Real-Time Systems, 5, 285-303 (1993)
© 1993 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Formalization of Priority Inversion

ÖZALP BABAOĞLU*
Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, I-40127 Bologna, Italy

KEITH MARZULLO** AND FRED B. SCHNEIDER***
Department of Computer Science, Cornell University, Ithaca, New York 14853

Abstract. A priority inversion occurs when a low-priority task causes the execution of a higher-priority task to be delayed. The possibility of priority inversions complicates the analysis of systems that use priority-based schedulers because priority inversions invalidate the assumption that a task can be delayed by only higher-priority tasks. This paper formalizes priority inversion and gives sufficient conditions as well as some new protocols for preventing priority inversions.

1. Introduction

Task specifications in real-time systems usually involve bounds on completion times. Use of a *priority scheduler* can simplify implementing such specifications (Liu and Layland 1973), (Zhao, Ramamritham and Stankovic 1987). Each task is assigned a priority; whenever there is contention for a resource, access is granted to the task with the highest priority among those competing. Thus, the system designer controls the order in which resources are granted to tasks by assigning priorities. This, in turn, allows control over how resource contention affects task completion times.

A *priority inversion* occurs when a lower-priority task delays execution of a higher-priority task (Lampson and Redell 1980), (Sha, Rajkumar and Lehoczky 1990). For example, a task holding a lock for some data will cause any task attempting to acquire a conflicting lock to be delayed. If the task holding the lock has a lower priority than the task attempting to acquire the conflictng lock, then a lower-priority task is delaying a higher-priority one and a priority inversion has occurred.

The possibility of priority inversions complicates the analysis of a system that uses a priority scheduler. Not only must a task compete for resources with higher-priority tasks,

*Supported by the Commission of the European Communities under the ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems) and the Italian Ministry of Research and University, and in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG-2-593.
**Supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, and by grants from IBM T.J. Watson Research Laboratory, the IBM Endicott Programming Laboratory, Siemens RTL, and Xerox Webster Research Center.
***Supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-8701103, DARPA/NSF Grant No. CCR-9014363, and by the IBM Endicott Programming Laboratory.

but during a priority inversion, it competes with lower-priority tasks as well. If a high-priority task τ_H can be delayed by a lower-priority task τ_L, then τ_H effectively competes with all tasks assigned priorities at least that of τ_L, rather than with only those tasks assigned priorities at least that of τ_H. Since τ_L can be an arbitrary task, establishing that τ_H will meet a response-time goal involves reasoning about all tasks in the system rather than just the subset having priority at least that of τ_H.

The simpler analysis made possible by completely avoiding priority inversions is not without cost, however. It is not difficult to construct task sets in which priority inversions are avoided and CPU utilization never exceeds 50%, but were priority inversions permitted, then 100% CPU utilization would be possible. Thus, the system designer is faced with a common tradeoff: choosing between reduced system complexity and increased efficiency. This is not to say that protocols avoiding priority inversion lead to inefficiency. But there is strong empirical evidence that this will be the case since avoiding priority inversions necessarily rules out possible executions.

If the duration of priority inversions can be bounded, then delays by lower-priority tasks can be considered part of the time required to allocate a resource. In *priority inheritance* protocols (Lampson, and Redell 1980), (Sha, Rajkumar and Lehoczky 1990), the duration of priority inversions is bounded and task deadlines are met by dynamically modifying task priorities. A task's priority is elevated to a level that is the maximum of its original priority and the priority of any task that is being delayed by it. Priority inversions are permitted, but only in a carefully controlled way. The analysis of these protocols, however, is quite subtle, which supports our conjecture that allowing priority inversions complicates the analysis of a system. Of course, users of these protocols need not themselves perform such an analysis, and the use of priority inheritance protocols in actual systems is increasing.

In this article, we give a formal framework for investigating resource allocation strategies that avoid priority inversions completely. We then use this framework to help understand techniques to avoid—rather than control—priority inversions. In Sections 2 and 3, we formalize priority inversion and give sufficient conditions for its prevention; we then illustrate these ideas using some simple reservation-based protocols. To further illustrate the use of our framework, in Section 5 we develop a new concurrency control scheduler for avoiding priority inversions in database systems. Of note is how we model transactions which, unlike ordinary tasks, can be aborted. In Section 6, we consider conditions for avoiding priority inversions in systems where there are multiple independent schedulers, each making allocation decisions for some subset of the resources. Section 7 puts our work in context and discusses some unsolved problems.

2. System Model

Formalizing priority inversion requires that we formalize the notions of priority assignment and delay. To do this, we model a system as a set of tasks $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ where a *task* is any computation that can be scheduled. Thus, our use of the term task is synonymous with alternatives such as *process*, *job*, and *transaction*.

2.1. Task Priorities

A *priority assignment* is an irreflexive, partial order[1] on T where $\tau \prec \tau'$ means that task τ has lower priority than τ'. Observe that this definition allows tasks to have incomparable priorities. Therefore, it is possible that neither $\tau \prec \tau'$ nor $\tau' \prec \tau$ holds for some pair of tasks τ and τ'. By assigning incomparable priorities to tasks, the number of constraints imposed by a priority assignment is reduced, avoiding the introduction of extraneous priority inversions in our model of a system.

Define the *peer group* of a task τ to be the set of tasks τ' such that either $\tau \prec \tau'$ or τ' is incomparable to τ. In the absence of priority inversions, we need only consider τ and tasks that are in its peer group when analyzing whether τ will satisfy given response-time constraints. This is because only tasks in the peer group of τ can cause it to be delayed.

2.2. Resources

Tasks can cause each other to be delayed in a variety of ways. Some of these causes are explicit, such as when one task awaits a message sent by another or when a lock held by one task prevents another from acquiring that lock. Other causes of delay are implicit. For example, the presence of finite-capacity, time-multiplexed resources, such as memory, processors, and I/O devices, can lead to the implicit delay of a task requiring use of a resource by the task using that resource. Notice that we do not distinguish between consumable and serially-reusable resources (Bic and Shaw 1988) as has been traditional when considering deadlock. In formalizing priority inversion, our concern is with the existence of a delay and not the details of the system activity to terminate that delay.

We postulate that a system comprises a set of resources and a scheduler.[2] A task obtains access to a single unit of a resource `r` by invoking the `request(r)` operation. If the resource is not available then the task invoking the request is delayed. A delayed task may later be granted the resource by the scheduler when another task invokes the `release(r)` operation. Observe that a task may have only one outstanding request at a time. Thus, in our model, if multiple units of a resource or several different resources are needed for execution, a task must request and acquire them sequentially. We return to the possibility of a task requesting multiple resources through a single operation in Section 7.3.

Request and release operations are not always explicitly invoked by tasks. Sometimes, these operations are invoked implicitly as part of some other system operation. For example, an operation to receive a message will implicitly invoke a request operation that is granted when the message becomes available for receipt. In other cases, request or release might not be invoked by tasks at all, instead being invoked due to other activity in the system. Consider a multiprogrammed processor that uses an interval timer to force task switches. An execution of the interval-timer interrupt handler can be regarded as (i) performing a release and a subsequent request on behalf of the task that was executing when the timer interrupt occurred and then (ii) granting the pending request of the task that is next selected for execution on the processor.

Our request/release model turns out to be quite general. It can even be used to describe situations in which tasks are delayed because of some application-dependent aspect of the system state. For example, it is not unusual for a concurrent program to contain some form of conditional wait statement that delays a task until the program variables satisfy some Boolean expression. (Most synchronization primitives are instances of such conditional wait statements.) We can model a conditional wait by regarding it as a request on a resource. The request is granted if the Boolean expression is true; otherwise, the request is delayed. Execution of any assignment statement in another task that makes the Boolean expression true is considered a release on the resource.

2.3. *Delay*

We model task delays using a binary relation. A task τ_i *waits-for* task τ_j at time t, denoted by $\tau_i \underset{t}{\rightarrow} \tau_j$, if and only if τ_i is delayed at time t in its request for some resource and τ_j can release that resource. Note that more than one task might be able to release the resource being requested. We can model this using a conjunction of waits-for relations. For example, if there are multiple units of the resource available, then the delay is attributed to the set of all tasks $\mathcal{T}$ that have been granted but not yet released the resource:

$$\bigwedge_{\tau \in \mathcal{T}} (\tau_i \underset{t}{\rightarrow} \tau).$$

3. Characterizing Priority Inversion

In order to characterize priority inversion, we must reason about the transitive closure $\rightarrow_t^+$ of the waits-for relation.[3] Priority inversion has occurred at time t when progress of a task is blocked by the actions of a lower priority task. Thus, a system contains a priority inversion at time t if and only if there exist tasks τ_i and τ_j such that $(\tau_i \rightarrow_t^+ \tau_j) \wedge (\tau_j \prec \tau_i)$.

It will be convenient to represent the priority assignment and waits-for relations in effect at a given time t with a directed graph. The directed graph corresponding to a priority assignment $\prec$ on a set of tasks T is $P = (T, E_P)$ where $E_P = \{(u, v) \mid u \prec v\}$. Thus, the nodes of P represent tasks and an edge is drawn from a task to all higher priority tasks. Similarly, the waits-for relation at time t can be represented as the directed graph $W = (T, E_W)$ where $E_W = \{(u, v) \mid u \underset{t}{\rightarrow} v\}$. Here, edges are drawn from a task to all other tasks that it waits for.

Given these graphs, the system state at a time t can be represented by a *composite system graph*, $G = (T, E_P \cup E_W)$. Figure 1 depicts such a graph for a system of four tasks. Single-arrow edges represent waits-for relations and double-arrow edges represent priority relations. Thus, the graph depicts the situation where there are four tasks such that $\tau_3 \prec \tau_1 \prec \tau_2$ and $(\tau_1 \underset{t}{\rightarrow} \tau_2)$, $(\tau_2 \underset{t}{\rightarrow} \tau_4)$, $(\tau_4 \underset{t}{\rightarrow} \tau_3)$. Note that there can be multiple edges between a pair of nodes.

Composite system graphs can be used to characterize priority inversions. In a composite system graph, a cycle containing exactly one priority edge is equivalent to the existence of a priority inversion.[4] We call such cycles *π-cycles* to distinguish them from cycles containing

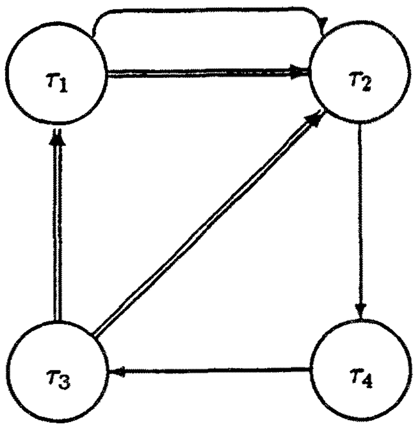

Figure 1. Four-task composite system graph.

no priority edges. Thus, the system depicted in Figure 1 contains two priority inversions, corresponding to the two *π-cycles* ($\tau_3 \prec \tau_1 \underset{t}{\rightarrow} \tau_2 \underset{t}{\rightarrow} \tau_4 \underset{t}{\rightarrow} \tau_3$ and $\tau_3 \prec \tau_2 \underset{t}{\rightarrow} \tau_4 \underset{t}{\rightarrow} \tau_3$).

Any condition that prevents a composite system graph from having a π-cycle is a sufficient condition for avoiding priority inversions. Examples of such conditions are the following:

1. *No Priority Assignment.* If all tasks were incomparable, then the priority graph P would be empty and a π-cycle could never form.
2. *No Delay.* Without delay, the waits-for graph W would be empty, guaranteeing the absence of the π-cycles.
3. *Preemption.* A waits-for relation persists until a task relinquishes a resource. Preemption causes a task to relinquish a resource, so preemption can change the waits-for relation in a way that prevents a π-cycle from forming.
4. *No π-Cycle.* The waits-for and the priority relations must form in a particular fashion for priority inversion to exist. The relevant property is a π-cycle.

Note that preemption both removes and adds elements to the waits-for relation. By preempting a resource r that had been granted to a task τ_i and granting r to τ_j, all waits-for relations from τ_j to other tasks are removed and waits-for relations from τ_i to all tasks that have access to the resource are added.

Inverses of conditions 2, 3, and 4 correspond to the three necessary and sufficient conditions for deadlock of (Coffman and Denning 1973). When negated, our conditions 2 and 3 correspond exactly to the *Mutual Exclusion* and *Nonpreemption* conditions of (Coffman and Denning 1973); negation of our condition 4 is a refinement of the *Resource Waiting* condition[5] of (Coffman and Denning 1973).

Any strategy for preventing priority inversions ultimately must be based on avoiding π-cycles. In the strategies that follow, we do just this by ensuring that at least one of the sufficient conditions above holds. First, in Section 4, we show how by eliminating the possibility of certain waits-for relations, a π-cycle is avoided. Thus, this priority inversion strategy is based on condition 4, No π-Cycle. Then, in Section 5, we show an application of preemption by giving a timestamp-based concurrency controller. This strategy is based on condition 3, Preemption.

4. Understanding Reservation Protocols

In nonpreemptive schedulers, π-cycles can be prevented by having each task reserve in advance the interval during which it will hold a resource and using that information to prevent certain waits-for relations from forming. If reservations from a low-priority task never overlap with reservations from a higher-priority task, then priority inversons do not occur. In this section, we describe two protocols that exploit this insight for avoiding priority inversions. The protocols themselves are not complicated, although getting all of their details right was helped by our formal characterization for priority inversions.

Let $H_i = \{\tau \mid \tau_i \prec \tau\}$ be the set of tasks that have higher priority than τ_i, and let $I_i = \{\tau \mid \neg(\tau_i \prec \tau) \wedge \neg(\tau \prec \tau_i)\}$ be the set of tasks incomparable to τ_i. Thus, the peer group for a task τ_i is $PG_i = H_i \cup I_i$. For each resource r, assume that each task τ_i is able to compute $hold_i^r(t)$, the upper bound on the amount of time that τ_i will hold r the next time (with respect to t) it is granted r, and $next_i^r(t)$, the lower bound on the next time (with respect to t) τ_i will request r. During the interval between when task τ_i requests resource r and when it releases r, define $next_i^r(t)$ to equal t. And, if τ_i holds r at time t then define $hold_i^r(t)$ to be an upper bound on the remaining amount of time τ_i will hold r.

The reservation protocols we derive base their scheduling decisions on the values of $hold_i^r(t)$ and $next_i^r(t)$. For this to be feasible, their computation must be independent of any particular schedule. This independence is achieved by defining $hold_i^r(t)$ and $next_i^r(t)$ as upper and lower bounds, respectively, over all schedules. In other words, the computation of $hold_i^r(t)$ must be based on a worst-case scenario where task τ_i is delayed by all other tasks in its peer group whereas that of $next_i^r(t)$ must be based on a best-case scenario where task τ_i executes with no interleavings.

Our assumptions about knowledge of $hold_i^r(t)$ and $next_i^r(t)$ are satisfied in systems where tasks are periodic and can be analyzed before execution commences—systems in which priority inversions could also have been avoided by using *a priori* static scheduling of tasks. However, it is not hard to imagine systems for which static scheduling is not possible, but $next_i^r(t)$ and $hold_i^r(t)$ are still computable.

Two simple allocation policies based on knowledge of $next_i^r(t)$ and $hold_i^r(t)$ are:

Policy 1. A request by task τ_i for resource r at time t is granted if

$$hold_i^r(t) \leq \min_{\tau_j \in PG_i} (next_j^r(t) - t).$$

Otherwise, the request is delayed.

Policy 2. Let R be the set of all shared resources. A request by task τ_i for resource r is granted if

$$hold_i^r(t) \leq \min_{\tau_j \in H_i} (\min_{r' \in R}(next_j^{r'}(t) - t)).$$

Otherwise, the request is delayed.

While it is probably clear that these policies prevent priority inversions, it may not be clear whether they are overly restrictive. For example, must incomparable tasks be included in the test of Policy 1? By giving a formal derivation of Policy 1—as we now do—the need for this part of the test can be understood.

In general, an allocation policy can be derived from an invariant[6] that precludes formation of π-cycles. For a nonpreemptive scheduler, the invariant must imply that there is no π-cycle present in the current state and that the current state is not one from which formation of a π-cycle is inevitable. Therefore, it suffices that the invariant imply the stronger condition that there is no π-cycle present in the current state and that the current state is not one from which formation of a π-cycle is *possible*. Although such a stronger invariant could rule out safe states, it is usually easier to construct and maintain.

Using this approach, we now derive Policy 1. In a system where resources are shared infrequently, the most common π-cycle would involve just two tasks τ_i and τ_j (say) such that $\tau_j \underset{t}{\rightarrow} \tau_i \wedge \tau_i \prec \tau_j$. An obvious invariant to choose is the negation of this predicate:[7]

$$(\forall \tau_i, \tau_j \colon \tau_j \underset{t}{\rightarrow} \tau_i \supset \neg(\tau_i \prec \tau_j)) \tag{1}$$

However, (1) does not imply that there can be no π-cycle in a composite system graph since a π-cycle might involve a chain of $\underset{t}{\rightarrow}$ edges. The following predicate does:

$$(\forall \tau_i, \tau_j \colon \tau_j \rightarrow_t^+ \tau_i \supset \neg(\tau_i \prec \tau_j)) \tag{2}$$

So, we strengthen (1). First, note that because $\prec$ is asymmetric, we have:

$$(\forall \tau_i, \tau_j \colon \tau_j \prec \tau_i \supset \neg(\tau_i \prec \tau_j)) \tag{3}$$

Thus,

$$(\forall \tau_i, \tau_j \colon \tau_j \underset{t}{\rightarrow} \tau_i \supset \neg(\tau_j \prec \tau_i)) \tag{4}$$

implies (1), and so if (4) also implied (2) then (4) would imply there can be no π-cycle in the composite system graph.

We now show that (4) implies (2). Assume (4) holds. Since $\prec$ is a transitive relation, we have from (4) and the definition of $\rightarrow_t^+$ that:

$$(\forall \tau_i, \tau_j \colon \tau_j \rightarrow_t^+ \tau_i \supset \tau_j \prec \tau_i)$$

This last equation together with (3) implies (2). Thus, (4) implies that there can be no π-cycle in the composite system graph and can be used as the basis for the invariant.

Unfortunately, (4) does not imply that the current state is one from which later formation of π-cycles is impossible—only that no π-cycle currently exists. For example, suppose a task τ_i can allocate a resource r that a task τ_j will later request and $\tau_i \prec \tau_j$ holds. If τ_i allocates r, then (4) is not invalidated. However, if τ_j subsequently attempts to allocate r, then (4) is invalidated because the request cannot be granted (without preempting r from τ_i). So, (4) is not strong enough to prevent subsequent formation of a π-cycle.

We can strengthen (4) by weakening its antecedent. The antecedent asserts that τ_j is waiting for τ_i; a weaker requirement is that τ_j may wait for τ_i at some future point. This weaker antecedent can be formalized in terms of *hold* and *next* as

$$(\exists t'\colon t \le t'\colon (\exists r\colon r \in R_i(t)\colon (hold_i^r(t) + t > t') \wedge (next_j^r(t) \le t')))$$

where $R_i(t)$ is the set of resources that τ_i has allocated at time t. After some manipulation using predicate logic, the strengthened (4) is:

$$\mathcal{I}\colon (\forall \tau_i, \tau_j\colon \neg(\tau_j \prec \tau_i) \supset (\forall r\colon r \in R_i(t)\colon hold_i^r(t) + t \le next_j^r(t)))$$

To ensure that $\mathcal{I}$ holds throughout execution, we must show that it is true initially and guarantee that execution does not invalidate it. Assuming that tasks are initially started with no resources allocated to them, $R_i(0)$ will be empty for all tasks τ_i, and so $\mathcal{I}$ is initially true. To guarantee that $\mathcal{I}$ is not invalidated by execution, assume that it is true and some task τ_i requests a resource r'. Let $ok(r', \mathcal{I})$ denote those systems states in which r' could be added to $R_i(t)$ without invalidating $\mathcal{I}$. Then, any policy that ensures a condition $C(r', \tau_i)$ holds before r' is granted to τ_i, such that

$$C(r', \tau_i) \wedge \mathcal{I} \supset ok(r', \mathcal{I})$$

is valid, will ensure that $\mathcal{I}$ holds throughout execution (hence the formation of π-cycles is precluded).

The allocation of a resource r' to a task τ_i can be modeled by an assignment statement $R_i(t) := R_i(t) \cup \{r'\}$. The set of states that must hold before execution of an assignment $x := E$ in order to ensure that some predicate Q holds afterwards is given by $wp(x := E, Q)$, Dijkstra's weakest precondition predicate transformer (Dijkstra 1976).[8] Thus, we compute

$$\begin{aligned} ok(r', \mathcal{I}) &= wp(R_i(t) := R_i(t) \cup \{r'\}, \mathcal{I}) \\ &= \mathcal{I}^{R_i(t)}_{R_i(t) \cup \{r'\}} \\ &= (\forall \tau_i, \tau_j\colon \neg(\tau_j \prec \tau_i) \supset \\ &\qquad (\forall r\colon r \in R_i(t) \cup \{r'\}\colon hold_i^r(t) + t \le next_j^r(t))) \end{aligned}$$

Thus, for

$$C(r', \tau_i) \overset{\text{def}}{=} (\forall \tau_j\colon \neg(\tau_j \prec \tau_i) \supset hold_i^{r'}(t) + t \le next_j^{r'}(t))$$

we have

$$C(r', \tau_i) \wedge \mathcal{I} \supset ok(r', \mathcal{I}).$$

Therefore, provided $C(r', \tau_i)$ holds before r' is granted to τ_i we can conclude that $\mathcal{I}$ will not be invalidated by program execution. This can be done by ensuring that the scheduler never grants r' to τ_i unless $C(r', \tau_i)$ holds, which is exactly Policy 1.

Policy 1 is conservative. It does not allow waits-for edges to develop between incomparable tasks in fear of a cycle developoing indirectly between two comparable tasks. Policy 2 is conservative in a different way: it avoids instances of Figure 2. And, it can be derived in a manner similar to Policy 1. In a system with a total order priority assignment, Policy 1 is superior, while in a system where there are many incomparable tasks and few shared resources, Policy 2 is appropriate. Depending on the application, there may be other properties of the possible composite systems graphs that can be exploited in order to derive a better resource allocation policy.

5. A New Concurrency Control Protocol

We now consider how π-cycles can be prevented in a database system by a timestamp-based concurrency controller. Although the utility of employing transactions in a real-time system is debatable, deriving a transaction concurrency controller for avoiding priority inversions does provide another illustration of our theory. Moreover, our concurrency controller appears to be the first to use timestamps for avoiding priority inversions. All other database concurrency controllers we know of that avoid priority inversions use locking (Abbott and Garcia-Molina 1989), (Sha, Rajkumar and Lehoczky 1990).

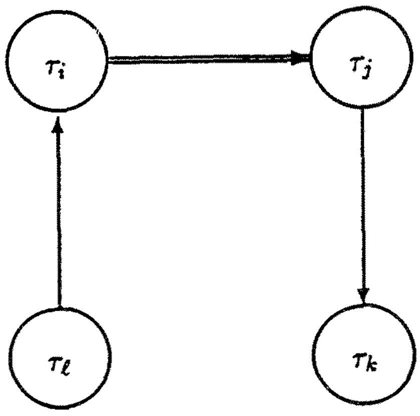

Figure 2. Composite system graph configuration avoided by Policy 2.

5.1. Serializability and Priority Inversion

A task accesses a database by encapsulating its read and write operations on the database within a *transaction*. We can model a transaction as a sequence of read and write operations on items of the database followed by either a commit operation or an abort operation. Given a set of transactions, a *history* is a total order of the operations of the transactions.

Two operations *conflict* if their execution order cannot be interchanged without altering the effects of one or the other. Two histories h_1 and h_2 are *equivalent* if they are over the same set of transactions and, for all pairs of conflicting operations α and β in the histories, if α precedes β in h_1 then α precedes β in h_2.[9] If α and β are conflicting and are from different transactions, we will say that the two transactions conflict.

A *concurrency controller* is a scheduler that takes as input operations from transactions and produces a history that is *serializable*—that is, each transaction either commits or aborts, and execution of the committed transactions is equivalent to executing them in some serial order. A transaction τ_i *precedes* a transaction τ_j in a history h, written $\tau_i \lessdot \tau_j$, if neither transaction aborts in h and in all serial executions equivalent to h, τ_i executes before τ_j. Serializability of a set of transactions T can be characterized in terms of a *serialization graph*, $G = (T, E_C)$ where $E_C = \{(u, v) \mid u \lessdot v\}$. By definition, $\lessdot$ is an irreflexive partial order, so G is acyclic.

By allowing only serializable executions, a concurrency controller ensures that the serialization graph G never contains cycles (Bernstein, Hadzilacos and Goodman 1987). For example, suppose some operation α_i of τ_i was executed before a conflicting operation β_j of τ_j. In this case, a concurrency controller effectively adds an edge from τ_i to τ_j in G. If two more conflicting operations, γ_i of τ_i and δ_j of τ_j, are to be executed, then γ_i must execute before δ_j, since to do otherwise would add an edge in G from τ_j to τ_i, creating a cycle.[10]

A concurrency controller can take one of three actions for each operation that a transaction submits. It can execute the operation; it can reject the operation and abort the transaction because if the operation were to be executed then a cycle could be formed; or else, it can postpone performing the operation until it can guarantee that no cycle will be formed were that operation executed.

Using the formalism of Section 2.2, we model transactions as tasks and a concurrency controller as a scheduler. Executing an operation from a transaction is equivalent to granting a reosurce to the corresponding task. Rejecting or postponing an operation from a transaction is equivalent to delaying the correspoinding task. We next discuss these two methods of delaying a task.

5.1.1. Postponing operations. A concurrency controller postpones an operation α_i of a transaction τ_i if executing α_i would introduce a cycle in the serialization graph. If some transaction τ_j would be part of that cycle then τ_i waits for τ_j and we have that $\tau_i \underset{t}{\rightarrow} \tau_j$ holds. Henceforth, we assume the following property about concurrency controllers:

POST: If $\tau_i \underset{t}{\rightarrow} \tau_j$ because an operation of τ_i is postponed, then τ_i and τ_j are both active at time t and $\tau_j \lessdot \tau_i$.

This is a reasonable assumption because were it not true, then there would exist an equivalent serial execution in which τ_i precedes τ_j yet the concurrency controller postpones operation α_i of τ_i until some operation β_j of τ_j completes. Such a delay would be capricious, since all the operations of τ_i could execute before any operation of τ_j and yield the same results as when α_i is delayed. It is, therefore, not surprising that all the concurrency control algorithms that we know of satisfy this assumption.

By postponing operations, a concurrency controller can cause priority inversions. Define a *priority-ordered* concurrency controller to be one that ensures

POCC$_1$: For all transactions τ_i and τ_j, if τ_j starts before τ_i completes and $\tau_i \prec \tau_j$, then $\tau_j \preccurlyeq \tau_i$.

POCC$_1$ ensures that higher-priority transactions are ordered before concurrently executed lower-priority ones.

We now show that a priority-ordered concurrency controller cannot introduce priority inversions. Assume that the composite systems graph contains the π-cycle $\tau_i \prec \tau_j \xrightarrow[t]{} \ldots \xrightarrow[t]{} \tau_i$. Thus, τ_i and τ_j are both active at time t and so τ_j started before τ_i completed. By POST, a delay edge in a composite system graph that can be attributed to the postponement of an operation has a corresponding conflict edge in the serialization graph, and so there is a path of conflict edges from τ_i to τ_j. By POCC$_1$, there is a conflict edge from τ_j to τ_i. Thus, if there is a π-cycle in the composite system graph then there is a cycle in the serialization graph. Since a concurrency controller ensures the absence of cycles in the serialization graph, there can be no π-cycle in the composite system graph.

5.1.2. Rejecting Operations. In terms of our model, rejecting an operation submitted by a task τ_a (and thereby aborting τ_a) can be regarded as having τ_a wait for some (virtual) resource held by other tasks. This is because a transaction that is aborted does no useful work prior to its restart, and in our model, a task whose transaction can do no useful work is considered blocked.

If a transaction τ_a is aborted in order to avoid a cycle in the waits-for graph, then a priority inversion can occur depending on the priorities of transactions holding the virtual resource being requested by (and blocking) τ_a. Define the set B_a of transactions holding this virtual resource to be the smallest set such that had none of these executed at all, then τ_a would not have been aborted. Thus, if each transaction in B_a is in the peer group of τ_a, then aborting τ_a to avoid a cycle in the serialization graph does not create a priority inversion. Let C be the set of transactions involved in cycles in the serialization graph, and let A be the subset of C that are aborted in order to remove those cycles. Then, $B_a \subset C - A$ holds, and we have:

POCC$_2$: A concurrency controller that aborts a transaction τ_a will avoid priority inversions provided B_a is a subset of the peer group of τ_a.

5.2. Timestamp-based Concurrency Controllers

Timestamp-based concurrency controllers work by assigning a unique *timestamp* $ts(\tau_i)$ to each transaction τ_i. These timestamps are used to totally order transactions. The ordering is such that if $ts(\tau_i) < ts(\tau_j)$, then there is an edge in the serialization graph from τ_i to τ_j.

Timestamp-based concurrency controllers both postpone operations and reject operations. In order to ensure that such a concurrency controller does not introduce priority inversions, two conditions suffice. The first condition is that the concurrency controller be priority-ordered, since being priority-ordered (i.e., satisfies POCC_1) implies that postponing operations will not introduce priority inversions. The second is that rejecting operations does not introduce priority inversions (i.e., satisfies POCC_2). We now consider each of these conditions in detail.

A timestamp-based concurrency controller can be made priority-ordered by suitable assignment of timestamps. This is because POCC_1 requires that certain edges exist in a serialization graph. Since the concurrency controller adds such edges by assigning timestamps, it suffices to ensure that

POTS: If τ_j starts while τ_i is active and $\tau_i \prec \tau_j$, then $ts(\tau_j) < ts(\tau_i)$.

It is not hard to assign such timestamps. For simplicity, assume integer priorities; extension to general priorities is straightforward. Also assume there exist P priority levels $1, 2, \ldots, P$, where level $\ell_i \prec \ell_j$ if and only if $\ell_i < \ell_j$, and assume that timestamps are real numbers. Let

max^c	be the largest timestamp of all committed transactions,
min_i^a	be the smallest timestamp of all active transactions of priority i, and
max_i^a	be the largest timestamp of all active transactions of priority i.

If no transactions of priority i are active, then the value of min_i^a and max_i^a is defined to be $\perp$, where $\min(x, \perp) = \max(x, \perp) = x$. Initially, $max^c = 0.0$ and for all priority levels ℓ, $min_\ell^a = max_\ell^a = \perp$. A timestamp s for a new transaction with priority p can be computed by finding upper and lower bounds for its value and selecting a unique value in that interval.[11] To be able to commit, the value of s must be larger than max^c, to satisfy POTS, it must be larger than the timestamps assigned to all transactions with higher-priority and smaller than the timestamps assigned to all transactions with lower priority. This is implemented by the code in Figure 3.

In order to illustrate this allocator, assume that $P = 2$, no transactions are active, and $max^c = 10$. If a priority 2 transaction starts, then it will be assigned a timestamp of 11 and $min_p^a = max_p^a = 11$. If a priority 1 transaction then starts before the first terminates, then *low* will be set to 10 and *high* will be set to 11 and so the transaction will be assigned a timestamp of 10.5.

```
$low$ := **max**[$max^c$, $max_\ell^a$: $1 \le \ell \le p$];
$high$ := **min**[$min_\ell^a$: $p < \ell \le P$];
**if** ($high = \perp$) **then** $s$ := $low + 1$
                    **else** $s$ := $(low + high)/2$;
$min_p^a$ := **min**[$min_\ell^a$, $s$];
$max_p^a$ := **max**[$max_\ell^a$, $s$];
```

Figure 3. Timestamp Allocator.

Having ensured that the concurrency controller is priority-ordered, it only remains to ensure that POCC$_2$ holds and so aborts do not introduce priority inversions. Suppose operation α_i from transaction τ_i is submitted before a conflicting operation β_j from τ_j. If $ts(\tau_i) > ts(\tau_j)$, then executing these operations in the order they are submitted would create a cycle in the serialization graph.[12] Thus, the concurrency controller must abort either τ_i or τ_j to avoid this cycle. From POTS, if $\tau_i \prec \tau_j$ then $ts(\tau_j) < ts(\tau_i)$ holds, so, according to POCC$_2$, no priority inversion can result by aborting $ts(\tau_i)$. The following rule, therefore, describes a rule for aborting transactions without introducing priority inversions.

Priority Abort Rule: If α_i from transaction τ_i is submitted before conflicting operation β_j from τ_j and $ts(\tau_i) > ts(\tau_j)$, then τ_i is aborted to avoid a cycle in the serialization graph.

This rule is the opposite of what is traditionally used in timestamp-based concurrency controllers (Bernstein, Hadzilacos and Goodman 1987). Traditionally, the transaction that submitted the last operation (e.g., τ_j above) would be aborted because this eliminates all cycles introduced by that operation. For example, suppose that $ts(\tau_i) = i$ and the following sequence of operations are submitted, where $r_i[x]$ denotes an operation by transaction τ_i to read x and $w_i[x]$ denotes an operation by transaction τ_i to write x:

$$r_2[x]\ r_3[x]\ w_1[x].$$

Performing $w_1[x]$ would introduce two cycles in the serialization graph—one involving τ_1 and τ_2, the other involving τ_1 and τ_3. The traditional abort rule would abort τ_1, but would create a priority inversion if $\tau_2 \prec \tau_1$ or $\tau_3 \prec \tau_1$. The Priority Abort Rule would abort both τ_2 and τ_3 and cannot cause a priority inversion because of the way timestamps are assigned.

Implementing the Priority Abort Rule is not completely straightforward. See (Marzullo 1989) for a detailed explanation of such an implementation.

6. Systems with Multiple Schedulers

It is not uncommon for system resources to be partitioned into disjoint subsets, where each subset is controlled by an independent scheduler. For example, the processors of a system are scheduled by a CPU scheduler, disk drives by some device driver module, and database accesses by the concurrency control module of a database manager. A database transaction would interact with all three separate schedulers during its execution.

Although using multiple, independent schedulers simplifies construction of a system, it complicates the avoidance of priority inversions. This is because a scheduler's decision to delay granting some resource to a task must be made using only partial information about the system state, and a delay caused by one scheduler might cause a priority inversion with tasks being delayed by other schedulers. To see this, consider a system with schedulers S_1 and S_2 and tasks τ_i, τ_j, and τ_k, where $\tau_k \prec \tau_i$. Further, suppose that an allocation decision by S_1 leads to $\tau_i \underset{t}{\rightarrow} \tau_j$ and an allocation decision by S_2 leads to $\tau_j \underset{t}{\rightarrow} \tau_k$. This is a priority inversion since $(\tau_i \rightarrow_t^+ \tau_k) \wedge (\tau_k \prec \tau_i)$. Notice that neither S_1 nor S_2 maintains sufficient local information to detect or prevent this priority inversion.

We can enjoy the benefits of having separate schedulers if freedom from priority inversions can be ensured by analyzing each scheduler in isolation. An obvious local criterion for correctness of a scheduler S is that S prevent priority inversions among tasks that have requested but not released resources from S. The two-scheduler example of the previous paragraph illustrates that this criterion by itself is not sufficient to ensure freedom from priority inversion—both S_1 and S_2 avoided such local priority inversions. We, therefore, now investigate useful conditions to ensure that avoiding local priority inversions is sufficient for avoiding all priority inversion.

Consider a system in which there is a set of independent schedulers and a single priority assignment that is known to all.[13] Define $\tau_i \; {}_S\!\underset{t}{\rightarrow} \; \tau_j$ to hold if and only if $\tau_i \underset{t}{\rightarrow} \tau_j$ and scheduler S is delaying τ_i's request for some resource. The *local state* of a scheduler S can be characterized by a directed graph $G_S = (T, E_S)$ where $E_S = \{(u, v) \mid u \prec v \vee u \; {}_S\!\rightarrow v\}$. Thus, G_S includes all of the priority edges of the composite system graph but only a subset of the waits-for edges. A *local priority inversion* exists for scheduler S if and only if G_S contains a π-cycle.

Since the composite system graph for a system with multiple schedulers is given by $G = (T, \bigcup_S E_S)$, a system contains a priority inversion at time t if and only if $\bigcup_S G_S$ contains a π-cycle. However, as illustrated in the two-scheduler example above, existence of a π-cycle in $\bigcup_S G_S$ does not imply a π-cycle in G_S for some scheduler S. The following theorem, reminiscent of one for deadlock avoidance presented in (Havender 1968), shows that this discrepancy is actually linked to our decision to specify priority assignments using partial orders.

THEOREM 1. A system with multiple schedulers is free from priority inversion if the priority assignment is a total order and each scheduler avoids local priority inversions.

Proof. By contradiction. Assume that the system has a priority inversion characterized by the π-cycle

$$\tau_i \prec \tau_j \underset{t}{\rightarrow} \ldots \underset{t}{\rightarrow} \tau_k \underset{t}{\rightarrow} \tau_\ell \underset{t}{\rightarrow} \ldots \underset{t}{\rightarrow} \tau_i$$

in the composite system graph. For each $\tau_k \underset{t}{\rightarrow} \tau_\ell$ in the π-cycle, we conclude $\tau_k \prec \tau_\ell$ because every pair of tasks is related by the priority assignment and no scheduler allows a local priority inversion. By the transitivity of $\prec$, we have $\tau_j \prec \tau_i$. Thus, from $\tau_i \prec \tau_j$ we obtain a contradiction. □

In many systems, the priority assignment is encoded by associating an integer priority $\Pi(\tau)$ with each task τ. If each task is assigned a unique priority, then the result is a total order and Theorem 1 implies that avoiding local priority inversions is sufficient for avoiding all priority inversions. However, constructing a total order from a partial order can require introduction of fictitious priority relations—avoiding priority inversions that involve these fictitious relations is unnecessary. Thus, we now consider the case where a unique priority is not assigned to each task, but Π does satisfy the following less restrictive conditions, which define a partial order (as opposed to an irreflexive partial order).

P1. $\Pi(\tau) < \Pi(\tau')$ if and only if $\tau \prec \tau'$

P2. $\Pi(\tau) = \Pi(\tau)$ implies $\neg(\tau \prec \tau') \wedge \neg(\tau' \prec \tau)$.

Observe that for a given priority assignment, a mapping that satisfies P1 and P2 might require adding some fictitious priority relations, but would require adding fewer priority relations than if Π defined a total order. The following theorem asserts that even though Π defines a partial order, avoiding local priority-inversions suffices to avoid all priority inversions.

THEOREM 2. If a task τ is only allowed to wait for a task τ' for which $\Pi(\tau) \leq \Pi(\tau')$ then the system is guaranteed to be free from all priority inversions.

Proof. By contradiction. Assume a priority inversion characterized by the π-cycle

$$\tau_i \prec \tau_j \underset{t}{\rightarrow} \dots \underset{t}{\rightarrow} \tau_k \underset{t}{\rightarrow} \tau_\ell \underset{t}{\rightarrow} \dots \underset{t}{\rightarrow} \tau_i$$

in the composite system graph. From the allocation policy, $\tau_k \underset{t}{\rightarrow} \tau_\ell$ implies $\Pi(\tau_\ell) \geq \Pi(\tau_k)$ for any two tasks τ_k and τ_ℓ in the π-cycle. By transitivity, we have $\Pi(\tau_i) \geq \Pi(\tau_j)$. By the hypothesis, $\tau_i \prec \tau_j$ and thus by P1 we have $\Pi(\tau_i) < \Pi(\tau_j)$, a contradiction. □

7. Discussion

A characterization of priority inversion is useful only to the extent that the formal model on which it is based correctly captures the relevant aspects of reality. We, therefore, now discuss the suitability of our model and the relaxation of certain of its restrictions.

7.1. Priority Assignments as Partial Orders

We have elected to formalize priority assugnments using irreflexive partial orders rather than mappings from tasks to integers (as is implemented in many operating systems). This selection was made because irreflexive partial orders are more expressive. As an example, consider a system with three tasks τ_1, τ_2 and τ_3 that implement *query*, *debit* and *credit* operations, respectively, against a given account in a banking database. To prevent false overdrafts, the bank has decided that credit operations should have priority over concurrent debit operations. Requests to query the account, on the other hand, should be performed without any preference relative to concurrent debits or credits. While it is trivial to express this priority relationship as a partial order, it is not possible to express it using an integer mapping Ψ since it would have to satisfy $\Psi(\tau_1) = \Psi(\tau_2)$, $\Psi(\tau_1) = \Psi(\tau_3)$ and $\Psi(\tau_3) > \Psi(\tau_2)$. Also, using an irreflexive partial order avoids fictitious priority relations, which, in turn, avoid fictitious priority inversions.

When irreflexive partial orders are used to specify priority assignments, there are two possible interpretations for the case where two tasks have incomparable priorities. One is that these tasks do not compete for resources; the other is that these tasks compete for resources but on an equal footing. In either case, if two tasks are incomparable then, by definition (see Section 2.1) each will be in the peer group of the other. At first, including in the peer group for τ tasks that do not compete for resources with τ might seem troubling. However, if two tasks do not compete for resources, then it doesn't matter whether one is in the peer group of the other—any analysis based on that peer group will be no more complicated by the presence of the noncompeting task.

Using irreflexive partial orders to specify priority assignments does have drawbacks. As shown in Section 6, using individual schedulers that ensure freedom from local priority inversion does not by itself guarantee that a system will be free of priority inversions. However, Theorem 2 shows that if the priority assignment is restricted to one that could be represented by a mapping from tasks to integers, guaranteeing freedom from local priority inversions does guarantee freedom from all priority inversions. Thus, in systems with multiple, independent schedulers, there are advantages to employing the less expressive formulation of priority assignment.

7.2. *Static and Global Prioriy Assignment*

One limitation of our model is the assumption of a single, static priority assignment. This rules out systems where a task's priority is a function of the system state. It also rules out systems with multiple independent schedulers that each assign different priorities to tasks.

A time-varying or dynamic priority structure can be modeled as a sequence of priority assignments,

$$\prec_1, \prec_2, \ldots, \prec_t, \ldots$$

where $\prec_t$ is the priority assignment in effect at time t. The formal characterization of priority inversion in Section 3 remains valid with this extension, but the protocols of Sections 4 and 5 require modifications. This is because if the priority assignment is not static, priority inversions can be caused simply by the passage of time changing the priority assignment; with a static priority assignment, a priority inversion can only occur from an (ungranted) request operation. Avoiding priority inversions for a dynamic priority structure, therefore, requires that changes to the priority assignment be coupled with the waits-for relation.

Our definition of priority inversion does not work for the case where different schedulers can assign different priorities to tasks. To understand the problem, consider a system with two resources—a processor and a communications channel—and two tasks τ_1 and τ_2. Suppose the priority assignment $\prec_P$ used by the processor has $\tau_1 \prec_P \tau_2$ and the priority assignment $\prec_C$ used by the channel has $\tau_2 \prec_C \tau_1$. It is possible for $\tau_1 \underset{t}{\rightarrow} \tau_2$ due to an allocation decision by the processor and for $\tau_2 \underset{t}{\rightarrow} \tau_1$ due to an allocation decision by the communications channel. We believe that this scenario should not be considered a priority inversion because no task is being prevented from using a resource by a task that has a lower priority for the use of that resource. However, $(\tau_2 \prec \tau_1) \wedge (\tau_1 \underset{t}{\rightarrow} \tau_2)$ holds, which means there would be a priority inversion according to the definition of Section 3.

7.3. Multiple-Unit Resource Requests

Another limitation of our model is the requirement that tasks request resources sequentially. As a result of this limitation, we have not had to define what constitutes a priority inversion when a task can request multiple resources simultaneously. It is not clear what the correct definition should be. Consider a system consisting of two processors P_1 and P_2 and three tasks τ_1, τ_2 and τ_3. Further, suppose $\tau_2 \prec \tau_1$, τ_1 requires two processors, and τ_2 and τ_3 each require a single processor. It seems reasonable to claim that τ_2 executing on P_1 while P_2 is idle constitutes a priority inversion, since τ_2 holds a resource required by higher-priority task τ_1. What is not clear is whether τ_3 executing on P_1 while P_2 is idle should also be considered a priority inversion. On the one hand, no task holds resources required by a higher-priority task, suggesting that this should not be considered a priority inversion. On the other hand, if this is not considered a priority inversion then the seemingly harmless act of putting idle processor P_2 to work executing τ_2 should not be considered a priority inversion, either. Yet, τ_2 now holds a resource required by τ_1, implying that a priority inversion exists. Note that this ambiguity disappears if integer mappings rather than partial orders are used as priorities.

7.4. Bounded Priority Inversions

This article has been concerned with avoiding priority inversions completely. However, it is sometimes acceptable for priority inversions to occur, provided they are bounded in duration. Suppose a task is being delayed by some lower priority task. If the duration of this delay has a known bound, then we could view the situation as if the delay were included in the fixed overhead associated with allocating the resource. So, by adding to the cost of a request operation any delay due to a priority inversion, an analysis using peer groups would remain valid (despite the priority inversion). The priority inheritance protocols in (Sha, Rajkumar and Lehoczky 1990), for example, provide a way to bound priority inversions under certain circumstances; the protocols of (Munch-Anderson and Zahle 1977) prevent priority inversions of unbounded duration.

Define a *D-bounded priority inversion* to be a priority inversion whose duration is not longer than D and an *unbounded* priority inversion to be one whose duration cannot be so bounded. Avoiding all but D-bounded priority inversions can be easier and less costly than avoiding all priority inversions.[14] For example, detection can be used to eliminate all but D-bounded priority inversions—tasks run their course and periodically a protocol is executed to detect and eliminate priority inversions that may have formed.[15] The frequency with which the detector runs determines the upper bound D on the duration of priority inversions.

In theory, it is easy to build a detector for priority inversions. The detector must construct the composite system graph from the information available to schedulers. In a distributed system with multiple independent schedulers, a distributed snapshot algorithm (Chandy and Lamport 1985) would have to be employed for this purpose. Priority inversion detection is then simply a matter of checking for π-cycles in the composite system graph of a snapshot. Note that a priority inversion might have vanished by the time it is detected because occurrence of a π-cycle is not a stable property (Chandy and Lamport 1985). Such "ghost" priority inversions do not cause problems, however, because they are not unbounded priority inversions. While theoretically feasible, the practicality of this detector is questionable since large amounts of processor time would be consumed if it were to run at any reasonable frequency.

8. Conclusions

This article gives a formal characterization of priority inversion and gives a set of sufficient conditions for its avoidance. Based on these conditions, we have been able to derive new protocols to avoid priority inversion. We have also been able to give conditions to avoid priority inversion in systems with multiple schedulers that do not communicate.

The existence of a theory characterizing priority inversions makes it possible to design both general and application-specific protocols to avoid priority inversions. The theory also permits the consequences of system design decisions to be better understood. For example, we were surprised to find that choosing between an irreflexive partial order and an integer-mapping representation of priority assignment can be significant. We were also surprised that the definition of priority inversion is elusive for systems in which multiple resource requests are possible or in which independent schedulers use different priority assignments.

Acknowledgments

We are grateful to Hideyuki Tokuda, John Lehoczky, and Lui Sha for their encouragement and reactions to an early draft of this article. Nancy Lynch, Rob Cook and two referees have also provided helpful comments on this material.

Notes

1. An irreflexive partial order $\prec$ on a set is an asymmetric, irreflexive, transitive relation on pairs of elements from that set.
2. Most real systems have multiple schedulers, but postulating a single scheduler is not a limitation. It is always possible to model the effect of a collection of schedulers by using a single scheduler that makes allocation decisions using only information that would be available to the relevant scheduler. We return in Section 6 to the subject of multiple schedulers.
3. The transitive closure of a binary relation R is the smallest relation R^+ such that $(a, b) \in R^+$ if and only if $(a, b) \in R$ or there exist $(a, c) \in R$ and $(c, b) \in R^+$.
4. Cycles containing more than one priority edge do not seem to correspond to any interesting property. Cycles consisting of only waits-for edges, on the other hand, model deadlock. Thus, it is possible to have a deadlock without a priority inversion and vice versa.
5. Some authors call this the *Circular Waiting* condition.
6. A *invariant* is an assertion about the program state that is not invalidated by program execution.
7. We write $A \supset B$ to denote the logical implication, A implies B.
8. For an assignment statement $x := E$, (Dijkstra 1976) defines $wp(x := E, Q)$ to be Q^x_E, the result of replacing in Q every free occurrence of x by E.

9. This definition is commonly called *conflict equivalence.* There are other definitions of equivalence, but determining whether a history is serializable under other definitions is computationally expensive (Bernstein, Hadzilacos and Goodman 1987).
10. A concurrency controller can be built that explicitly maintains G, in which case rules for removing edges must be given as well (Bernstein, Hadzilacos and Goodman 1987). Such rules, however, are not important for the discussion in this article.
11. Such a value will exist because we have assumed that timestamps are real numbers. If timestamps are implemented using finite-precision numbers, then such a value may not exist. To avoid this problem, we can modify the code in Figure 3 to expand the range when necessary by aborting the transaction whose timestamp is the lower bound of the interval.
12. Actually, if both operations are writes, the second write can be ignored and no conflict occurs (Thomas 1979).
13. The case where schedulers do not share a common priority assignment is discussed in Section 7.2.
14. Analyzing a scheduling protocol to determine a bound D can be a hard problem, however (Sha, Rajkumar and Lehoczky 1990).
15. Elimination of the priority inversion requires either that resource allocations to tasks be preemptable or that task executions be abortable.

References

Abbott, R. and Garcia-Molina, H. 1989. Scheduling real-time transactions with disk resident data. *Proc. 15th VLDB Conference*. Amsterdam, The Netherlands, pp. 385-396.

Bernstein, P., Hadzilacos, V. and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley.

Bic, L. and Shaw, A.C. 1988. *The Logical Design of Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall.

Chandy, K.M. and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions of Computer Systems,* 3(1):63–75.

Coffman, E.G. Jr. and Denning, P.J. 1973. *Operating Systems Theory.* Englewood Cliffs, NJ: Prentice-Hall.

Dijkstra, E.W. 1976. *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall.

Havender, J.W. 1968. Avoiding deadlock for multitasking systems. *IBM Systems Journal,* 7(2):74–84.

Lampson, B.W. and Redell, D.D. 1980. Experience with processes and monitors in Mesa. *Communications of the ACM,* 23(2):105–117.

Liu, C.L. and Layland, J.W. 1973. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM,* 20:46–61.

Marzullo, K. 1989. Concurrency control for transactions with priority. Technical Report 89-996, Department of Computer Science, Cornell University.

Munch-Anderson, B. and Zahle, T.U. 1977. Scheduling according to job priority with prevention of deadlock and permanent blocking. *Acta Informatica.* 8:153–175.

Sha, L., Rajkumar, R. and Lehoczky, J.P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers.* C-39:1175–1185.

Thomas, R.H. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems.* 4(2):180–209.

Zhao, W., Ramamritham, K. and Stankovic, J.A. 1987. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers.* C-36:949–960.