority Inversion Limit Enforcement plus Reserve Inheritance Fall-Through, PIPILE+RI. Under this scheme, the behavior shown in Figure 4 is restored *without* any additional penalty above the regular priority inheritance.

3.7. Comparison

We now first summarize and compare the schemes that are derived from the priority inheritance protocol for accessing shared resources in reservation-based systems.

- **Basic Reserve Inheritance (RI)** The owner of the shared resource inherits the highest-priority undepleted reserve among the tasks blocked on the resource.
- **Priority Inheritance with Priority Inversion Limit Enforcement (PI-PILE)** adds enforcement of the priority inversion time to the basic priority inheritance protocol. To implement this enforcement, the priority inversion time for *each* reservation is tracked at runtime.
- **Priority Inheritance with Priority Inversion Limit Enforcement plus Reserve Inheritance Fall-Through (PIPILE+RI)**: This is based on PIPILE, and

falls back to Reserve Inheritance on priority inversion time enforcement.

We now compare all these proposed schemes along four dimensions: temporal isolation, schedulable utilization, blocking time, and protocol complexity. Table 1 summarizes this comparison, and shows that *no* single scheme is best. However, among these schemes, we recommend the multi-reserve PCP plus Reserve Inheritance Fall-Through (multi-rsv PCP + RI). This scheme combines the advantages of the Priority Ceiling and the RI Fall-Through protocols: the high schedulable utilization and deadlock avoidance properties of PCP and the high degree of temporal isolation obtained by RI Fall-Through. The RI Fall-Through protocol allows the lower-priority task to inherit the reservation of blocked higher-priority tasks when the priority inversion limit is reached. This can consequently reduce any additional delay suffered by higher priority tasks. Comparing RI with Multi-reserve PCP+RI schemes, both have the same degree of temporal isolation but the latter protocol can achieve higher schedulable utilization with somewhat higher protocol complexity.

4. Experimental Evaluation

We implemented the two categories of resource-sharing protocols based on both the basic priority inheritance and priority ceiling protocols in Linux/RK, and tested their effectiveness through a series of experiments.

4.1. Using Reserve Inheritance Policies

We first verified the necessity and the effectiveness of the reserve inheritance policies using a video player. For this purpose, we implemented an MPEG player using the Linux MPEG Library for decoding the video frames and the SVGA Library for displaying them. We ran the MPEG player in a thread τ_1 concurrently with other two threads τ_2 and τ_3 that also acquire the display in mutually exclusive fashion. τ_1 is a periodic thread decoding one frame per period, while threads τ_2 and τ_3 continuously alternate between CPU execution and the "critical section" in which display accesses are executed.

We conducted four experiments on a Pentium Pro 200 Mhz computer with 64 MB of RAM and a cache of 256 KB. In the first experiment, we measured the frame decoding times for thread τ_1 playing a simple MPEG video, obtaining a worst-case C of $12ms$. In order to verify the correctness of this measurement, in a second experiment, we ran τ_1 alone, requiring a frame rate of 20 frames per second (fps), and attaching it to a reserve $RSV_1 = (12ms, 50ms)$. In this second experiment, we measured the inter-frame times $if_j = f_{j+1} - f_j$, where f_j is the time at which τ_1 finishes

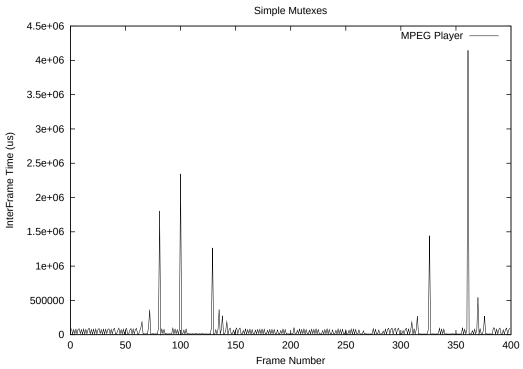

Figure 5. Video player accessing display in mutual exclusion. No inheritance is used.

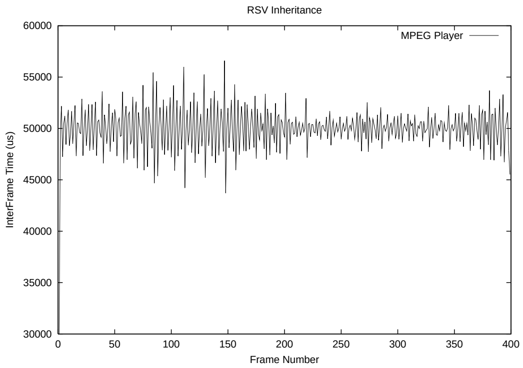

Figure 6. Video Inter-Frame times with RI. Note that, compared to Figure 5, the y-axis range is much smaller.

decoding frame j. In the ideal case, the interframe time should be constant and equal to the thread period ($50ms$), and in fact the inter frame times (not reported due to lack of space) resulted to be quite stable around $50ms$.

In the third experiment, we activated the two threads τ_2 and τ_3 that repeatedly access the video for a time $C_2^R = C_3^R = 2.5ms$ and are attached to two reserves $RSV_2 = (3ms, 60ms)$ and $RSV_3 = (3ms, 60ms)$ respectively. In order to enforce mutual exclusion for display accesses, we used a simple mutex. The results of the experiment are shown in Figure 5: we can see that the high-priority video player suffers the priority inversion problem, experiencing a wide variation in the interframe times. In the fourth experiment, we applied the reserve inheritance protocol to the mutex protecting video accesses, and we dimensioned RSV_1 (see [15]) $C_1' = C_1 + C_2^R = 12 + 2.5 = 14.5$. The results, given in Figure 6, show that reserve inheritance nicely solves the problem encountered in the previous experiment. We repeated the experiment with $RSV_1 = (14.5, 50)$ and *no* reserve inheritance, but there was no improvement with

Scheme	Temporal Isolation	Schedulable Utilization	Blocking Term	Comments
PI	Low	High	$B_i = \sum_k \max_{j<i}(C_j^{R_k})$	No temporal containment
PI-PILE	Medium	High	$B_i = \sum_k \max_{j<i}(C_j^{R_k})$	The higher-priority reserves are unused when the priority inversion time is depleted.
PIPILE+RI	Best	High	$B_i = \sum_k \max_{j<i}(C_j^{R_k})$	High implementation complexity
RI	Best	Low	$C_i'/T_i =$ $(C_i + \sum_k \max_{j<i}(C_j^{R_k}))/T_i$	Simple with effective temporal isolation but high cost (utilization)
Single rsv PCP	High	Medium	$U = \max_k(\sum_i C_i^{R_k})/min_i(T_i)$	U is the utilization of the PCP single rsv
Multi-rsv PCP	Medium	Best	$B_i = \max_{j<i,k}(C_j^{R_k})$	Schedulable utilization is the same as PCP
Multi-rsv PCP+RI	Best	Best	$B_i = \max_{j<i,k}(C_j^{R_k})$	Higher complexity than RI

Table 1. Comparison between different inheritance schemes.

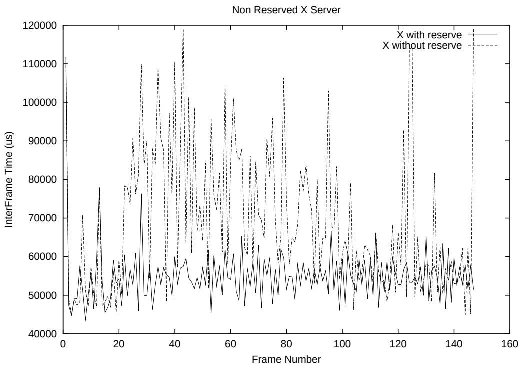

Figure 7. Two video players. X server with & w/o reserve.

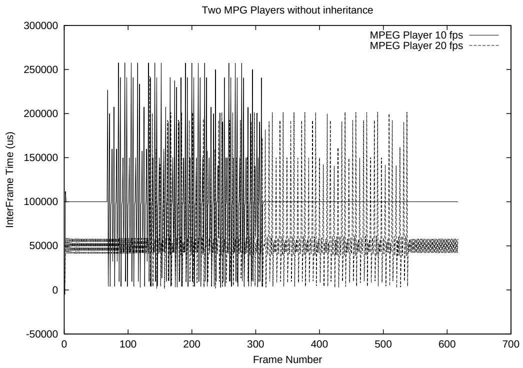

Figure 8. Two video players. X server w/o reserve.

respect to Figure 5. This experiment clearly demonstrates the need to use a reservation inheritance scheme.

4.2. Using The Reservation Ceiling Protocol

We next studied the efficacy of the reservation ceiling protocol implemented with a single reservation assigned to a server to display graphical contents on the screen.

In this second set of experiments, we ran two concurrent MPEG players with periods $T_1 = T_2 = 55ms$, using the standard X server to display the decoded frames on the screen. If the X server is not assigned a reservation of its own, it is not guaranteed to be able to display the frame in time. This problem is highlighted in the plot *"X without reserve"* in Figure 7, that shows the inter-frame times for a single player. As can be seen, the jitter is rather high and the number of deadline misses (not shown) was found to be rather high.

If an appropriate reservation is assigned to the X server in order to emulate a ceiling protocol, this problem is solved, as shown in the plot *"X with reserve"* in Figure 7.

Finally, we ran an experiment for the multiple-reserve PCP on a Pentium III computer running at 645 Mhz with 128 MB of RAM and a cache of 256 KB. We ran two concurrent MPEG players along with a greedy process that consumes CPU cycles. In this case, the players run at 20 and 10 fps (that is, the periods are $T_1 = 50ms$ and $T_2 = 100ms$). We checked the effectiveness of the multiple-reserve PCP by performing two experiments. In the first case, we attached the CPU reserves $RSV_1 = (7ms, 50ms)$ and $RSV_2 = (7ms, 100ms)$ to the two players but we did not use any inheritance. In the second case, we used the same CPU reserves, but we also attached a ceiling reserve to each player. This reserve has a reserved time $C = 5ms$. Figure 8 shows the inter-frame times in the case with no inheritance. In this case, the wide variations in the interframe times is due to the preemption suffered by the X server that in turn blocks the players (waiting for it to display the video frames). In the other case, when we used a ceiling reserve, the inter-frame times remained flat as shown in Figure 9.

5. Conclusions

Resource-sharing in priority-based systems has been well-understood with complete families of protocols de-

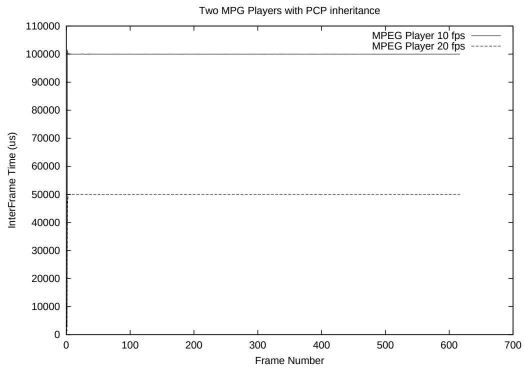

Figure 9. Two video players. X server with Multi-reserve PCP.

signed, analyzed and standardized for use in both fixed-priority and dynamic-priority systems. In this paper, we studied the resource-sharing problem in the context of temporal isolation properties obtained in reservation-based systems. In many cases, due to the need for common OS services such as network protocol stacks, middleware services such as display and database managers, resource sharing cannot be completely avoided. The desirable time-isolation properties offered by reservation-based systems may be affected, perhaps significantly, whenever tasks share resources. We presented two classes of solutions: one derived from the basic priority inheritance protocol and the other from the priority ceiling protocol (and its variants). These schemes vary in the degree of temporal isolation, simplicity of implementation and the schedulability penalty that they incur. We have implemented these schemes in Linux/RK, a resource-reservation version of Linux developed and distributed by the Real Time & Multimedia Systems Laboratory at Carnegie Mellon University.

Experimental evaluations show that our schemes are not only necessary but are also effective in the context of real-time multimedia applications. More detailed studies, both theoretical and empirical, remain to be conducted. In particular, the design of reservation-based operating systems are in their infancy in the context of resource-sharing.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real Time Systems Symposium*, Madrid, Spain, December 1998.

[2] T. P. Baker. A stack-based allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, december 1990.

[3] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London. UK, December 2001.

[4] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2, 1990.

[5] H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[6] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *IEEE Real-Time Systems Symposium*, December 1997.

[7] T. M. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9, 1995.

[8] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, London. UK, December 2001.

[9] G. Lipari. Resource constraints among periodic and aperiodic tasks. Technical Report 9702, RETIS Lab - Scuola Superiore di Studi Universitari e Perfezionamento S. Anna, Via Carducci, 40 - 56100 Pisa, November 1997.

[10] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[11] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems Computing System*, May 1994.

[12] A. Miyoshi and R. Rajkumar. Protecting resources with resource control list. In *The 7th IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

[13] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, June 1999.

[14] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[15] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[16] D. Reed and R. F. (eds.). Nemesis, the kernel – overview, May 1997.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.