Technical Report

CMU/SEI-89-TR-018
ESD-89-TR-026

A Real-Time Locking Protocol

Lui Sha
Ragunathan Rajkumar
Sang Son
Chun-Hyon Chang

April 1989

Technical Report
CMU/SEI-89-TR-018
ESD-89-TR-026
April 1989

A Real-Time Locking Protocol

Lui Sha

Real-Time Scheduling in Ada Project

Ragunathan Rajkumar

Carnegie Mellon University

Sang Son

University of Virginia

Chun-Hyon Chang

Kon Kuk University, Seoul, Korea

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through Asset Source for Software Engineering Technology (ASSET) / 1350 Earl L. Core Road ; P.O. Box 3305 / Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / Fax: (304) 284-9001 / e-mail: sei@asset.com / WWW: http://www.asset.com/sei.html

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

A Real-Time
Locking Protocol

Abstract: When a database system is used in a real-time application, the concurrency control protocol must satisfy not only the consistency of shared data but also the timing constraints of the application. In this paper, we examine a priority-driven two-phase lock protocol called the read- or write-priority ceiling protocol. We show that this protocol is free of deadlock, and in addition a high-priority transaction can be blocked by lower priority transactions for at most the duration of a single embedded transaction. We then evaluate system performance experimentally.

1. Introduction

In a real-time database context, concurrency control protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database.

Both concurrency control [2, 3, 4, 5, 7, 16, 17, 18, 20, 21, 23, 26] and real-time scheduling algorithms [10, 11, 13, 14, 15, 19, 22, 27] are active areas of research in their own right. It may seem that the development of a real-time locking protocol is a simple matter of combining priority scheduling with a locking protocol. For example, we may require each transaction to use a well-known concurrency protocol such as the two-phase lock protocol [6] and assign priorities to transactions according to some well-known scheduling algorithms such as the earliest deadline algorithm [19]. Next, we process transactions in priority order. Unfortunately, such an approach may lead to unbounded priority inversion, in which a high-priority task would wait for lower priority tasks for an indefinite period of time.

Example 1: Suppose $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$ are three transactions arranged in descending order of priority, with $\mathbf{T}_1$ having the highest priority. Assume that transaction $\mathbf{T}_1$ and $\mathbf{T}_3$ share the same data object **O**. Suppose that at time t_1 transaction $\mathbf{T}_3$ obtains a write-lock on **O**. During the execution of $\mathbf{T}_3$, the high-priority task $\mathbf{T}_1$ arrives and attempts to read-lock the object **O**. Transaction $\mathbf{T}_1$ will be blocked, since **O** is already write-locked. We would expect that $\mathbf{T}_1$, being the highest priority transaction, will be blocked no longer than the time for $\mathbf{T}_3$ to complete and unlock **O**. However, the duration of blocking may, in fact, be unbounded. This is because transaction $\mathbf{T}_3$ can be preempted by the intermediate-priority transaction $\mathbf{T}_2$ that does not need to access **O**. The preemption of $\mathbf{T}_3$, and hence the blocking of $\mathbf{T}_1$, will continue until $\mathbf{T}_2$ and any other pending intermediate-priority level transactions are completed.

The blocking duration in Example 1 can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long, low-priority transaction starts execution, a high-priority transaction not requiring access to the same set of data objects may be needlessly blocked.[1] An objective of this paper is to design an appropriate priority management protocol for a given concurrency control protocol so that deadlocks can be avoided and the duration of blocking can be tightly bounded.

[1] The priority inversion problem was first discussed by Lampson and Redall [9] in the context of monitors. They suggest that each monitor always be executed at a priority level higher than all tasks that would ever call the monitor.

2. The Read- or Write-Priority Ceiling Protocol

2.1. Basic Concepts

Real-time databases are often used by applications such as tracking. Since tracking operations consist of both signal processing and database accessing, we assume that each instance of a periodic task consists of data-processing code and embedded transactions operating on the database. We assume that an embedded transaction consists of a sequence of read and write operations operating upon the database. A task can have multiple embedded transactions. However, embedded transactions in a task do not overlap. Each embedded transaction will follow the two-phase lock protocol [6], which requires a transaction to acquire all the locks before it releases any lock. Once a transaction releases a lock, it cannot acquire any new lock.

In this section, we also assume that all the tasks are periodic, which models the periodic operation of sensors. In addition, we assume that the database resides in the main memory. We will, however, relax both assumptions in the next section. When tasks are periodic, we assume that their priorities are assigned by the rate monotonic algorithm in which a shorter period task has a higher priority. It was shown in [15] that the rate monotonic algorithm is an optimal static-priority scheduling algorithm for periodic tasks. A high-priority task will preempt the execution of lower priority tasks unless it is blocked by the read- or write-priority ceiling protocol defined later in this report.

With only two-phase locking and priority assignment, we can encounter the problem of unbounded priority inversion as illustrated in Example 1. However, the idea of priority inheritance [24] solves the unbounded priority inversion problem. In the context of preemptive scheduling, a higher priority task τ can preempt the execution of lower priority tasks unless τ is blocked by the locking protocol. The priority inheritance rule states that when the transaction of task τ blocks the execution of higher priority tasks, it executes (inherits) at the highest priority of all the tasks blocked by τ. To illustrate this idea, let us apply this protocol to Example 1. Suppose that task τ_1 is blocked by task τ_3. The priority-inheritance protocol requires task τ_3 to execute its transaction at the priority of task τ_1 until it releases the lock on data object **O**. As a result, task τ_2 will be unable to preempt τ_3. Once task τ_3 unlocks data object **O**, it returns to its assigned priority and will immediately be preempted by τ_1. As we can see, this simple priority-inheritance idea reduces the blocking time of a higher priority task from the entire execution time of lower priority tasks to only the duration of lower priority tasks' embedded transactions.

The second idea is a total priority ordering of active transactions. A transaction embedded in a task is said to be active if it has started but not yet completed its execution. Thus a transaction can be active in one of two ways: executing or being preempted in the middle of its execution. The idea of a total priority ordering is that we want our protocol to ensure that each active transaction is executed at a higher priority level, taking priority inheritance and the read and write semantics into consideration. Together with the first idea, we get the properties of freedom from deadlock and a worst-case blocking of at most a single embedded transaction. We shall refer to the latter property as the *block-at-most-once property.*

To ensure the total priority ordering of active transactions, we define three parameters for each data object in the database: the write-priority ceiling, the absolute priority ceiling and the read- or write-priority ceiling. The write-priority ceiling of a data object **O** is simply the priority of the highest priority task that may write **O**. The absolute priority ceiling of **O** is the priority of the highest priority task that may read or write **O**. The read- or write-priority ceiling of the data object is, however, set dynamically. We shall use the rule that a task τ cannot read or write-lock a data object and execute its transaction unless its priority is higher than the highest priority read- or write-priority ceiling locked by tasks other than τ. We shall refer to this rule as the *ceiling rule*.

When a task write-locks a data object **O**, **O** cannot be read or written by another task. To ensure that, we can set the read- or write-priority ceiling of **O** equal to its absolute priority ceiling. Since the absolute priority ceiling of **O** is equal to the priority of the highest priority task that may either read or write **O**, it prevents another task from reading or writing **O** until the lock on **O** is released. Similarly, when a task read-locks a data object **O**, **O** cannot be written by another task. To ensure this, when a data object **O** is read-locked by a transaction, we set the read- or write-priority ceiling of **O** equal to the write-priority ceiling of **O**. Since the write-priority ceiling equals the priority of the highest priority task that may write **O**, the ceiling rule prevents another transaction from writing **O**. Read transactions with priorities higher than the write-priority ceiling of **O** can share the read-lock on **O** however. On the other hand, this protocol forbids read transactions with priorities lower than or equal to the write-priority ceiling of **O** from sharing the read-lock on **O**. This is important. Should we allow these low-priority read transactions to share a read-lock on **O**, when the high-priority write transaction arrives and attempts to write **O**, it has to wait for multiple readers. That is, a task can be blocked by multiple lower priority embedded transactions. As we shall see in Theorem 8, longer blocking durations lead to lower schedulability.

From the viewpoint of priority management, the objective of the read- or write-priority ceiling is to ensure that each embedded transaction is executed at a higher priority level than the priority levels which can be inherited by preempted transactions. When a transaction **T** write-locks a single data object **O**, the read- or write-priority ceiling of **O** represents the highest priority that **T** can inherit through **O**. For example, when **T** write-locks **O**, it can block the highest priority task τ_H that may read or write **O** and hence inherit the priority of τ_H. Therefore, the read- or write-priority ceiling of a write-locked object is defined to be equal to the absolute priority ceiling. Alternatively, let a low-priority transaction hold a read-lock on a data object **O** and let transaction $\mathbf{T}_w$ be the highest priority transaction that may request a write-lock on **O**. Transaction **T** can block τ_w and inherit the priority of τ_w. Therefore, the read- or write-priority ceiling of a read-locked data object is defined as the data object's write-priority ceiling.

Under the read- or write-priority ceiling protocol, a task τ cannot acquire a lock and execute its embedded transaction unless its priority is higher than all the read- or write-priority ceilings of the data object locked by tasks other than τ. Since the highest read- or write-priority ceiling of the locked data objects represents the highest priority level that the currently active transactions can execute (inherit), we ensure that the transaction of τ executes at a priority level higher than all the preempted transactions, should τ be able to execute its transaction.

Such total priority ordering of active transactions leads to some interesting behavior. For example, the read- or write-priority ceiling protocol may forbid a transaction from locking an *unlocked* data object. At first sight, this seems to introduce unnecessary blocking. However, this is, in fact, the "insurance premium" for preventing mutual deadlock and the block-at-most-once property.

Example 2: Suppose that we have three tasks, τ_0, τ_1, and τ_2, arranged in descending order of priority. In addition, there are two data objects **O**$_1$ and **O**$_2$.

$\tau_0 = \{ \cdots,$ *write-lock*$(O_0), \cdots,$ *unlock*$(O_0), \cdots \}$

$\tau_1 = \{ \cdots,$ *read-lock*$(O_1), \cdots,$ *write-lock*$(O_2), \cdots,$ *unlock*$(O_2), \cdots,$ *unlock*$(O_1), \cdots \}$

$\tau_2 = \{ \cdots,$ *read-lock*$(O_2), \cdots,$ *write-lock*$(O_1), \cdots,$ *unlock*$(O_1), \cdots,$ *unlock*$(O_2), \cdots \}$

The sequence of events described in Example 2 is depicted in Figure 2-2. A line at a low level indicates that the corresponding task is blocked or has been preempted by a higher priority task. A line raised to a higher level indicates that the task is executing. The absence of a line indicates that the task has not yet arrived or has completed. Shaded portions indicate execution of transactions.

First, we establish the priority ceiling of each of the data objects. The write-priority ceiling and absolute priority ceiling for data object **O**$_1$ are the priorities of tasks τ_2 and τ_1, P_2 and P_1, respectively. For data object **O**$_2$, both the write and absolute priority ceiling are equal to P_1. For data object **O**$_0$, both ceilings are equal to P_0.

Suppose that at time t_0, task τ_2 starts its execution. At time t_1, τ_2 has executed read-lock (**O**$_2$) and the read- or write-priority ceiling of **O**$_2$ is set at the write-priority ceiling of **O**$_2$, i.e., P_1. Having locked **O**$_2$, task τ_2 starts executing its embedded transaction **T**$_2$. At this instant, task τ_1 is initiated and preempts transaction **T**$_2$. However, when task τ_1 tries to execute its embedded transaction at time t_2 by making an indivisible system call to execute read-lock (**O**$_1$), the scheduler will find the priority of P_1 of task τ_1 is *not* higher than the read- or write-priority ceiling of *locked* data object **O**$_2$, which was set at P_1. Hence, the scheduler suspends transaction τ_1 without letting it lock **O**$_1$. Note that τ_1 is blocked outside its embedded transaction. Transaction **T**$_2$ now *inherits* the priority of task τ_1 and resumes execution. Since τ_1 is denied the lock on **O**$_1$ and suspended instead, a potential deadlock between **T**$_1$ and **T**$_2$ is prevented. If τ_1 were granted the lock on **O**$_1$, then τ_1 would later wait for τ_2 to release the lock on **O**$_2$, while τ_2 would wait for τ_1 to release the lock on **O**$_1$.

On the other hand, suppose that at time t_3, while **T**$_2$ is still in its transaction, the highest priority task τ_0 arrives and attempts to write-lock data object **O**$_0$. Since the priority of τ_0 is

Figure 2-1: Sequence of Events Described in Example 2

higher than the read- or write-priority ceiling of locked data object $\mathbf{O}_2$, task τ_0's transaction $\mathbf{T}_0$ will be granted the lock on the data object $\mathbf{O}_0$. Task τ_0 will therefore continue and execute its transaction, thereby effectively preempting $\mathbf{T}_2$ in its transaction and not encountering any blocking. At time t_4, $\mathbf{T}_0$ completes execution and $\mathbf{T}_2$ is awakened, for $\mathbf{T}_1$ is blocked by $\mathbf{T}_2$. $\mathbf{T}_2$ continues execution and write-locks $\mathbf{O}_1$. At time t_5, $\mathbf{T}_2$ releases $\mathbf{O}_1$. At time t_6, when $\mathbf{T}_2$ releases $\mathbf{O}_2$, task τ_2 resumes its assigned priority. Now $\mathbf{T}_1$ is signaled. Having a higher priority, it preempts $\mathbf{T}_2$ and completes execution. Finally, $\mathbf{T}_2$ resumes and completes.

Note that in the above example, τ_0 is never blocked. τ_1 was blocked by the lower priority task τ_2 during the intervals $[t_2, t_3]$ and $[t_4, t_6]$.[2] However, these two intervals correspond to

[2] The interval $[t_3, t_4]$ is not considered blocking for τ_1 since it was only preempted by the higher priority task τ_0.

the duration that $\mathbf{T}_2$ needs to lock the two data objects. Thus, the blocking duration of τ_1 is equal to the duration of a single embedded transaction of a lower priority transaction $\mathbf{T}_2$, even though the actual blocking occurs over disjointed time intervals. It is, indeed, a property of this protocol that any task τ can be blocked by, at most, one lower priority embedded transaction until τ suspends itself or completes.

2.2. Definitions and Properties

Having reviewed the basic concepts, we now review our assumptions and state the notation used. We assume that we are given a centralized database system and there is a set of periodic tasks. In addition, we assume that all the data objects reside in the main memory. Since tracking operations consist of both signal processing and database accessing, we assume that each instance of a periodic task executes signal processing codes and embedded transactions. We assume that the rate-monotonic algorithm is used to assign a priority to each task. This algorithm assigns higher priorities to tasks with shorter periods and is an optimal static-priority algorithm for periodic tasks [15]. If two tasks are ready to run on a processor, the higher priority task will run. Equal priority tasks are run in a FCFS (first come, first served) order. We also assume that a transaction does not attempt to lock an object that it has already locked and thus deadlock with itself. We also assume that either multiple read-locks or a single write-lock can be held on a data object.

Notation: We denote the given tasks as an ordered set $\{\tau_1, \cdots, \tau_n\}$ where the tasks are listed in descending order of priority, with τ_1 having the highest priority.

Notation: We use $\mathbf{T}_{i,j}$ to denote an embedded transaction of task τ_i. We will also use the simplified notation $\mathbf{T}_i$ when the identity of *j* is not important.

Notation: We use the notation P_i to denote the priority of task τ_i.

Definition: The lock on a data object can either be a read-lock or a write-lock. A task τ that holds a read-lock (write-lock) on a data object **O** is said to have read-locked (write-locked) object **O**. The *write-priority ceiling* of a data object is defined as the priority of the highest priority task that may write this object. The *absolute priority ceiling* is defined as the priority of the highest priority task that may either read or write this data object. When a data object **O** is write-locked, the *read- or write-priority ceiling* of **O** is defined to be equal to the absolute priority ceiling of **O**. When a data object **O** is read-locked, the read- or write-priority ceiling of **O** is defined to be equal to the write priority ceiling of **O**.

Having stated our objectives and our assumptions, we now define the read- or write-priority ceiling protocol.

1. Task τ, having the highest priority among the tasks ready to run, is assigned the processor. Before task τ starts to execute an embedded transaction **T**, task τ must first obtain the locks on the data objects that it accesses. In addition, each embedded transaction follows the two-phase lock protocol and all the locks will be released at the end of the transaction.

2. Let $\mathbf{O}_H$ be the data object with the highest read- or write-priority ceiling of all data objects currently locked by transactions other than those of τ. When the transaction of task τ attempts to lock a data object **O**, τ will be blocked and the lock on an object **O** will be denied, if the priority of task τ is not higher than the read- or write-priority ceiling of data object $\mathbf{O}_H$. In this case, task τ is said to be blocked by the task whose transaction holds the lock on $\mathbf{O}_H$. If the priority of task τ is higher than the read- or write-priority ceiling of $\mathbf{O}_H$, then τ is granted the lock on **O**[3].

3. A task τ and its transaction **T** uses the priority assigned to τ, unless **T** blocks higher priority transactions. If transaction **T** blocks higher priority tasks, **T** *inherits* P_H, the highest priority of the tasks blocked by **T**. Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of original priority must be indivisible.

4. When a task τ does not attempt to execute an embedded transaction, it can preempt other tasks and their embedded transactions executing at a lower priority level.

Remark: Under this protocol, we need not explicitly check for the possibility of read-write conflicts. For instance, when an object **O** is write-locked by a task τ, the read- or write-priority ceiling of **O** is equal to the priority of the highest priority task that can access **O**. Hence, the protocol will block a higher priority task that may want to write or read **O**. On the other hand, suppose that the object **O** is read-locked by τ. Then, the read- or write-priority ceiling of **O** is equal to the highest priority task that may write **O**. Hence, a task that attempts to write **O** will have a priority no higher than the read- or write-priority ceiling and will be blocked. Only the tasks that read **O** and have priority higher than the read- or write-priority ceiling will be allowed to read-lock **O**, and read-locks are compatible.

Under the read- or write-priority ceiling protocol, mutual deadlock of transactions cannot occur and each task can be blocked by at most one embedded transaction until it completes or suspends itself. We shall now prove both these properties of the read- or write-priority ceiling protocol.

Lemma 1: Under the read- or write-priority ceiling protocol, each transaction will execute at a higher priority level than the level that the preempted transactions can inherit.

Proof: By the definition of the read- or write-priority ceiling protocol, when a task τ locks a set of data objects, the highest priority level τ can inherit is equal to the highest read- or write-priority ceiling of the data objects locked by τ. Hence, when the priority of task τ_H is higher than the highest read- or write-priority ceiling of the data objects locked by a transaction **T** of task τ, the transactions of τ_H will execute at a priority that is higher than the preempted transaction **T** can inherit.

[3] Under this condition, there will be no read-write conflict on the object **O**, and we need not check if **O** has been locked. See the remark that follows the protocol definition.

Theorem 2: There is no mutual deadlock under the read- or write-priority ceiling protocol.

Proof: Suppose that a mutual deadlock can occur. Let the highest priority of all the tasks involved in the deadlock be *P*. Due to the transitivity of priority inheritance, all the tasks involved in the deadlock will eventually inherit the same highest priority *P*. This contradicts Lemma 1.

Lemma 3: Under the read- or write-priority ceiling protocol, until task τ either completes its execution or suspends itself, task τ can be blocked for at most a single embedded transaction of a lower priority task τ_L, even if τ_L has multiple embedded transactions.

Proof: Suppose that task τ is blocked by a lower priority task τ_L. By Theorem 2, there will be no deadlock and hence task τ_L will exit its current transaction at some instant t_1. Once task τ_L exits its transaction at time t_1, task τ_L is preempted by τ. Since τ_L is no longer within a transaction, it cannot inherit a higher priority than its own priority unless it executes another transaction. However, τ_L cannot resume execution until τ completes or suspends itself. The Lemma follows.

Theorem 4: Under the read- or write-priority ceiling protocol, a task τ can be blocked by at most a single embedded transaction of one lower priority task until either τ completes its execution or suspends itself.

Proof: Suppose that τ is blocked by *n* lower priority transactions. Given Lemma 3, τ must be blocked by the transactions of *n* different lower priority tasks, τ_1, ..., τ_n, where the priority of τ_i is assumed to be higher than or equal to that of τ_{i+1}. Under the protocol, a task not in a transaction can always be preempted by a higher priority task. Hence, a lower priority task cannot block a higher priority task unless it is already in its transaction. Therefore, tasks τ_1, ..., τ_n must be in their transactions when τ arrives. By assumption, τ is blocked by τ_n and τ_n inherits the priority of τ. Since τ can be blocked by τ_n, the priority of task τ cannot be higher than the highest priority *P* that can be inherited by τ_n. On the other hand, by Lemma 1, the priority of task τ_{n-1} is higher than *P*. It follows that the priority of task τ_{n-1} is higher than that of task τ. This contradicts the assumption that the priority of τ is higher than that of tasks τ_1, ..., τ_n.

Corollary 5: If a task τ_i suspends itself at most *k* times, then the above theorem holds with the duration of blocking equal to *k+1* embedded transactions.

Remark: The read- or write-priority ceiling protocol is selectively restrictive on the sharing of read-locks. The reason is that a direct application of the read and write semantic can lead to prolonged durations of blocking. For example, suppose that we have a single write transaction at the highest priority level and ten lower priority read transactions. If we let ten transactions concurrently hold read-locks on data object **O**, then when a higher priority task arrives later and attempts to write **O**, it has to wait for all ten of these transactions to complete. That is, some forms of concurrency can lengthen the worst-case duration of blocking, resulting in poorer schedulability.

We now develop a set of *sufficient* conditions under which a set of periodic tasks with hard deadlines at the end of the periods can be scheduled by the rate-monotonic algorithm [15] when the read- or write-priority ceiling protocol is used.

Liu and Layland propose the following theorem, which was proved under the assumption of independent tasks; i.e., there is no blocking due to data sharing and synchronization.

> **Theorem 6:** A set of *n* periodic tasks scheduled by the rate-monotonic algorithm can always meet their deadlines if
>
> $$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \le n(2^{1/n}-1)$$
>
> where C_i and T_i are the execution time and period of task τ_i respectively.

Theorem 6 offers a sufficient (worst-case) condition that characterizes the rate-monotonic schedulability of a given periodic task set. An exact characterization of rate-monotonic schedulability can be found in [12].

When tasks are independent of one another and do not access shared data, Theorem 6 provides us with the condition under which a set of *n* periodic tasks can be scheduled by the rate-monotonic algorithm.[4] Although this theorem takes into account the effect of a task being preempted by higher priority tasks, it does not consider the effect of blocking caused by lower priority tasks upon schedulability analysis. We now consider the effect of blocking.

> **Theorem 7:** A lower priority write transaction $\mathbf{T}_w$ can block a higher priority task τ with priority *P*, if and only if $\mathbf{T}_w$ may write-lock a data object whose absolute priority ceiling is higher than or equal to *P*. A lower priority read transaction $\mathbf{T}_r$ can block a higher priority task τ with priority *P*, if and only if $\mathbf{T}_r$ may read-lock a data object whose write-priority ceiling is higher than or equal to *P*.
>
> **Proof:** It directly follows from the definitions of the read- or write-priority ceiling protocol.

Let *Z* be the set of embedded transactions that could block task τ. By Theorem 4, task τ can be blocked for at most the duration of a single element in *Z* if it does not suspend itself. Hence the worst-case blocking time for τ is the duration of the longest embedded transaction in *Z* when τ does not suspend itself. If the task τ suspends itself *k* times, then the worst-case blocking time is equal to the sum of the *k+1* longest elements in *Z*. We denote this worst-case blocking time of task τ_i as B_i. Note that given a set of *n* periodic tasks, B_n = 0, since there is no lower priority task to block τ_n.

Theorem 6 can now be generalized in a straightforward fashion. In order to test the schedulability of τ_i, we need to consider both the preemptions caused by higher priority tasks and blocking by lower priority tasks, along with the utilization of τ_i. The blocking of any instance of τ_i is bounded by B_i. Thus, Theorem 6 becomes

> **Theorem 8:** Suppose that a task does not suspend itself from initiation to completion. A set of *n* periodic tasks using the read- or write-priority ceiling protocol can

[4] That is, the conditions under which all the instances of all the *n* tasks will meet their deadlines.

be scheduled by the rate monotonic algorithm if the following conditions are satisfied:

$$\forall\ i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

Proof: Suppose that for each task τ_i the inequality is satisfied. It follows that the inequality of Theorem 6 will also be satisfied with $n = i$ and C_i replaced by $C_{i,j} = (C_i + B_i)$. That is, in the absence of blocking, any instance of task τ_i will still meet its deadline even if it executes for $(C_i + B_i)$ units of time. It follows that task τ_i, if it executes for only C_i units of time, can be delayed by B_i units of time and still meet its deadline. Hence the theorem follows.

Remark: The first *i* terms in the above inequality constitute the effect of preemptions from all higher priority tasks and the execution time of τ_i, while B_i of the last term represents the worst case blocking time due to *all* lower priority tasks for one instance of task τ_i.

Corollary 9: A set of *n* periodic tasks using the read- or write-priority ceiling protocol can be scheduled by the rate monotonic algorithm if the following condition is satisfied:

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} + max(\frac{B_1}{T_1}, \cdots, \frac{B_{n-1}}{T_{n-1}}) \le n(2^{1/n} - 1)$$

Proof: Since $n(2^{1/n} - 1) \le i(2^{1/i} - 1)$ and $max(\frac{B_1}{T_1}, \cdots, \frac{B_{n-1}}{T_{n-1}}) \ge \frac{B_i}{T_i}$, if this inequality holds then all the inequalities in Theorem 8 also hold.

3. Performance Evaluation

In the previous section, we have assumed that all the tasks are periodic and that all the database objects are in the main memory. In this section, we relax these two assumptions and examine the performance of read- or write-priority ceiling protocol versus the performance of the two-phase lock protocol with and without priority assignments to tasks. This experiment investigates the performance of the performance characteristics in a single-site database system, using the University of Virginia's database prototyping tool [25]. In this experiment, we assume that the transaction system is a soft real-time system, in the sense that we do not guarantee the transaction deadlines. However, each transaction has a deadline and we assume that there will be no value in completing a transaction once it has missed its deadline. Transactions that miss the deadline are aborted, and disappear from the system immediately with some abort cost. In this experiment, each task consists of a single transaction with an execution profile that alternates database access requests with equal computation requests, and some processing requirement for termination (either commit or abort). Thus the total processing time of a transaction is directly related to the number of data objects accessed.

In the experiments, transactions are generated with exponentially distributed interarrival times, and the data objects updated by a transaction are chosen uniformly from the database. Due to space considerations, we cannot present all our results but have selected the graphs which best illustrate the difference and performance of the algorithms. For example, we have omitted the results of an experiment that varied the size of the database, and thus the number of conflicts. This is because they only confirm and do not increase the knowledge yielded by other experiments. The measure of merit is the throughput and the percentage of transactions that miss their deadlines. The measure of throughput is records accessed per second for successful transactions, not in transactions per second. This is to account for the fact that bigger transactions need more database processing.

For each experiment and for each algorithm tested, we collected performance statistics and averaged over 10 runs. We have used the transaction size (the number of data objects a transaction needs to access) as one of the key variables in the experiments. It varies from a small fraction up to a relatively large portion (10%) of the database so that conflict would occur frequently. The high conflict rate allows synchronization protocols to play a significant role in the system performance. We chose the arrival rate so that protocols are tested in a heavily loaded system, because when designing real-time systems, one must consider high-load situations. Even though high-load situations may not arise frequently, one would like to have a system that misses as few deadlines as possible when the system is under stress [1].

In Figures 3-1 and 3-2, the throughput of the priority-ceiling protocol (C), the two-phase locking protocol with priority mode (P), and the two-phase locking protocol without priority mode (L), is shown for transactions of different sizes with balanced workload and I/O bound workload. The two important factors affecting the performance of locking protocols are their abilities to resolve the locking conflicts and to perform the I/O and transactions in parallel.

When the transaction size is small, there is little locking conflict and the problem such as deadlock and priority inversion has little effect upon the overall performance of a locking protocol. On the other hand, when transaction size becomes large, the probability of locking conflict rises rapidly. In fact, the probability of deadlocks goes up with the fourth power of the transaction size [8]. Hence, we would expect that the performance of protocols will be dominated by their abilities to handle locking conflicts when transaction size is large.

Figure 3-1: Balanced Workload

As illustrated in Figures 3-1 and 3-2, the performance of the two-phase lock algorithm, with or without priority assignments to transactions, degrades very fast when transaction size increases. This can be attributed to the inability of this protocol to prevent deadlock and priority inversions. On the other hand, the read- or write-priority ceiling protocol handles locking conflicts very well. The protocol is free from deadlocks and exhibits the block-at-most-once property. Hence, this protocol performs much better than the two-phase lock protocol when the transaction size is large. The main weakness of the read- or write-priority ceiling protocol is its inability to perform I/O and transactions in parallel. For example, suppose that transaction **T** has locked $\mathbf{O}_1$ and it now wants to lock data object $\mathbf{O}_2$. Unfortunately, $\mathbf{O}_2$ is not in the main memory. As a result, **T** is suspended. However, neither are transactions with priorities lower than the read- or write-priority ceiling of $\mathbf{O}_1$ allowed to execute. This could lead to the idling of the processor until either $\mathbf{O}_2$ is transferred to the main memory or a transaction whose priority is higher than the read- or write-priority ceiling arrives. We call this *I/O blocking*. When transaction size is small, the locking conflict rate is small. Hence, the two-phase lock performs well. However, due to I/O blocking the through-

Figure 3-2: I/O Bounded Workload

put of read- or write-priority ceiling protocol is not as good as that of two-phase lock protocol, especially when the workload is I/O bounded.

Since I/O cost is one of the key parameters in determining performance, we have investigated an approach to improve system performance by performing the I/O operation before locking. This is called the *intention I/O*. In the intention mode of I/O operation, the system "pre-fetches" data objects that are in the access lists of transactions submitted, without locking them. This approach will reduce the locking time of data objects, resulting in higher throughput. As shown in Figure 3-3, intention I/O improves throughput of both the two-phase locking and the ceiling protocol. However, improvement in the ceiling protocol is much more significant. This is because intention I/O effectively solves the I/O blocking problem of the read- or write-priority ceiling protocol.

Another important performance statistic is the percentage of transactions missing deadlines, since the synchronization protocol in real-time database systems should satisfy the timing constraints of individual transactions. In our experiments, each transaction’s deadline is set proportional to its size and system workload (number of transactions), and the transaction with the shorter deadline is assigned a higher priority. As shown in Figure 3-4, the percentage of transactions missing deadlines increases sharply for the two-phase locking protocol as the transaction size increases due to the protocol’s inability to deal with deadlock and to give preference to transactions with shorter deadlines. Two-phase lock with priority assignment performs somewhat better, because the timing constraints of transactions are con-

Figure 3-3: With Intention I/O

sidered, although the deadlock and priority-inversion problems still handicap performance. The read- or write-priority ceiling protocol has the best relative performance because it addresses both the deadlock and priority-inversion problem.

Figure 3-4: Percentage of Missing Deadline

4. Conclusions

Real-time database is an important area of research, with applications ranging from surveillance to reliable manufacturing and production control. In this paper, we have investigated the read- or write-priority ceiling protocol, which integrates the two-phase lock protocol with priority-driven real-time scheduling. We have shown that this protocol is free from mutual deadlock and that a task τ can be blocked for at most the duration of a single embedded transaction of a lower priority task until τ suspends itself or completes. We have also developed schedulability bounds for periodic tasks in a centralized in-core database. Finally, we experimentally evaluated the performance of this protocol when the tasks are invoked aperiodically and the database is no longer in-core.

References

[1] Abbott, R. and Garcia-Molina, H.
Scheduling Real-Time Transactions: A Performance Study.
Proceedings of VLDB Conference, pp 1-12 , September 1988.

[2] Attar, R., Bernstein P. A., and Goodman, N.
Site Initialization, Recovery and Backup in a Distributed Database System.
IEEE Transaction on Software Engineering , November 1984.

[3] Beeri, C., Bernstein, P.A., Goodman, N., and Lai, M. Y.
A Concurrency Control Theory for Nested Transactions.
ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing , 1983.

[4] Bernstein, P. A., Shipman, D. W., and Wong, W. S.
Formal Aspects of Serializability in Database Concurrency Control.
IEEE Transactions on Software Engineering :203 - 216, 1979.

[5] Bernstein, P. A., Hadzilacos, V., and Goodman, N.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley Publication Co. , 1987.

[6] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L.
The Notion of Consistency and Predicate Lock in a Database System.
CACM 19, No 11, November 1976.

[7] Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2 , June 1983.

[8] Gray, J., et al.
A Straw Man Analysis of Probability of Waiting and Deadlock.
IBM Research Report, RJ 3066 , 1981.

[9] Lampson, B. W. and Redell, D. D.
Experiences with Processes and Monitors in Mesa.
Communications of the ACM 23[2]: 105-117 , February 1980.

[10] Lehoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review, Special Issue 14, No. 1, May 1986.

[11] Lehoczky, J. P., Sha, L., and Strosnider, J.
Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

[12] Lehoczky, J. P., Sha, L., and Ding, Y.
The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.
Technical Report, Department of Statistics, Carnegie Mellon University, 1987.

[13] Leinbaugh, D. W.
Guaranteed Response Time in a Hard Real-Time Environment.
IEEE Transactions on Software Engineering , January 1980.

[14] Leung, J. Y. and Merrill, M. L.
A Note on Preemptive Scheduling of Periodic, Real-Time Tasks.
Information Processing Letters 11 [3]:115 - 118, November 1980.

[15] Liu, C. L. and Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.
JACM 20 [1]:46 - 61, 1973.

[16] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4 , December 1983.

[17] Mohan, C., Russell, D., Kedem, Z. M., and Silberschatz, A.
Lock Conversion in Non-Two-Phase Locking Protocols.
IEEE Transaction on Software Engineering , January 1985.

[18] Mohan, C., Lindsay, B., and Obermarck, R.
Transaction Management in The R^* Distributed Database Management System.
ACM Transactions on Database Systems 11, No. 4, December 1986.

[19] Mok, A. K.
Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.
PhD thesis, Massachusetts Institute of Technology, 1983.

[20] Papadimitriou, C. H. and Kanellakis, P. C.
On Concurrency Control by Multiple Versions.
ACM Transaction on Database Systems , March 1984.

[21] Papadimitriou, C.
The Theory of Database Concurrency Control.
Computer Science Press, 1986.

[22] Ramaritham, K. and Stankovic, J. A.
Dynamic Task Scheduling in Hard Real-Time Distributed Systems.
IEEE Software , July 1984.

[23] Schwarz, P.
Transactions on Typed Objects.
PhD thesis, Department of Computer Science, Carnegie Mellon University, 1984.

[24] Sha, L., Rajkumar, R., and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report, Department of Computer Science, Carnegie Mellon University , 1987.

[25] Son, S. H.
A Message-Based Approach to Distributed Database Prototyping.
Fifth IEEE Workshop on Real-Time Software and Operating Systems :71-74, May 1988.

[26] Weihl, W. E. and Liskov, B.
Specification and Implementation of Resilient Atomic Data Types.
Proceedings of The SIGPLAN Symposium on Programming Language Issues , June 1983.

[27] Zhao, W., Ramamritham, K., and Stankovic, J.
Preemptive Scheduling Under Time and Resource Constraints.
IEEE Transactions on Computers , August 1987.

Table of Contents

List of Figures