Priority Inversion with Fungible Resources

Gertrude Levine
Fairleigh Dickinson University
1000 River Road
Teaneck, NJ 0766
levine@fdu.edu

ABSTRACT
Priority inversion occurs when the execution of a task is unnecessarily delayed by the dispatch of a lower priority task. This anomaly can result in failure if it causes the delay of hard real-time tasks. Priority inheritance protocols have been developed to limit priority inversions during competition over shared resources. Such methods are designed for individually identifiable resource units. Other approaches are indicated when shared resources contain interchangeable resource units. Current practices with resource deadlock, a scheduling anomaly that has many characteristics in common with priority inversion, provide insight into the control of priority inversion with these fungible resources.

Keywords
Priority inversion, deadlock, priority inheritance

1. INTRODUCTION
Priority inversion is a scheduling anomaly containing tasks of different priorities such that the execution of a higher priority task is delayed by the dispatch of a lower priority task either because of

a) the "needless" [3] lowering of the effective priority of the higher priority task or
b) the "needless" raising of the effective priority of the lower priority task.

"It should be possible to program the following two assertions about the interactions between a set of client tasks and a server task. First, when the highest priority client requests service by calling a server's entry, the maximum delay until the server starts the rendezvous (the blocking time) should be whatever time it takes the server to finish with an existing client, if any. Second, when the highest priority client is not requesting service from the server, it should not be possible for the server to delay that highest priority client. It should also be possible to apply these two assertions to lower priority clients provided we allow for preemption by higher priority clients, or by the server on behalf of higher priority clients. Violation of either of these assertions leads to *priority inversion,* a condition where low priority tasks can needlessly block a high priority task, possibly for an indefinite period of time." [3]

The assertions cited above were formulated for competition synchronization using the tasking construct of the Ada83 programming language. We conclude that priority inversion does not result from "needed" actions, such as blockage from a resource held by a lower priority task that is executing within its critical section. Interleaved access to such a resource must be prevented in order to ensure resource consistency. Similarly, an inversion does not result from the delay of an intermediate priority task by the dispatch of a low priority task that is rendering "needed" service on behalf of a high priority task.

2. DEFINITIONS AND RELATED WORK
Some of the computer science literature states that priority inversion is the blockage of a higher priority task by a lower priority task [17], using a shared resource [6] and/or for an unbounded or indefinite period of time [2, 17]. These definitions, however, are too general and lead to inconsistencies.

1) Priority inversion is a scheduling anomaly [1]. When a low priority task is scheduled, it generally cannot be anticipated that it will lock a resource during the time that the resource is requested by a high priority task. Thus blockage of the high priority task from the shared resource following the scheduling of the low priority task is not an anomaly. On the other hand, an inversion does occur if an intermediate priority task is scheduled to preempt the processor from a low priority task that is within its critical section and blocking a high priority task. The high priority task's effective priority has been lowered to that of the blocking task, causing its needless delay by the premature preemptive dispatch of the intermediate priority task.
2) A task within a critical section must block any task that requests its non-concurrently shareable resource, even if the waiting task has higher priority, since interleaved access to critical sections can cause failures. This blockage is necessary for service requirements. Interrupt handlers (Interrupt Service Routines) delay any higher priority ISRs that become active while they are saving the states of shared registers, yet these needed delays are never characterized as priority inversions.
3) The priority inheritance protocol (PIP) is said to prevent priority inversion [12] or uncontrolled priority inversion [17]. When a high priority task requests a locked resource, PIP temporarily raises the priority of blocking tasks to prevent the lowering of the effective priority of the high priority task. Yet PIP does not address the blockage of a high priority task that becomes ready for execution after a lower priority task has obtained a resource that it requests. Indeed, the high priority task may be blocked by an entire blocking (perhaps cyclical) chain of tasks, each of which is waiting for another task that is executing within a critical section. PIP

does not prevent blocking chains, even those that delay a high priority task past its deadline. PIP does prevent preemption by intermediate priority tasks, however, and thus controls some priority inversion scenarios [14].

4) Assume that a blocking chain of tasks exists, in which PIP has raised the priority of all tasks in the chain to that of a high priority task at the end of the chain. If an independent intermediate priority task becomes ready for execution, it will be dispatched after the low priority tasks, whose priorities have been temporarily raised. Yet this is not a priority inversion, even if the intermediate priority task has hard real-time requirements. The prioritized executions of the low priority tasks are "needed" to hasten the execution of the highest priority task.

   Assume that PIP has raised the priority of low priority tasks in a blocking chain to that of a high priority task at the end of the chain. An intermediate priority task is then prevented from preempting the (previously) low priority tasks because of their raised priorities. Assume that the high priority task aborts, yet a network message to reset the priorities of the low priority tasks is delayed [9]. The low priority tasks are then dispatched before the intermediate priority task, although their effective priority was needlessly raised – a clear inversion.

5) A definition of priority inheritance as the blockage of a high priority task from a resource held by a low priority task is particularly inadequate when resources are fungible, i.e., interchangeable in terms of client services (such as real memory). A high priority task can be blocked by multiple lower priority tasks, each holding some units of the resource that it requests. Most of the holding tasks need not unblock their resource units for the high priority task and their scheduling cannot be considered an anomaly.

6) Priority inversions can occur during "synchronous inter-thread communication" [18]. For example, a high priority task that is waiting for a signal from or about a low priority task will be continuously delayed if newly arriving intermediate priority tasks are scheduled to the processor. The clause "using a shared resource" in the traditional definition of priority inversion does not accommodate inversions caused by cooperation mechanisms.

7) Priority inversion can fatally damage a required service with a hard deadline, even when delays are bounded. Indefinite delay degradations [11] involving tasks without real-time constraints or essential services, however, are tolerated by many system requirements. For such systems, the delay caused by prevention or recovery mechanisms may be greater than any gains to be achieved.

3. SCHEDULING AND BLOCKING

Priority inversion is a scheduling anomaly. Scheduling determines the order in which processors are delegated to ready tasks during periods of competition for the processor(s), using priorities to further system goals. Priority assignments assist a system in meeting deadlines by providing preferred access to processors and other resources. Priorities are adjusted dynamically by schedulers to promote secondary objectives, such as fairness. Priorities are also adjusted dynamically by priority inheritance protocols for the control of some types of priority inversion [14, 17] and deadlock [2, 17].

Preemptive scheduling supports prioritized execution by allowing a high priority task to usurp the processor from a lower priority task. Such scheduling assists hard real-time tasks in meeting their deadlines, but can cause priority inversion. For example, assume that a higher priority task is waiting for a resource held by, or for an event to be achieved by, a low priority task. If intermediate priority tasks preempt the processor from the low priority task, the higher priority task can be delayed indefinitely. Non-preemptive execution, obtained perhaps by disabling interrupts while a task is within a critical section, prevents this type of priority inversion, but will cause the blockage of a newly ready, independent, highest priority task. Priority inheritance is more effective for this scenario, by dynamically adjusting priorities during contention.

Blocking can be caused by competition synchronization. Assume that tasks are independent, synchronizing only when competing for resources that must be used in mutual exclusion. Competition synchronization mechanisms guarantee consistent resource usage. Java threads, for example, lock shared data while within a critical section. Threads that need locked data are blocked, typically on a prioritized queue, waiting for resources to be freed. Sometimes a task requests another resource while it is inside its critical section. This second resource may be locked by another task that is also blocked waiting for a resource. Such a "transitive" [17] blocking chain causes neither needless blocking nor priority inversion, according to our definition. Yet a high priority task that accesses the last resource of the chain is (directly or indirectly) blocked by the other tasks in the chain. Its effective priority has been lowered to the lowest priority of the blocking tasks. Priority inheritance protocols raise the priority of all of the blocking tasks to that of the high priority task. If an intermediate priority task becomes ready for execution, it will have to wait for the execution of the critical sections of all of the tasks in the blocking chain, even if it does not request any resource. The Priority Ceiling Protocol (PCP) [17], the Priority Ceiling Emulation Protocol [12], and the Ceiling Locking Protocol (CLP) [2] are priority inheritance protocols that assign a ceiling priority to each resource, based on the highest priority of the statically determined tasks that may access it. Using PCP, a task's priority is raised only when it is blocking a higher priority task from a resource (as in the PIP). The PCP also prevents a task from obtaining a resource unless its priority is higher than the ceiling priority of all resources that are currently held by other tasks. This mechanism restricts blocking chains and resource deadlock, but also hampers concurrency. For example, a task with a lower priority than currently held resources cannot obtain an uncontested resource. Maintenance overhead also makes this protocol unpopular, specifically in distributed systems [12]. The CLP has been implemented in Ada environments, but, to reduce implementation costs, the protocol raises the priority of a task within a critical section to the resource's ceiling priority whether or not another task is blocked on the resource. Such restrictions can lead to inversions, but are effective when critical sections are minimized [2].

Blocking can also be caused by cooperation mechanisms. Assume that tasks are dependent upon events of other tasks, such as a high priority parent process that is waiting for a signal of termination from a low priority child process or a high priority Java thread that is waiting for a low priority thread to reach a barrier for garbage collection [15]. If the child process or the low priority Java thread is continuously preempted by independently executing, intermediate priority entities, it will indefinitely block the high priority entities. These priority inversions are similar to

inversions with shared resources. The delay of a high priority entity that waits for signals from low priority entities, however, is NOT an inversion.

It is important to distinguish between the above two categories of priority inversions since they are characterized by differences in controlling mechanisms. In competition synchronization, interleaved access to shared resources must be prevented. Critical sections of code are protected with competition mechanisms, such as servers, monitors, semaphores, or spinlocks. If tasks can be rolled back, locked resources can be freed. For example, a resource being held by a low priority Java thread that is blocking a high priority thread can be preempted [18], assuming consistency is maintained and time restrictions remain satisfied during roll back and recovery. Or a database transaction that is blocking a higher priority transaction can be aborted and restarted [6].

If blockages are caused by cooperation synchronization, on the other hand, and a high priority task is delayed by a lower priority task upon whose service it is dependent, it is counterproductive to roll back the blocking task. Nor are priority inheritance protocols applicable to priority inversion with cooperation mechanisms.

4. PRIORITY INVERSION AND RESOURCE DEADLOCK

A resource deadlock is an infinite waiting condition involving mutually independent tasks, each of which is blocked from a non-preemptible resource by competition mechanisms while holding a resource that is requested by another task in the set. Cooperation mechanisms ensure that each task holds its locked resource(s) until it obtains service from the resource(s) from which it is blocked. Resource deadlock is characterized by interleaved task execution during competition over resource acquisition [10]. This paper uses the corresponding term, **resource priority inversion,** for priority inversions occurring during competition over resource acquisition.

Communication deadlock is an infinite waiting condition involving mutually dependent tasks, each of which is blocked by cooperation mechanisms as they wait for signals of events from other tasks in the set [10]. This paper uses the term **communication priority inversion** for inversion among mutually dependent tasks during which blockage is caused by cooperation mechanisms. We stress that communication priority inversion is a scheduling anomaly but communication deadlock is NOT. Yet, like communication deadlock, communication priority inversion cannot be managed by schemes that structure resources (excluding the processor). Control mechanisms for cooperation synchronization are more complex than mutual exclusion mechanisms.

Both resource deadlock and resource priority inversion are caused by faults that sometimes remain dormant during development, testing, and repeated system execution [8, 14]. As a precondition for both anomalies, a task requests a resource that must be used in mutual exclusion and waits if the resource is locked by another task. If resultant delays are unacceptable, the waiting task fails. If the task's service is essential, the system will fail. When services are delay tolerant, priority inversions and resource deadlock can be handled with detection and recovery mechanisms. Hard real-time tasks are time-sensitive, however, so that prevention methods are more urgent for the control of priority inversion.

Both resource deadlock and resource priority inversion occur only if tasks wait for a locked resource. Thus, these anomalies cannot occur with concurrently shareable resources, such as read-only memories. They are also prevented if tasks do not wait for a blocked resource, perhaps switching to an alternate resource or abandoning that service altogether. Alternatively, resources may be preempted if detection and recovery are not too onerous [5, 18]. For example, to prevent resource deadlock in databases, transactions are assigned linearly ordered priorities, based partially on access times. "Younger" transactions that hold a resource requested by older transactions are killed (or only wounded if deemed able to complete). Younger transactions that request a resource held by older transactions are allowed to wait. [16] Since database systems using two-phase locking alter system states only following a "commit," transactions can be aborted and their temporary workspaces cleared. Similarly, to control resource priority inversion, database systems order transactions by real-time priorities. If a transaction is blocked from a resource, by a lower priority transaction, the blocking transaction is wounded, or killed with its temporary workspace flushed. A transaction that is blocked by a higher priority transaction is allowed to wait [6]. In order to implement such a scheme effectively, however, systems have to be designed so that the overhead involved in preempting blocking chains does not cause unacceptable waits [6]. Similarly in Java programming environments, previous states can be saved, enabling roll back of threads and reordering of resource access by thread priorities [18]. Hard real-time constraints on threads, however, make them vulnerable to roll backs of cascades of dependent threads [6]. If any of the dependent tasks are of high priority, restarts can cause system failure [18].

Resource pre-allocation is an effective mechanism, under limited conditions, for the prevention of resource deadlock. Tasks pre-state their resource needs, which are allocated as a unit, so that tasks do not wait for resources after the initial acquisition. For example, some operating systems pre-allocate real memory partitions to tasks. Pre-allocation is also effective for restricting the involvement of low priority tasks in blocking chains, a vital issue in resource priority inversion. Suppose pre-allocation is applied selectively to low priority tasks that share resources with high priority tasks. Before a low priority task obtains these shared resources, it sets a bi-level lock. The inner level lock (or semaphore) is applied when the low priority task is in its critical section and prevents preemption by all tasks. The outer level lock signals that the resource is not currently in use and is tested by low priority tasks only. High priority tasks bypass outer level locks, allowing them to preempt idle resources; such resources are released back to previous owners by high priority tasks. (Multiple level locks can be used as needed.) Resource pre-allocation restricts concurrency, denying others access to pre-allocated resources even when they are idle. Static declaration of resource requirements also prevents use of dynamic resources. In addition, tasks that have high resource needs can be denied their resources for an unbounded period of time. Yet, priority ceiling protocols also restrict concurrency, require static declaration of resource needs, and can delay tasks indefinitely. Unlike resource pre-allocation, however, priority ceiling protocols can (unnecessarily) block non-competing tasks from obtaining resources. Unlike priority ceiling protocols, the restricted application of resource pre-allocation to only low priority tasks allows high priority tasks to create and request resources dynamically.

A time-out is a heuristic that is in common use for deadlock control. A service that cannot be completed by a specified time is either restarted or abandoned. Restarting a task is only effective for priority inversion if tasks are sufficiently delay-tolerant [6, 18]. Abandoning a service after it has exceeded its maximum allocated time is a common alternative [14].

5. FUNGIBLE RESOURCES

The chief mechanism for hardware fault tolerance is redundancy. Duplicate components are installed to enable recovery from fault activation; these improve performance when faults are dormant and enable degraded service when faults are activated. Fungible resources, resources that provide service of equal value to clients, such as memory, transmission channels, buffers, etc. are the most effective duplicates. Systems have averted resource deadlocks with fungible resources by adding sufficient additional units; this mechanism is effective for priority inversion with fungible resources as well. When resources units are limited, other methods must be sought.

As a simple example of priority inversion with fungible resources, consider a preemptive scheduling system with 10 units of resource A. Task 5 with priority 5 is executing independently. Task 2, task 3, and task 4 each has a priority of 10 and holds units of A: 1 unit for task 2, 3 units for task 3, and 6 units for task 4. Within its critical section manipulating the allocated units, each task has requested resources held by task 21, task 31, and task 41 respectively; each of these tasks has a low priority of 20. Task 1, with a high priority of 1, becomes ready for execution, requests 5 units of A, and is blocked. Assuming that the system uses priority inheritance, the priorities of tasks 2, 3, 4, 21, 31, and 41 are raised to 1. (Table 1)

Table 1. Allocation of fungible resources

	Priority	Resources held	Resources needed	Scheduling order
Task 5	5			1
Task 1	1		5A	8
Task 2	10/1	1 A	B	5
Task 3	10/1	3 A	C	6
Task 4	10/1	6 A	D	7
Task 21	20/1	B		2
Task 31	20/1	C		3
Task 41	20/1	D		4

The scheduler chooses which of ready tasks 21, 31, and 41 to preempt task 5; it chooses 21, which executes its critical section and unblocks task 2. Task 31 is chosen next, preempting task 21 and unblocking task 3. Task 41 is scheduled next, followed by task 2, then task 3, each preempting the previous task when it completes its critical section and has its priority lowered. None of these release enough units of A to unblock task 1. At last task 4 is executed and enough units of resource A are released to unblock task 1, which should be able to execute except that it has missed its deadline. Task 5 was preempted by task 21 needlessly, and task 31, task 3, and task 2 were given priority over task 5 needlessly. But how could the scheduler determine this? Must it keep track of the number of resources that each task is holding and compare that value to task 1's request? This scenario becomes more complicated when resources B, C, and D are also fungible, with different units locked by other tasks. Assuming no cycles exist, the blocking structure is no longer a chain, but a tree. There are implementation problems in preventing priority inversion with blocking chains. Delays resulting from handling blocking trees are less tolerable. Deadlock prevention schemes must control similar blocking scenarios.

The above computations resemble safe state algorithms that prevent resource deadlocks with any combination of fungible and uniquely identifiable resources. Their implementations are rare in operational systems because of their overhead and restrictiveness. Heuristics for maintaining safe states are preferred. For example, in virtual memory systems, operating systems periodically identify the number of free memory frames. An operating system assumes that (perhaps) 70% of memory is sufficient for resident tasks to maintain their working sets, allowing them to continue service relatively efficiently. If the percentage of frames in the "free pool" falls below 30% of total memory, an OS task chooses "victims" from pages resident in the 70% pool. These are not overwritten immediately. Nor are other "free" frames left vacant; they are filled with pages of background tasks, pages waiting to be copied out, as well as pages brought from the swap area by anticipatory fetching. When a page fault occurs, one of these pages is overwritten. Preemption of frames prevents resource deadlock, but too frequent preemptions cause page thrashing and starvation.

Apply the above discussion to the control of priority inversion with fungible resources. Assume that the system has obtained sufficient resources units, on average, of each fungible resource type used by high priority tasks, as determined by static analysis. At least 70% of the units of each bestowed type are reserved in a high priority resource pool, for the establishment of "working sets" of resources for high priority tasks. The operating system monitors the use of the high priority resource pools. If the percentage of free units in a high priority pool falls below 30%, units are taken from the corresponding low priority pool as they become available and added to the high priority pool. Any preemptible units assigned to low priority tasks can be counted as free units in the high priority pool. All units in the now enlarged high priority pool are reserved for high priority tasks until its percentage of free units is greater than 30%. Depending on traffic patterns of resource requests, this heuristic can provide sufficient availability to fungible resources for high priority tasks so that they rarely block, forestalling resource priority inversion and resource deadlock. Increased delays for low priority tasks will be tolerable. The above discussion takes advantage of the fact that hard real-time systems are typically resource heavy.

In order to develop an overall strategy for the control of priority inversion, consider how resource deadlock is handled. Although resource deadlock can be prevented, any single mechanism is considered too time consuming and/or restrictive. Modern dynamic systems control resource deadlock with combinations of prevention and recovery schemes. First, the system is designed with deadlock control as part of the requirements. Whenever possible, resources are made concurrently shareable or preemptible. Critical sections are minimized to reduce blocking times. Uniquely identifiable resources are structured into groups with linear orderings differentiating the groups. Sometimes fungible resources are differentiated through an ordering, restricting fungibility but assisting deadlock prevention. Threads

and resources that are created dynamically are monitored by deadlock detection daemons that continuously analyze blocking relations for cycles. Snapshots and breakpoints enable system rollback if deadlock is suspected. Resource deadlocks with fungible resources are handled with heuristics and preemption.

Priority scheduling for hard real-time systems also requires careful system analysis and design [7]. Whenever possible, resources are made concurrently shareable or preemptible. Static analysis determines priorities for scheduling access. Systems avoid the sharing of resources by hard real-time tasks and low priority tasks. Of specific importance to the control of resource priority inversion is a restriction on blocking chains, perhaps requiring low priority tasks to use resource pre-allocation methods. Critical sections are minimized [2], perhaps by placing computation in modules separate from critical sections and using non-blocking buffers for communication between modules [7]. Multi-level allocation pools can assist in the control of priority inversion with fungible resources during non-uniform traffic conditions.

Multiprocessing systems can eliminate priority inversion entirely (according to our definition of priority inversion). Sufficient fungible resource units eliminate the wait for resources that is a requisite for resource priority inversion and resource deadlock. Similarly sufficient fungible processors ensure that an intermediate priority task executing independently can be assigned to a different processor than a low priority task that is blocking a high priority task. In addition, each intermediate priority task executing independently can be assigned to a different processor than a low-priority task that has had its priority raised needlessly. Assuming that a system contains more processors than tasks competing for processors, does the subsequent elimination of priority inversion guarantee that high priority tasks are given fast access to resources? Unfortunately, more and faster processors result in greater competition for resources, since more tasks are executing at any given time. If more processors are added [13] or if the speed of a system's current processor is increased [4], more lower-priority tasks obtain resources that are soon needed by high-priority tasks or by tasks upon which they are dependent, thus delaying the high priority tasks. This situation worsens with an increase in the ratio of low priority tasks to high priority tasks sharing the same resources. Nor should a system refuse to install faster and/or more processors; these are too valuable for throughput, reliability, and availability purposes, as well as for providing considerable assistance for the scheduling of hard real-time tasks. Faster processors enable critical sections to be executed more rapidly, assisting in unblocking hard real-time tasks. In addition, multiple processors provide a rare tool for the prevention of communication priority inversion, as well as preventing resource priority inversion when enough processors are available.

Multi-processing systems use heuristics in handling the complexity of real-time scheduling [12]. For example, the fungibility of processors is restricted for tasks that are part of a cooperating group. Gang scheduling policies in multi-processor systems have been developed for dependent tasks that are frequently involved in cooperation synchronization. These tasks are scheduled at the same time (perhaps in the same time slice), reducing synchronization and context switching overhead, as well as average response time. In preemptive real-time systems, gang members are assigned affinities to different processors so that they execute in parallel whenever possible. The scheduler also chooses a gang priority to assign to all members of the same cooperating group [15]. Tasks that share resources, on the other hand, are assigned affinities to the same processor to limit competition over resources, with systems using priority scheduling and priority inheritance to control scheduling order at the processor. Thus, at the potential loss of some concurrency, different types of restrictions on the fungibility of processors provide important mechanisms for the control of priority inversion with both cooperation and competition synchronization.

Deadlock control is generally not concerned with fungible processors that typically can be preempted. However, multiple processors have deleterious effects on deadlock occurrence. Some systems do not choose to prevent resource deadlock, estimating that its occurrence is rare and recovery is not too onerous; detection and recovery are preferred. However, as discussed previously, multiple processors enable more tasks to compete for resources, so that resource deadlock becomes more probable. Similar to the control of resource priority inversion with fungible resources, tasks competing for resources should be assigned affinities to the same processor, restricting the fungibility of processors, in order to ameliorate the control of resource deadlock in multiprocessing systems.

6. CONCLUSION

This paper has investigated priority inversion, a scheduling anomaly that can have critical consequences for hard real-time systems. Inconsistencies that result from common definitions of priority inversion have been discussed, with some modifications recommended.

Priority inversion with resources has many characteristics in common with resource deadlock, an anomaly that has been examined at length in the computer science community for at least forty years. Concepts gained from the study of resource deadlock aid in an understanding of "resource priority inversion." In particular, mechanisms for the control of resource deadlock with fungible resources have been shown to be applicable to the control of priority inversion. In addition, restricted processor fungibility assists in the control of all types of priority inversions as well as resource deadlock.

7. REFERENCES

[1] Chen, Y.S., Chang, L.P., Kuo, T.W., and Mok, A.K. 2009. An anomaly prevention approach to real-time task scheduling, *Journal of Systems and Software*, 82, 1 (Jan. 2009), 114-154.

[2] Cheng, A. and Ras, J. 2007. The implementation of the priority ceiling protocol in Ada-2005, *Ada Letters*, 27, 1 (April 2007), 24-39.

[3] Cornhill, D. and Sha, L. 1987. Priority inversion in Ada, *Ada Letters*, 7, 7 (Nov. Dec. 1987), 30-32.

[4] Graham, R. L. 1976. Bounds on the performance of scheduling algorithms, *Computer and Job-Shop Scheduling Theory*, E.G. Coffman, ed., John Wiley and Sons, NY, 165-227.

[5] Holt, R.C. 1972. Some deadlock properties of computer systems, *ACM Computing Surveys*, 4, 3 (Sept. 1972) 179-195.

[6] Huang, J., Stankovic, A., Ramamritham, K., and Towsley, D. 1991. On using priority inheritance in real-time databases, *12th Real-Time Systems Symposium* (Dec. 1991) 210-221.

[7] Kim, K. H. 2003. Basic program structure for avoiding priority inversion, *IEEE Proc. Of 6th International Symposium on Object-oriented Real- time Distributed Computing* (ISORC03), Hakodate, Japan (May 2003) 26-34. DOI=http://dream.eng.uci.edu/TMO/pdf/isorc2003.pdf

[8] Kleinrock, L. and Opderbeck, H. 1977. Throughput in the ARPANET- protocols and measurement, *IEEE Transactions on Communications***,** COM-25, 1 (Jan. 1977) 95-104.

[9] Levine, G. 1988. The control of priority inversion in Ada, *Ada Letter*s, 8, 6 (Nov. Dec. 1988) 53-56.

[10] Levine, G. N. 2003. Defining deadlock, *Operating Systems Review*, 37, 1 (Jan. 2003) pp. 54-64.

[11] Levine, G. N. (2009) Defining defects, errors, and service degradations, *ACM SIGSOFT Software Engineering Notes*, 24, 2 (March 2009) 1-14.

[12] Mueller, F. 1999. Priority inheritance and ceilings for distributed mutual exclusion, *20th IEEE Proceedings on Real-Time Systems Symposium*, (1999) 340 – 349.

[13] Naeser, G. 2005. Priority inversion in multi processor systems due to protected actions, *Ada Letters*, 25, 1 (March 2005) 43-47.

[14] Reeves, G. E. 1997. What really happened on Mars? (Dec. 1997). DOI= http://research.microsoft.com/enus/um/people/mbj/mars_pathfinder/authoritative_account.html

[15] Röck, H., Auerbach, J., Kirsch,*C.*, and Bacon, D. 2009. Avoiding unbounded priority inversion in barrier protocols using gang priority management. *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems* (2009) 70-79. DOI= http://www.cs.unisalzburg.at/~hroeck/papers/JTRES09-GPM.pdf

[16] Rosenkrantz, D.J., Stearns, R. E., and Lewis, P.M. 1978. System level concurrency control for distributed database systems, *ACM Trans. on Database Systems*, 3, 2 (June 1978) 178-198.

[17] Sha, L., Rajkumar, R., and Lehoczky, J. P. 1990. Priority inheritance protocols: an approach to real-time synchronization, *IEEE Transactions on Computers*, 39, 9 (Sept. 1990) 1175-1185.

[18] Welc, A., Hosking, A. L., and Jagannathan, S. 2006. Preemption-based avoidance of priority inversion for Java, *Int. Journal of Computer Science & Applications*, 8, 11 (2006) 40-50. DOI=www.adamwelc.org/papers/icpp04.pdf.