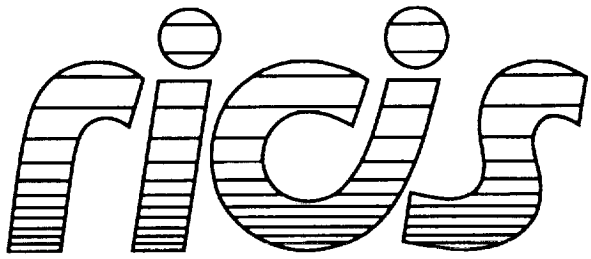
Sources of Unbounded Priority Inversions in Real-Time Systems and a Comparative Study of Possible Solutions

Sadegh Davari
University of Houston-Clear Lake

Lui Sha
Carnegie Mellon University

May 1992

-- RICIS Program Office Research Support --



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

Sources of Unbounded Priority Inversions in Real-Time Systems and a Comparative Study of Possible Solutions

RICIS Preface

This research was conducted by Dr. Sadegh Davari of the Department of Computer Science at the University of Houston-Clear Lake and Dr. Lui Sha of Carnegie Mellon University. The research was supported by the RICIS Program Office and in part by the Software Engineering Institute of Carnegie Mellon University.

RICIS research support funds are derived from Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake.

SOURCES OF UNBOUNDED PRIORITY INVERSIONS IN REAL-TIME SYSTEMS AND A COMPARATIVE STUDY OF POSSIBLE SOLUTIONS[1]

Sadegh Davari
Computer Science Department
University of Houston-Clear Lake
Houston, Texas 77058
Davari@cl.uh.edu

Lui Sha
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
LRS@sei.cmu.edu

ABSTRACT

In the design of real-time systems, tasks are often assigned priorities. Preemptive priority driven schedulers are used to schedule tasks to meet the timing requirements. Priority inversion is the term used to describe the situation when a higher priority task's execution is delayed by lower priority tasks. Priority inversion can occur when there is contention for resources among tasks of different priorities. The duration of priority inversion could be long enough to cause tasks to miss their deadlines. Priority inversion cannot be completely eliminated. However, it is important to identify sources of priority inversion and minimize the duration of priority inversion. In this paper we present a comprehensive review of the problem of and solutions to unbounded priority inversion.

1. INTRODUCTION

The rate-monotonic scheduling (RMS) and the deadline driven scheduling (DDS) algorithms are two well known preemptive priority scheduling algorithms for scheduling tasks in hard real-time systems[1, 2]. In RMS, a periodic task with high rate is given higher priority. In DDS, the earlier the deadline of an instance of a task, the higher is the priority. The priority assignment in RMS is static, meaning priorities are assigned to tasks before execution starts and all the instances of a given task will have the same priority assignment during execution. However, when there is resource conflict, a task is allowed

[1]This work was supported in part by the Research Institute for Computing and Information Systems of UHCL and in part by the Software Engineering Institute of CMU.

to temporarily change its execution priority from the assigned priority. DDS, on the other hand, recomputes the priority of every instance of a task dynamically during execution. Priority inversion degrades the performance of both types of scheduling algorithms.

As is stated in [2], in the schedulability analysis of tasks, each task, within the period of time starting with its arrival and ending with its deadline, must accommodate the worst case of each of the following CPU times:

- the time needed by all higher priority tasks (preemption time)
- the time needed to do the task's own work (execution time)
- the delays caused by lower priority tasks because of priority inversions (blocking time)

Priority inversions occur, when there is contention for shared resources among tasks of different priorities. We would expect that the duration of priority inversion is a function of the duration of critical sections, i.e. the duration in which tasks are using the shared resources. When the duration of priority inversion is not bounded by a function of the duration of critical sections, unbounded priority inversion is said to occur. To improve the performance of real-time systems, we must minimize the duration of priority inversion. In particular, we must identify sources of unbounded priority inversions and eliminate them.

In this paper, we present a comprehensive study of this problem. The rest of this paper is organized as follows: In Section 2 we list common sources of priority inversions. In Section 3 we discuss the known solutions and we conclude this paper in Section 4.

2. COMMON SOURCES OF PRIORITY INVERSIONS

There are two major sources of unbounded priority inversion: task synchronization and communication activities at various levels of computation, from hardware queues to tasking constructions in Ada.

2.1. Semaphores and Critical Sections

Semaphores and the associated critical sections are commonly used synchronization primitives to share resources such as a linked list, a table or a file. The segments of codes in a task that access the shared resources are called critical sections. In order to ensure the integrity of shared resources, critical sections must be executed mutualy exclusively. Semaphores are a common OS primitive that provide indivisible lock and unlock operations. To realize mutual exclusion, before a task enters its critical section, it must

first obtain the lock on the semaphore used to guard the shared resource. Tasks that fail to obtain the lock are typically placed in a queue associated with the semaphore. In some operating systems, the semaphore queue is ordered in FIFO and a prioritized semaphore queue is preferred.

Nevertheless, prioritized semaphore queues alone are insufficient to prevent unbounded priority inversion. For example, let T1 and T3 share a resource and let T1 have a higher priority. Let T2 be an intermediate priority task that does not share any resource with either T1 or T3. Consider the following scenario:

(1) T3 obtains a lock on the semaphore S and enters its critical section to use a shared resource

(2) T1 becomes ready to run and preempts T3. Next, T1 tries to enter its critical section by first trying to lock S. But S is already locked and T1 is blocked and moved from running state to the semaphore queue

(3) T2 becomes ready to run. Since only T2 and T3 are ready to run, T2 preempts T3 while T3 is in its critical section.

When a high priority task like T1 gets blocked by a lower priority task such as T3, we say that the priority of task T1 is inverted. We would prefer that, T1 being the highest priority task, be blocked no longer than the time for T3 to complete its critical section. However, the duration of blocking is, in fact, unpredictable. This is because T3 can be preempted by the medium priority task T2. As a result, task T1 will be blocked until T2 and any other pending tasks of intermediate priority are completed. The duration of priority inversion becomes a function of task execution times and is not bounded by the duration of critical sections. That is, semaphores and critical sections are a potential source of unbounded priority inversions.

2.2 Software Queues

Software queues are often used for communication and data buffering. FIFO queues are obviously a source of priority inversions because high priority tasks get queued up behind lower priority tasks. However, prioritized queues are insufficient to prevent unbounded priority inversion. Assume Ts is a server task that always executes with the priority of its client task. Let T1 and T3 be two client tasks with T1 having a higher priority. Let T2 be a non-client task with an intermediate priority. Consider the following scenario:

(1) Ts is serving T3 with the priority of T3

(2) T1 requests for service and gets blocked in the queue of Ts

(3) T2 preempts Ts, prolonging the blocking time of T1

Since there could be any number of intermediate priority tasks like T2 preempting the server, while T1 is blocked, there is a potential for an unbounded priority inversion.

2.3. Ada Tasking

Ada tasks provide language level support for managing concurrent activities. The synchronization and communication of Ada tasks are provided by a mechanism known as rendezvous. An Ada task performing the function of a server can have one or more entries each of which represents a different type of service. There is a FIFO queue associated with each entry. A client task calls an entry of the server task and gets blocked in the entry queue. Once the server task becomes the highest priority task among all the ready tasks, it will pick an entry that is ready to be served in an arbitrary order. Next, the server executes the accept statement to start performing the service. The server will execute at the highest priority level of client and server. The duration that a server is serving a client is called rendezvous. A client task at the entry queue will be dequeued after it is serviced.

Ada task rendezvous is a potential source of unbounded priority inversion. Assume that T1, T2, and T3 are three client tasks whose priorities are in decreasing order with T1 having the highest priority and T3 having the lowest priority. Let Ts be a server task whose priority is less than the priority of T2. Consider the following scenario:

(1) Ts has accepted an entry call from T3 and is executing the accept statement with the priority of T3

(2) T1 makes an entry call to rendezvous with Ts. Since Ts is not ready to accept the entry call, T1 is blocked on the entry queue

(3) T2 preempts Ts

T1 will be blocked until T2 and any other tasks of intermediate priority complete or block themselves. Therefore, we can have unbounded priority inversions in Ada rendezvous. There are two additional sources for potential priority inversion: the FIFO entry queue and the arbitrarily ordered selective wait statement.

2.4 Hardware Queues

To support message passing over a communication media such as a backplane bus, high speed FIFO hardware queues are commonly used for both the transmission queue and the receiving queue. Messages will be first transferred from slower system memory to

the hardware transmission queue in the Bus Interface Unit (BIU) before the bus arbitration. Messages received by a BIU will be first stored in a receiving hardware queue in the BIU before transferring to the system memory.

At the receiving end, one can use standard high speed FIFO queues, as long as the software can always empty the entire hardware FIFO queue and re-order the messages in priority order before processing. In contrast, on the output side, messages in the high speed hardware buffer should be queued in priority order. However, while software priority queues can be arbitrarily long, but high speed hardware priority queues are typically short due to the cost.

It turns out that a short transmission priority queue can also lead to unbounded priority inversion. Assume that we have a node A with a priority queue of size 4. Assume that the entire queue is filled by lower priority messages first. Next the highest priority message is ready but it cannot be transferred to the BIU since the transmission queue is full. Being the highest priority message, we would expect that it needs not wait more than the duration of a single message transmission.

Unfortunately, unbounded priority inversion can occur. This is because node A will request the bus with the low priority associated with all the low priority messages filling node A's transmission queue. Let node B and C be filled with medium priority messages. Node B and C will preempt node A and send out all the medium priority messages. As a result, the high priority message at A has to wait first for all the medium priority messages to transmit and then the transmission of a low priority message at node A.

3. SOLUTIONS APPROACHES

In the discussion of the previous section we pointed out that synchronization and communication are major sources of unbounded priority inversions in priority driven hard real-time systems. Although priority inversions can never be completely prevented due to resource sharing, there are possible ways of limiting the duration of priority inversions.

3.1. Selectively Disable Task Preemption

From the discussion above, we see that unbounded priority inversion during synchronization happens when a high priority task is blocked by a low priority task, and then the low priority task is preempted by medium priority tasks. Hence, one way to solve the problem is not to let medium priority tasks to preempt a low priority task when the low priority task is in its critical session.

In a uni-processor, this can be achieved by disallowing preemption during the execution of all critical sections. The drawback is that high priority tasks can be blocked by lower

priority tasks even if they are not involved in the synchronization. However, due to its simplicity, it is an effective procedure when the longest critical section is much shorter than the shortest task deadline. Disallowing task preemption can be readily implemented by turning-off interrupts before entering critical section and turning it back on after leaving critical section. The advantage of turning interrupts off and on is that there is no need to call the OS and hence the resulting efficiency. Furthermore, this method can be used by static priority and dynamic priority algorithms alike. However, there is the risk of losing interrupts when interrupts turn-off is not brief.

An improvement to disabling all preemption during the execution of critical sections is a method known as priority ceiling protocol emulation[4, 5]. This method is best explained in the context of static priority scheduling, although it has been generalized into mixed static and dynamic priority scheduling[7]. The idea here is to selectively disable preemption. That is, we make the priority of executing a low priority task's critical section sufficiently high to effectively disable the possible preemption from medium priority tasks.

To implement this method, the highest priority of all the tasks that will lock a semaphore is copied into a field associated with the semaphore. This is called the priority ceiling of a semaphore. When the OS grants a semaphore lock to a task, it also raises the priority of the task to the priority ceiling of the semaphore. When the task makes a call to the OS to unlock the locked semaphore, the OS returns the task to its assigned priority.

Under this protocol, tasks are free from deadlock and a task can be blocked by lower priority tasks at most once as long as tasks do not suspend within their critical sections[4,5]. This same result holds if preemption is disallowed completely. An intuitive explanation of this result is as follows. Since a task is executing at the ceiling priority of a semaphore, no other task that may lock this semaphore can start execution. As a result, there is only one task among the group of tasks that may lock S can be in its critical section at any given time. This makes deadlock impossible, since a necessary condition for deadlock requires that at least two tasks sharing resources be in their critical sections. The fact that only one task among a group of resource sharing tasks can be in its critical section at any given time also leads to the result that a task can be blocked by at most one lower priority task.

This argument will not hold if a task ever suspends itself within a critical section, say, waiting for an I/O call to return. Suppose that a low priority task locks a semaphore S1 and then suspends. Another low priority task can start and lock another semaphore S2. When the high priority task with the need to lock S1 and S2 becomes ready to execute, it has to wait for both the task locking S1 and the task locking S2. Hence the blocked-at-most-once result does not hold. Nor does the no deadlock argument hold when tasks can suspend within their critical sections.

This approach can also be applied to synchronization problem using Ada tasking[4]:

(1) the body of each server task should consist of a selective wait statement within an endless loop

(2) each server task should be given a priority which is higher than the priority of any of its client tasks

(3) the server task should not block itself within the accept statement

The block at most once and no mutual deadlock also holds here[4,5] and the reason is similar to the argument above. When calculating the blocking time of a given task, T, under this solution, we should only be concerned with (1) the resources that task T shares with lower priority tasks, and (2) the resources that are shared by both tasks of lower priority than T and tasks of higher priority than T. The worst case blocking time for T would then be the execution time of the longest critical section among all lower priority tasks that share such resources.

The method of priority ceiling emulation is a simple and effective procedure. It is used by the protected record construction in the current draft of the Ada 9x requirement mapping document[9].

3.2 Priority Inheritance Protocol

In the previous solution, a task's priority is immediately raised when entering a critical section (turning-of preemption is equivalent to raising the priority to the highest level). However, the priority inheritance protocol[4,5] is invoked only when a higher priority task is blocked by a lower priority task. When a lower priority task T blocks the execution of higher priority tasks it inherits the priority of the highest priority task blocked by T. Task T returns to its assigned priority when exiting its critical section.

When a low priority task inherits a high priority, medium priority tasks will have to wait for the execution of the lower priority tasks. Nonetheless, this is worthwhile. The duration of push-through blocking (blocking of medium priority tasks) is in terms of critical sections. Without paying the price of push-through blocking, the high priority task may have to wait for the entire execution time of medium priority tasks.

This solution can be adopted for the problem of unbounded priority inversion resulting from either critical sections or message queues. To adopt this solution for message queues we first need to prioritize the message queue. Secondly, the server will use the message priority. However, when new messages with higher priorities entering the queue, the server must inherit the highest priority of all the new messages. The priority inheritance can also be applied to Ada tasking. First, the entry queues must be prioritized. Secondly, the task must select the highest priority entry. Finally, when higher

priority client tasks entering the entry queue, the server must inherit the highest priority client in the queue.

Although priority inheritance protocol solves the unbounded priority inversion problem, it suffers from the possibilities of "chained blocking" and offers no help to the "mutual deadlock" problem. To avoid mutual deadlock under this solution, we may totally order the sequence of locking shared resources.

The following example illustrates a situation where chained blocking can occur: Assume that tasks T1, T2 and T3 share semaphores S1 and S2 and further T1 has higher priority than T2, while T3 has the lowest priority. Consider the following scenario:

(1) T3 locks S2

(2) T2 locks S1

(3) T1 tries to lock S1 and S2 but now has to wait for both T2 and T3. That is, chained blocking.

The worst case blocking time of a given task T, under this solution, is the sum of the blocking time from each shared resource (because of the possibility of the chained blocking). The worst blocking time from each resource is calculated the same way as it was done for priority ceiling emulation. We should consider (1) the queues that task T shares with lower priority tasks and (2) the queues that tasks of higher priorities share with tasks of lower priorities. The blocking time of task T from each shared resource is calculated to be the longest blocking time caused by any such lower priority task.

The advantage of priority inheritance is that it can be directly applied to both dynamic and static priority scheduling algorithms. In addition, it can prevent unbounded priority inversion for both task synchronization and communication even if tasks suspend during their critical sections. Priority inheritance protocol is currently supported by many Ada vendors and real-time OS vendors. It also appears as an option in the draft of OS standard known as real-time POSIX, IEEE P1003.4a[10] and is permitted by the Draft Ada Requirement Mapping Document[9].

3.3 Priority Ceiling Protocol(PCP)

The priority ceiling protocol[5, 6] can be viewed as a generalization of two solutions above. It eliminates the possibilities of chained blocking and mutual deadlocks, even if tasks suspend within critical sections. The priority ceiling of a semaphore (or a server task) S is simply the priority of the highest priority task that may lock the semaphore (or may call the server) S. This solution can be adopted for the problem of unbounded priority inversion resulting from either semaphores or queues. But, in what follows the

focus of our discussion will be on semaphores in the context of static priority scheduling, although PCP has been extended to dynamic schedulings[8]. PCP has the following rules:

1. A task with a higher execution priority always preempts tasks with lower execution priorities.

2. A task cannot enter its critical section unless its priority is higher than the priority ceilings of all semaphores that have been locked by other tasks.

3. A lower priority task that blocks a higher priority task T inherits the priority of task T.

Rules 1 and 3 of PCP also apply to the priority inheritance protocol. Therefore, the only difference between PCP and the priority inheritance protocol is rule number 2. The idea behind PCP is to create a total ordering of executing and suspended critical sections. This protocol, although more expensive to implement, has all the benefits of the priority inheritance protocol plus it has the "block at most once" property, which prevents chained blocking and mutual deadlock.

To illustrate how mutual deadlock is eliminated, consider the previous example where tasks T1 and T2 share semaphores S1 and S2 and T1 has higher priority than T2. Note that the ceiling priority of both S1 and S2 is the priority of task T1. Let us try to follow the following scenario.

(1) T2 locks S2

(2) T1 tries to lock S1 but fails because its priority is not greater that the ceiling priority of S2 which has been locked by T2

(3) T2 locks S1

(4) T2 first unlocks S1 and then S2

(5) T1 takes over and locks first S1 and then S2

As we can see PCP eliminates the possibility of deadlock. A similar scenario would show that PCP also eliminates the possibility of chained blocking. The main drawback of this protocol is that it is relatively complex to implement.

3.4 Hardware Priority Queue with Overwrite

We now address the problem of unbounded priority inversion problem associated with hardware transmission queues. None of the above methods are effective for this

hardware transmission queue problem[3]. A practical solution is the use of a short priority queue with priority overwrite to emulate an ideal priority queue[3]. When the BIU transmission queue gets filled up and a higher priority message waits at the host, the higher priority message overwrites the lowest priority message in the queue.

This overwrite is carried out as follows. First, each message queued for transmission is held in system memory until the message has been successfully transmitted. In addition, the software remembers the messages transferred to the transmission queue. When a new message arrives at the system queue and the transmission queue is full, the software compares its priority with the lowest priority message in the transmission queue. If the new message has higher priority, then it replaces the lowest priority message in the transmission queue. Finally, we want to point out that the overwrite does not affect the performance of transmission queue since with proper hardware support the BIU can send the high priority message in the queue while its lowest priority message in queue is concurrently being overwritten[3].

4. CONCLUSION

Task synchronization and communication are two common sources of priority inversion. Priority inversion degrades the performance of real-time scheduling algorithms and should be minimized. In particular, it is important to identify the sources of unbounded priority inversion and eliminate them.

In this paper, we have reviewed the common sources of priority inversion. We also reviewed four different solutions to the unbound priority inversion problem, namely, selectively disabling preemption, priority inheritance protocol, priority ceiling protocol and hardware priority queue with overwrite. There are strength and weakness in each of these solutions and users must choose them according to their application needs.

REFERENCES

[1] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real Time Environments," J. ACM, Vol. 20, No. 1, 1973, pp. 46-61.

[2] L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada," IEEE Computer, April 1990, pp. 53-62.

[3] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Real-Time Computing Using IEEE Future Bus+," IEEE Micro, June 1991.

[4] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," IEEE Transaction on Computer, Sept. 1990.

[5] J.B. Goodenough and L.Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High priority Ada Tasks," Ada Letters, Special Issue: Proc. 2nd Int'l Workshop in Real-Time Ada Issues VIII, Vol. 7, Fall 1988, pp. 20-31.

[6] R. Rajkumar, L. Sha, and J.P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," Proc. IEEE Real-Time Systems Symposium, 1988, pp. 259-269.

[7] T. Baker, "Stack Based Scheduling of Real-Time Resources", Technical Report, Department of Computer Science, Florida State University, Tallahassee, FL 32306, April 1990.

[8] M. I. Chen and K. J. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems.", Technical Report, Department of Computer Science, University of Illinoise at Urbana-Champaign, 1989.

[9] Draft Ada 9x Project Report: Ada 9x Mapping Document Vol. II, December 1991.

[10] Tread Extension to Portable Operating System Standard, IEEE P1003.4a, Draft 5, IEEE 1990.