Resource Sharing in Reservation-Based Systems

Dionisio de Niz, Saowanee Saewong
Ragunathan (Raj) Rajkumar
Real-Time and Systems Multimedia Laboratory
Carnegie Mellon University Pittsburgh, PA
{dionisio, ssaewong}@cs.cmu.edu, raj@ece.cmu.edu

Luca Abeni
RETIS Lab
Scuola Superiore S. Anna, Pisa
luca@sssup.it

Abstract

In recent years, real-time operating systems began to support the resource reservation paradigm. This technique has proved to be very effective in providing QoS to both, real-time and legacy applications, ensuring that the temporal misbehavior of an application does not affect any other (temporal isolation). However, resource sharing in a reservation system is still not well understood, and can break the temporal isolation property due to priority inversion [3]. In this paper, we address such a problem, presenting some solution strategies that can be considered extensions to Priority Inheritance and Priority Ceiling or Priority Ceiling Protocol emulation for reservation systems.

1. Introduction

Operating systems providing resource reservations, such as the Resource Kernel (RK) in [2], are becoming increasingly popular in real-time and multimedia, since they provide important advantages over traditional priority-driven systems. In particular, resource reservations allow the operating system to monitor and enforce application resource usage and timing requirements, obtaining *temporal isolation* between real-time tasks. This property ensures that the temporal behavior of a task does not depend on the behavior of any other and is as powerful as spatial protection in systems with separate address-space protection. Resource reservations (reserves) are used as a general framework for accessing time-multiplexed resources, grouped in *Resource Sets*. Barely speaking, a reserve permits to guarantee that a specified resource will be available for an amount C each period T with a deadline D.

In the current generation of reservation-based operating systems, resource reservations can generally provide temporal isolation only if application tasks are independent. In practice, many applications do not respect this assumption. For instance, in unix-like systems, multimedia applications use the X server to display graphical content to the screen: in this case, the QoS guarantee can be compromised, due to the well-known priority inversion problem [3].

2. The Problem

The uncontrolled priority inversion problem is well-known in the real-time literature, and has been successfully addressed using priority inheritance techniques [3], that applies to fixed or dynamic priority systems. However, in order to be applied to reservation based systems, those techniques need to be properly extended.

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, which in turn can be used in the schedulability analysis.

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

1. **Prioritization policy**: What (relative) priority is used by the task accessing the shared resource (and under what conditions)?

2. **Charging policy**: Which reservation(s), if any, gets charged?

3. **Enforcement policy**: What happens when the reservations being charged by the charging policy expire?

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, and must be carefully designed in order to provide temporal isolation.

For example, a bad combination of charging/enforcement can break 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, if τ_j execution is always charged to τ_j's own reservation: this causes a lack of enforcement on the duration of priority inversion that can compromise the QoS guaranteed performed by the RK.

3. Solution Strategies

In order not to break temporal isolation, the code accessing shared resources must execute and be attached to an appropriate reservation. It can be the reserve of the task requiring the resource, or a special reserve dedicated to access to the resource.

The first choice ends up in a family of solutions that can be considered as an extension of priority inheritance protocols to reservation schemes. We call them *inheritance techniques*. The second choice corresponds to an extension of the Priority Ceiling or Priority Ceiling Protocol emulation protocol (also known as Highest Locking Priority, HLP) to reservation systems.

Inheritance approaches are based on ensuring that the critical section, currently accessing the resource R, runs with the reserve with the highest priority among those blocked waiting for R and its own reserve. In the PCP/HLP approach, one or more dedicated reservations will be assigned to a common server process that acts as the critical section for a shared resource. Various inheritance protocols, characterized by different charging/enforcement policies, can be used, and we are currently investigating *basic reserve inheritance* and *priority inheritance with priority inversion enforcement*.

Basic reserve inheritance is analogous to priority inheritance. When a high priority task is blocked waiting for a low priority one, the low priority task is allowed not only to execute with the blocked task priority, but also to consume its capacity. With this approach, the blocking time due to sharing a resource with a lower priority task is added to a higher priority task's capacity. This addition decreases the available capacity even for tasks that do not share the resource. Such a penalty is not observed in the traditional priority inheritance.

In contrast, priority inheritance with priority inversion enforcement enhances priority inheritance with a new mechanism called *blocking time accounting*. This will allow to reserve some additional time that is used *only* to deal with priority inversion. That is, no task 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 enforcement*: when a task tries to cause a blocking time greater than the reserved one, it is enforced and blocks, or returns to its original priority. The implementation of this scheme is somewhat more complex due to the need for tracking and updating blocking time but eliminates the schedulability penalty in basic reserve inheritance scheme. Finally the enforcement can use the basic reserve inheritance approach falling back to the highest priority reserve that is available to avoid blocking inside the critical section.

For what concerns the ceiling-like solutions, it is worth noting that a simple priority ceiling protocol emulation policy is directly supported by the normal use of reservations in reservation-based systems and conserves the ability of deadlock prevention and the simpler implementation than priority inheritance. A common shared server process can be given a reservation with a priority assignment equal to what would be the priority ceiling in a fixed priority system. 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 issue to be addressed extending the ceiling approach to reservation based systems is to properly dimension the size of the reservation that must be assigned to the shared resource server. We are currently addressing this problem by using a pessimistic solution to serve as an *initial* starting point and by further optimizing it.

This initial solution creates a reserve with total capacity needed by all tasks sharing the same resource within a deadline equal to the shortest client's deadline. The source of the pessimism of the above scheme is that we are constrained to the use of 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.

The above scheme can be relaxed by associating *multiple* reservations to the server process, with one reservation *per* client task. The server switches from one reservation to another reservation depending on which client's request is currently processing. Each reservation uses the same capacity and period but applies the shortest client's deadline. The enforcement is done individually for each task. With this scheme, the schedulability of the system is no different from PCP (or PCP emulation) but needs to add some ability to the server to track and switch the reserves dynamically. We refer to this scheme as the *multi-reserve server*.

Exact schedulability tests for all the proposed schemes and the proof is shown in [1].

References

[1] D. de Niz, L. Abeni, S. Saewong, and R. R. Rajkumar. Resource sharing in reservation-based systems. Technical report, CMU, 2001.

[2] 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.

[3] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), september 1990.