Interruptible Critical Sections for Real-time Systems

Theodore Johnson
Dept. of Computer and Information Science
University of Florida

Abstract

In this paper, we present a new approach to synchronization in real-time systems. Existing methods for synchronization in real-time systems are pessimistic, and use blocking to enforce concurrency control. Protocols such as the priority ceiling protocol have been proposed to reduce the priority inversion that occurs when low priority tasks block high priority tasks. However, the priority ceiling protocol still allows a low priority task to block a high priority task, and requires the use of a static-priority scheduler. We propose optimistic synchronization methods as an alternative to pessimistic synchronization methods. Our synchronization algorithms never allow a low priority task to block a high priority task, and can be used with dynamic-priority schedulers. We show how the current research in non-blocking concurrent objects and in low-overhead uniprocessor synchronization can be synthesized to implement low-overhead optimistic synchronization.

1 Introduction

The scheduling of independent real-time tasks is well understood, as optimal scheduling algorithms have been proposed for periodic and aperiodic tasks on uniprocessor [7, 9] and multiprocessor systems [8, 4, 15]. However, if the tasks communicate through shared critical sections, a low-priority task that holds a lock may block a high priority that requires the lock, causing a *priority inversion*. In this paper, we present a method for real-time synchronization that avoids priority inversions.

Rajkumar, Sha, and Lehoczky [19] have proposed the Priority Ceiling Protocol (PCP) to minimize the effect of priority inversion. The *priority ceiling* of a semaphore S is the priority of the highest priority task that will ever lock S. A task may lock a semaphore only if its priority is higher than the priority ceiling of all locked semaphores (except for the semaphores that it has locked). The PCP guarantees that a task will be blocked by a lower priority task at most once during its execution. However, the tasks must have static priorities in order to apply the Priority Ceiling Protocol. In addition, blocking for even the duration of one critical section may be excessive. Rajkumar, Sha, and Lehoczky have extended the Priority Inheritance Protocol to work in a multiprocessor system [18].

Blocking-based synchronization algorithms have been extended to work with dynamic-priority schedulers. Baker [2] presents a pre-allocation based synchronization algorithm that can manages resources with multiple instances. A task's execution is delayed until the scheduler can guarantee that the task can execute without

blocking a higher priority task. Tripathi and Nirkhe [21], and Faulk and Parnas [10] also discuss pre-allocation based scheduling methods. Chen and Lin [5] extend the Priority Inheritance Protocol to permit dynamically-assigned priorities. Chen and Lin [6] extend the protocol in [5] to account for multiple resource instances.

Previous approaches to real-time synchronization suffer from several drawbacks. First, a high-priority task might be forced to wait for a low-priority task to complete a critical section. Mercer and Tokuda [14] note that the blocking of high-priority tasks must be kept to a minimum in order to ensure the responsiveness of the real-time system. Jeffay [12] discusses the additional feasibility conditions required if tasks have pre-emption constraints. Second, dynamic-priority scheduling algorithms are feasible with much higher CPU utilizations than static-priority scheduling algorithms [7], and dynamic-priority schedulers mught be required for aperiodic tasks. The simple Priority Inheritance Protocol of Rajkumar, Sha, and Lehoczky [19] can be applied to static-priority schedulers only. The dynamic-priority synchronization protocols [5, 6, 2] are complex, and must be closely integrated with the scheduling algorithm.

In this paper, we present a different approach to synchronization, one which guarantees that a high-priority task never waits for a low-priority task at a critical section. We introduce the idea of an *Interruptable Critical Section* (ICS), which is a critical section protected by optimistic concurrency control instead of by blocking. A task calculates its modifications to the shared data structure, then attempts to *commit* its modification. If a higher priority task previously committed a conflicting modification, the lower priority task fails to commit, and must try again. Otherwise, the task succeeds, and continues in its work. The synchronization algorithms are not tied to the scheduling algorithm, simplifying the design of the real-time operating system. We synthesize several recent developments in uniprocessor and multiprocessor synchronization to present a low-overhead method for implementing interruptible critical sections.

Our method has several limitations, which we explicitly state here. First, our methods are intended to protect shared data structures. An ICS cannot be directly used to reserve access to a resource. The code that reserves a resource can be protected by an ICS, but contention for the resource might cause blocking, and thus priority inversion. A blocking-based protocol (such as those in [19, 2, 6]) should be applied in this instance. Alternatively, resources can be pre-allocated, or enough resources supplied to the system to ensure that no process will block. Similarly, an ICS cannot be used to ensure exclusive access to a device. However, an ICS can be used to protect the data structures used to submit requests to a device driver. Second, our method requires that all critical sections be written using a special protocol. However, we provide transformations to turn a blocking-based critical section into an interruptable critical section, and the transformations are sufficiently straight-forward that they can be made automatically. Third, optimistic

synchronization is not free. A small amount of extra work must be performed by the kernel during a context switch. Though high priority operations are never blocked, low priority operations can be aborted, leading to an increase in critical section execution times. We provide some estimates on this increase.

2 Restartable Atomic Sequences

We build our optimistic synchronization methods on *Restartable Atomic Sequences* (RAS) [3]. An RAS is a section of code that is re-executed from the beginning if a context occurs while a process is executing in the code section. The re-execution of an RAS is enforced by the kernel context-switch mechanism. If the kernel detects that the process program counter is within a RAS on a context switch, the kernel sets the program counter to the start of the RAS. Bershad et al. show that an RAS implementation of an atomic test-and-set has better performance than a hardware test-and-set on many architectures, and is much faster than kernel-level synchronization [3].

We note that the idea of kernel support for critical sections is well established. In 4.3BSD UNIX, a system call that is interrupted by a `signal` is restarted using the `longjump` instruction [13]. Anderson et al. [1] argue that the operaing system support for parallel threads should recognize that a pre-empted thread is executing in a critical section, and execute the pre-empted thread until the thread exits the critical section. In addition, Moss and Kohler coded several of the run-time support calls of the Trellis/Owl language so that they could be restarted if interrupted [16].

We indicate a restartable region by explicitly declaring it so:

```
restartable{
   stmt1;
      :
   stmtn;
     }
```

Bershad et al. propose the RAS as a mechanism for fast user-level synchronization in a uniprocessor system. The authors present a program segment that implements an atomic test-and-set on a processor that has no atomic read-modify-write instructions. We observe that we can atomically calculate an arbitrary function of local and global data and perform an arbitrary update on one global memory location (not necessarily fixed in advance). The following code segment atomically computes return values `r1` through `rn`, and performs one update into shared memory (all variables in the code segment are stored locally). If a task is executing the code segment and is pre-empted, the task will re-execute the code segment and re-compute the return values and the global update. The last instruction performs the update, so the code segment will

not re-execute after the update has been performed. Thus, the update is performed exactly once, and all return values are computed atomicly with the update.

```
restartable{
        r1 := F1(···)
        r2 := F2(···)
             :
        rn := Fn(···)
        lhs := L(···)
        temp := G(···)
        &lhs:= temp
     }
```

As an example, we can implement a shared stack as an ICS by using the following code:

```
struct stack_elem{
     data item;
     struct stack_elem *next;
  } *sp

push(elem){
stack_elem *elem
restartable{
        elem→next=sp;
        sp=elem;
     }
}

stack_elem *pop(){
struct stack_elem *temp
restartable {
        temp=sp;
        if(sp!=NULL)
           sp=sp→next;
     }
  return(temp);
}
```

3 Implementing Interruptible Critical Sections

We have seen that we can write interruptible critical sections that perform only one update to global memory. For the ICS technique to be useful, we need methods for implementing more interesting critical sections. On the surface, it seems that creating an ICS that performs two writes is a difficult task, because a process executing in the ICS may be interrupted after the first but before the second write. Setting locks to avoid pre-emption is unacceptable, since the goal is to create arbitrarily interruptible critical sections. We will instead make use of non-blocking synchronization algorithms.

Herlihy [11] introduces the idea of non-blocking concurrent objects. An algorithm for a non-blocking object provides the guarantee that one of the process which accesses the object makes progress in a finite

number of steps. Herlihy provides a method for implementing non-blocking objects that swaps in the new value of the object in a single write.

Prakash, Lee, and Johnson [17] devise a non-blocking queue that permits enqueuers to work concurrently with dequeuers. The enqueue operation requires two writes: one write to attach the new element and one write to move the tail pointer. If an operation detects that it is blocked by a half-finished operation, the new operation will *altruistically* complete the half-finished operation and unblock itself.

Turek, Shasha, and Prakash [22] propose a general transformation for creating non-blocking data structure algorithms from lock-based data structure algorithms. Instead of setting a lock, a process posts a pointer to its own program. If a process finds that it is blocked because some other process has set a 'lock', the blocked process executes the blocking process' program until the 'lock' is removed.

In the context of real-time synchronization, non-blocking shared objects are desirable because a high priority task will not be blocked by a low priority task. In a uniprocessor system, only one process at a time will access the shared data structures. We can take advantage of the serial but interruptible access to simplify the specification of the existing non-blocking techniques, and to improve on their efficiency. We present methods for implementing interruptible critical sections based on each of the above three non-blocking techniques.

The three methods that we propose are *decisive instruction serializable* [20]. A concurrent execution sequence of a set of operations has an *equivalent serial execution* if the execution is equivalent (with respect to return values and final value) to a serial execution of the operations. A concurrent execution sequence is decisive instruction serializable if every operation has a *decisive instruction*, and operations in an equivalent serial execution are ordered by the points at which the decisive instructions are executed in the concurrent execution.

The decisive instruction of an ICS should naturally be the last instruction in the restartable region. The execution of the decisive instruction 'commits' the operation execution. If an operation requires several global writes, then they can be performed in a *cleanup* phase that preceeds or follows the decisive instruction, as long as all operations agree on the set of cleanup actions. The typical execution sequence of an ICS will be for an operation to cleanup after any previous operation executions, to calculate the decisive instruction, then to perform the decisive instruction as the last instruction in the restartable region.

3.1 Prakash-style Interruptible Critical Section

In [17], Prakash, Lee and Johnson present a simple shared non-blocking linked-list queue that is Their queue is *decisive instruction serializable* Their technique uses the *compare-and-swap* instruction to atomically commit

changes to the data structure.[1]

The `enqueue` operation requires two writes to the global data, as a newly inserted record must be linked to the end of the list, and the tail pointer set to the new record. Linking the new record to the end of the list is the decisive instruction for the operation. Moving the tail pointer is a cleanup action that can be performed by an operation different from the one that performed the decisive instruction. If there is a single record in the queue, then the enqueue and dequeue operations conflict, so a special protocol is required.

The non-blocking queue algorithm is designed so that every update takes the queue from one well-defined state to another. The current state of the queue is defined by the values of the head pointer, the tail pointer, and the value of the next pointer of the tail record. If an operation finds that it is blocked by the half-completed execution of a different operation, the blocked operation completes the execution of the half-finished operation. The actions that must be taken to complete the defined by the current state of the queue.

We can use a technique similar to that developed in [17] to write an ICS in which the operations might need to perform several global writes. A set of *object states* are defined, along with a set of transitions on the states. A state s is *valid* for an operation if the decisive instruction for the operation can be performed on the object when the object is in state s. If s is not valid for operation o, then o must perform *clean-up* actions to put the object into a valid state. For the object to be non-blocking, every state must have a transition for every operation, and all transitions form invalid states must lead towards valid states. We call an ICS written with this technique a *Prakash-style* ICS.

The general form of a Prakash-style ICS looks like the following code. In the first restartable sequence, the operation cleans up after any previous operations by moving the queue towards a valid state. The operation then calculates its decisive instruction. The last instruction in the restartable sequence is the decisive instruction, which establishes that the operation is performed. In the second restartable sequence, the operation cleans up the data structure and restores it to a valid state. The cleanup is optional, since all operations ensure that they work on a valid data structure before performing their decisive instruction.

```
Critical_section(...) {
   restartable{
         initialize local variables
         loop
             perform clean-up actions
             calculate decisive action
         until done
         *lhs=rhs     // Perform decisive instruction
```

[1] A compare-and-swap instruction writes a new value to a memory location only if the current value of the memory location is equal to a specified value

```
    }
    restartable{     // optional
          perform clean-up actions
    }
}
```

As an example, we can modify the Prakash et al. non-blocking queue so that it executes in a restartable atomic sequence. An operation calculates the current queue state by reading the value of the head and tail pointer, and possibly the value of the `next` pointer of the record pointed to by `tail` If the queue is in an invalid state, the operation moves the queue to a valid state (the queue states are listed in figure 1). Finally, the operation performs its decisive instruction. The code listed below specifies the `enqueue` and `dequeue` operations:

```
struct queue_elem{
      data item;
      struct stack_elem *next;
   } *head,*tail;

NQ(elem)
queue_elem *elem
restartable{
      if(tail==NULL or head==NULL){
         if(head==NULL and tail==NULL)
            lhs=&tail
         elseif(head==NULL and tail!=NULL){
            head=tail
            lhs=&(tail→next)
         }
         elseif(head→next==NULL)
            lhs=&(tail→next)
         else {
            tail=head→next
            lhs=&(tail→next)
         }
      }
      else {
         if(tail→next!=NULL)
            tail=tail→next
         lhs=&(tail→next)
      }
      *lhs=elem;
   }
}


stack_elem *DQ() {
struct stack_elem *temp
restartable{
      if(tail==NULL or head==NULL){
```

```
        if(head==NULL and tail==NULL){
           return_value=NULL
           lhs=NULL
        }
        elseif(head==NULL and tail!=NULL){
           return_value=head
           head=tail
           tail=NULL
           lhs=&head
        }
        elseif(head→next==NULL){
           return_value=head
           lhs=&head
        }
        else {
           return_value=&head
           tail=head→next
           lhs=&head
        }
     }
     else {
        if(head→next==NULL) {
           tail=NULL
           return_value=head
           lhs=&head
           }
        else {
           tail=tail→next
           lhs=&(tail→next)
     }
     if( lhs != NULL)
        *lhs=head→next;
  }
  return(return_value)
}
```

Each queue operation cleans up for the previous queue operation before committing its action. Although a high-priority task might clean up after a low priority task, the high priority task does not clean up after its own update. The two execution costs balance. Since the states are well defined, and the cleanup phase takes the queue from one well defined state to another, we can see that the queue is correct.

The code for the ICS queue differs from the code for the non-blocking queue [17] in several details. First, the non-blocking queue must take an *atomic snapshot* of the variables that define the queue state (by using a special protocol). The ICS queue needs only to read the variables since an ICS operation executes serially and is guaranteed of taking an atomic snapshot. Second, the non-blocking queue performs its writes using the *compare-and-swap* instruction to avoid interference with competing operations. The ICS operations perform

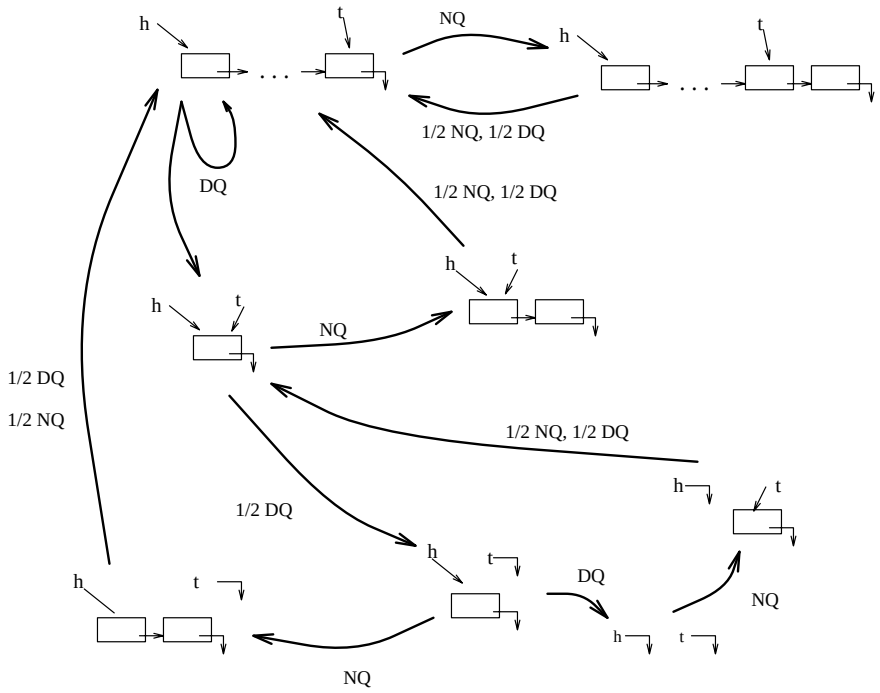

Figure 1: States of a restartable queue

simple writes. Third, the non-blocking queue stores the pointers in double words, and attaches version numbers to avoid problems due to the re-use of records. The ICS queue uses simple pointers. Fourth, the non-blocking queue uses one bit of the version number as a *deleted* bit, to distinguish whether or not a record is in the queue when there is a single record in the queue. Since the ICS queue uses simple pointers, we need to use a more complex protocol to distinguish the states. We note that we could have created an additional state variable to store this information.

3.2 Turek-style Critical Section

A simple data structure such as a queue has only a few states, and is easy to specify. A more complex structure, such as a linked list or a binary search tree, is much harder to describe. For these data structures a more powerful technique, one which explicitly specifies the cleanup actions, is needed.

In [22], Turek et al. propose a method for transforming locking data structures into non-blocking data structures. The key to the transformation is to post a *continuation* instead of a lock. The continuation contains the modifications that the process intends to perform. If a process attempts to post a continuation but is blocked (because a continuation is already posted), the 'blocked' process performs the actions listed in the continuation, removes the continuation, then re-attempts to post its own continuation. As a result, a blocked process can unblock itself.

Although Turek's approach simplifies the process of writing a critical section, a direct translation of Turek's algorithm can require a high priority process to perform the work of many low priority processes that have posted but not yet performed their actions. An easy modification of Turek's approach results in an simple algorithm which guarantees that a high priority process does the work for at most one low priority process. Every shared concurrent object has a single *commit record*, and a flag indicating whether the commit record is valid or invalid. When a process starts executing a critical section, it check to see if a previous operation left an unexecuted commit record (the flag is `valid`). If so, the process executes the writes indicated by the commit record, then sets the flag to `invalid`. The process then performs its operation, recording all intended writes in the commit record. For the decisive instruction, the process sets the flag to `valid`. A typical critical section has the following form:

```
struct commit_record_element{
         word *lhs,rhs} commit_record[MAX]
boolean valid

critical_section()
   restartable{
         if(valid)
            instruction=0
            while(instruction<MAX and commit_record[instruction].lhs != NULL)
               *(commit_record[instruction].lhs)=commit_record[instruction].rhs
            valid=FALSE
         calculate modifications
         load modifications into  commit_record
         valid=true
   }
```

For example, the following code can implement a double linked list:

```
struct list_elem{
      data item;
      struct list_elem *forward,*backward;
   } *head;


struct commit_record_element{
         word *lhs,rhs} commit_record[2]
boolean valid

insert(elem)
list_elem *elem
   list_elem *prev,*next
   restartable{
         if(valid)
            instruction=0
            while(instruction<2 and commit_record[instruction].lhs != NULL)
```

```
            *(commit_record[instruction].lhs)=commit_record[instruction].rhs
         valid=FALSE

      prev=NULL;
      next=head
      while(not_found_position(next))
         prev=next
         next=next→ forward
            // Found the insertion point
      elem→forward=next
      elem→backward=prev
      if(prev==NULL)
         commit_record[0].lhs=&head
      else
         commit_record[0].lhs=&(prev→forward)
      commit_record[0].rhs=elem
      if(next != NULL)
         commit_record[1].lhs=&(next→backward)
         commit_record[1].rhs=elem
      else
         commit_record[1].lhs=NULL
      valid=TRUE
   }
delete(elem)      // elem is the element to be removed
list_elem *elem
   list_elem *prev,*next
   restartable{
      if(valid)
         instruction=0
         while(instruction<2 and commit_record[instruction].lhs != NULL)
            *(commit_record[instruction].lhs)=commit_record[instruction].rhs
         valid=FALSE

      if(elem→backward==NULL)
         commit_record[0].lhs=&head
      else
         commit_record[0].lhs=&((elem→backward)→forward)
      commit_record[0].rhs=elem→forward
      if(elem→forward != NULL)
         commit_record[1].lhs=(elem→forward)→backward
         commit_record[1].rhs=elem→backward
      else
         commit_record[1].lhs=NULL
      valid=TRUE
   }
```

THe transformation from a blocking-based critical section to a Turek-style ICS is straightforward. The cleanup phase is inserted in the beginning of the critical section. Whenever a write is performed into global data in the blocking-based critical section, the write is recorded in the commit record in the ICS. The last statement of the ICS is to set `valid` to `TRUE`. If operations perform few writes, then a high priority task

performs at most a few instructions on behalf of a low priority task. Further, the costs balance becuase the high priority task leaves the commit record for a different task to execute..

4 Herlihy-style Critical Section

If the critical section requires a small modification (or can be broken into several sections, each requiring only a small modification), then the Turek-style approach allows a low priority operation to block a high priority operation for only a short period. If an operation performs a substantial modification and the number of modifications that an operation commits might vary widely, then a high priority operation might spend a substantial amount of time performing a low priority operation's updates to the data structure. In [11], Herlihy proposes a 'shadow-page' method for implementing a non-blocking concurrent data structure. An operation calculates its modifications to the data structure in set of privately allocated (shadow) records, then links its records into the data structure in its decisive instruction. The process is illustrated in figure 2. The blocks in the data structure marked 'g' are replaced by the shadow blocks. An operation performs its decisive instruction by swapping the anchor pointer from the current root to the shadow root. The blocks that are removed from the data structure are garbage collected by the successful operation and are (eventually) made available to other operations. We note that the decisive instruction always must be to swap the anchor, in order to ensure serializability in a parallel system.

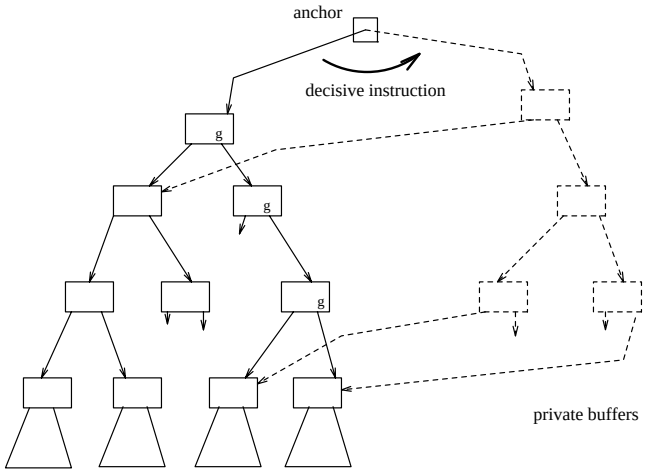

Figure 2: Herlihy's non-blocking data structures

The most complicated part of Herlihy's protocol is managing the garbage-collected records. The protocols are complex, and require $O(P^2)$ space, where P is the number of processes that access the shared object.

We can take advantage of the serial access to the data structure in the ICS to simplify the implementation and reduce the space overhead.

The process of implementing a Herlihy-style ICS is illustrated in Figure 3. A process obtains the records it needs to prepare its modifications from a global stack of records. The global record stack provides the records for all operations that use records of the size it stores. When a process obtains a record from the global stack, it does not remove the record from the stack. Instead, the modifications are made to records while they are still on the stack. A local variable, `current`, keeps track of the last allocated record from the record stack. Another pair of local variables, `g_head` and `g_tail`, keep track of the records to be removed from the data structure. To commit the modification to the data structure, the operation must remove the records it used from the stack of global records, add the garbage records to the global stack, and adjust a pointer in the data structure. These three modifications can be performed using a Turek-style commit record.

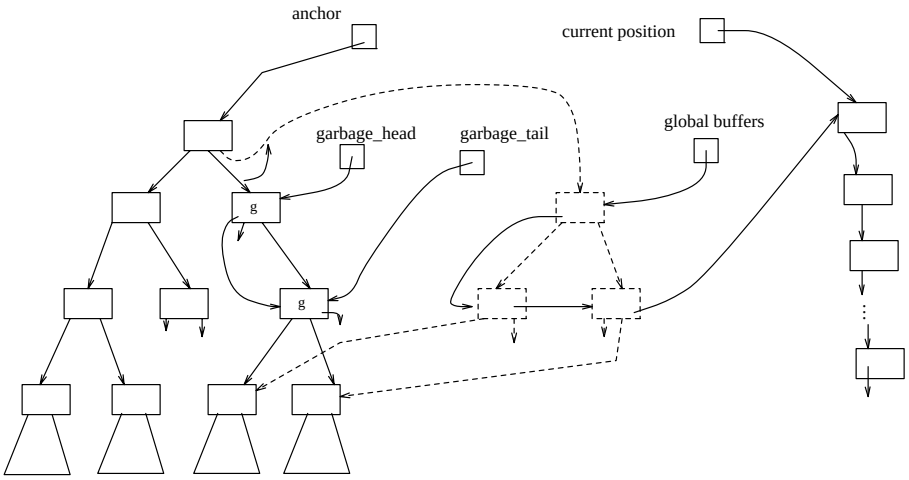

Figure 3: Herlihy-style ICS

Before listing the procedures to implement the Herlihy-style ICS, we note a couple of details. First, every record in the data structure must contain enough additional space to thread a list through it, whether the garbage list or the global record stack. Second, the critical instruction of the operation is to declare that the commit record is valid. As a result, the commit record can contain instructions to change any links in the data structure. As an example, in Figure 3, a link from the root instead of the anchor is modified.

We assume that every record has a field `next` that is used to thread the global record and the garbage lists through the nodes. The procedure for acquiring a new record is:

```
record *getbuf(record **current)
   buffer *temp
      temp=*current
      *current=(*current)→next
```

The procedure to declare that a node is garbage is given by:

```
garbage(record *elem,**g_head,**g_tail)
   if(*g_tail==NULL)
      *g_tail=elem
   elem→next=*g_head
   *g_head=elem
```

A typical critical section is given by:

```
struct commit_record_element{
         word *lhs,rhs} commit_record[3]
boolean valid
Global record *pool

critical_section()
record *current,*g_head,*g_tail
   restartable{
         if(valid)
            instruction=0
            while(instruction<3 and commit_record[instruction].lhs != NULL)
               *(commit_record[instruction].lhs)=commit_record[instruction].rhs
            valid=FALSE
            // Initialize the list pointers
         current=pool
         g_head=g_tail=NULL

         Compute the modifications to the data structure
         using the getbuf and garbage procedures

            // Prepare the commit record

         commit_record[0].lhs=&(g_tail→next)
         commit_record[0].rhs=current
         commit_record[1].lhs=&pool
         commit_record[1].rhs=g_head
         commit_record[2].lhs=critical_link
         commit_record[2].rhs=critical_link_value

         valid=TRUE        // commit your update
   }
```

The Herlihy-style ICS requires that a high priority operation perform at most three writes on the behalf of a low priority operation. Since a high-priority operation does not perform its own clean-up, the costs balance. The space requirements for a Herlihy-style ICS are independent of the number of competing processes, as the global pool must be initialized with enough records to allow the data structure to reach its maximum size,

plus the number of records in the largest modification. Furthermore, the global pool can be shared among several data structures (in which case they must share a commit record). The linked list that is threaded through the data structure imposes an $O(1)$ penalty on every node in the data structure.

5 Correctness

In this version of the paper, we present heuristic arguments for the correctness of the protocols (we will make formal proofs in the journal version). However, we note that we can show the correctness of the ICS implementations 'by inspection'. The fact that operations execute serially greatly simplifies the correctness arguments.

In a Prakash-style implementation, an operation correctly determines the current state of the object because it serially reads variables that define the state. The operation can then perform an arbitrary number of writes to take the object from an invalid state to a valid state, as long as each write takes the queue from one state to another well-defined state. Since the instruction that commits the operation is the last instruction in the restartable region, the operation performs its intended update exactly once. The process re-executes the restartable region until the final write is performed, and the restartable region is not re-executed after the final write is performed.

In a Turek-style implementation, the object is correctly modified by a posted update, because every operation that executes a posted commit record performs the same writes. A half-written commit record will not be executed, because the valid bit must be `invalid` before an operation starts writing into the commit record buffer, and the valid bit is not set to `valid` until the commit record is completed. An operation is performed exactly once because setting the valid bit commits the operation, and it is the last instruction in the restartable region. The Herlihy-style implementation is an optimization of the Turek-style implementation.

6 Analysis

If a real-time system is built using interruptible critical sections, then no task is blocked by a lower priority task. However, the execution times of the critical sections may increase due to pre-emptions by higher-priority tasks. In this section, we estimate the increase in task execution times under the assumption of both periodic and aperiodic task arrivals.

We define:

E_i**:** Time to execute the i^{th} priority task, not including the time spent executing the critical sections.

Z_i^j: Time to execute the j^{th} critical section of the i^{th} priority task.

C_i: Total time to execute the i^{th} priority task.

A real-time system with periodic tasks is often scheduled using the Rate Monotonic algorithm. If a task is pre-empted by the arrival of a higher priority task during the execution of a critical section, the critical section section is re-executed. If we assume that whenever a task is scheduled to use the CPU, it executes for at least the time of its longest critical section, then in the worst case, each of a task's critical sections executes twice. Therefore,

$$C_i \leq E_i + 2\sum_{j=1}^{n_i} Z_i^j$$

To analyze an aperiodic real-time system that uses interruptible critical sections, let us assume that priorities are fixed. In addition, let us assume that tasks with priorities 1 through $i-1$ arrive at rate λ_i, and the arrival process is Poisson. The i^{th} priority task is pre-empted during its j^{th} critical section with probability

$$\Pr[\text{pre}-\text{emption during the } j^{\text{th}} \text{ critical section}] = e^{-\lambda_i Z_i^j}$$

The probability that the j^{th} critical section is executed k times is

$$\Pr[k \text{ executions}] = (1 - e^{-\lambda_i Z_i^j})^k e^{-\lambda_i Z_i^j}$$

Let $\mathcal{Z}_i^j$ be the random variable representing the execution time (including restarts) of the j^{th} critical section of the i^{th} priority task. The probability that the i^{th} priority task execution time is $\leq X$ seconds is a convolution of the distributions of the $\mathcal{Z}_i^j$.

$$\begin{aligned}\Pr[C_i < X] &= \Pr[E_i + \mathcal{Z}_i^1 + \cdots + \mathcal{Z}_i^{n_i} < X] \\ &= \Pr[\mathcal{Z}_i^1 + \cdots + \mathcal{Z}_i^{n_i} < X - E_i]\end{aligned}$$

7 Interruptible Locks

We observe that we an make much more optimistic predictions of critical section execution times if a task in a critical section restarts only when an actual conflict occurs. As an example, consider a Herlihy-style ICS implementation of a buffer shared by readers and writers. The buffer is referenced by a pointer (`buf_ptr`). When a writer completes its critical section, it sets the buffer pointer to the buffer into which it wrote (`newbuf`). A reader reads the buffer pointer into `mybyffer`, then copies its contents into local memory. Each buffer contains a version number field. When a writer obtains a record from the global queue, it increments

the buffer's version number field before writing into the buffer. A reader stores a local copy of the buffer's version number before copying the buffer contents into local storage. After copying the buffer, the reader compares the buffer version number to the stored version number. If the numbers are different, the reader starts again.

The code for the writer is specified by the components to be inserted into the Herlihy-style ICS template:

```
newbuf := getbuf(current)
newbuf->version := newbuf->version + 1
write new data into  newbuf
garbage(buf_ptr,&g_head,&g_tail)
```

The code for the reader is complicated by the need to take a consistent snapshot of the buffer pointer and the buffer version number. The reader protocol ensures that the buffer version number is associated with the data the reader desires.

```
repeat {
   repeat {
      mybuffer := current_buffer
      myversion=current_buffer− >version
   until( mybuffer == current_buffer)
   read the contents the buffer pointed to by mybuffer into local storage
until( myversion == mybuffer->version)
```

Since the writer executes in a restartable atomic sequence, any context switch will force the writer to re-execute. The reader does not execute in a restartable atomic sequence, so the reader will restart only if a writer pre-empts it. This information might let us calculate much tighter bounds on critical section execution times of the readers.

We used restartable atomic sequences as the basis for the interruptible critical section because they have been shown to be fast and easily implemented. However, a task that is executing in a critical section needs to re-execute only if it is pre-empted by a task that performs a conflicting operation. Bershad et al.'s proposal for restartable atomic sequences [3] assumed that the protected region would be very short, and did not plan implementations for real-time systems. To better support the implementation of interruptible critical sections, we propose the idea of *pre-emptable locks*. A pre-emptable lock protects a critical section. A process that is executing in the protected area restarts if a different process commits a conflicting operation.

A possible implementation of an interruptible lock can consist of a version number and a protected region of code:

```
version_number v

pre-emptable_lock(v)
```

```
        :
        < critical section >
        :
end_lock(v,lhs,rhs)
```

When a process enters a region protected with a pre-emptable lock, it records the lock's current version number in an memory area that is shared with the kernel. The `end_lock` code increments the lock's version number, then performs the final write that commits the update. The protected region of code is the critical section code, and the code that performs the `pre-emptable_lock` and `end_lock` functions. When the kernel performs a context switch, it tests whether the process' recorded lock version number matches the current lock version number. If the version numbers are different, the process restarts from the beginning of the critical region. Otherwise, the process continues from its current position.

8 Conclusion

We have presented methods for implementing interruptible critical sections (ICS). Interruptible critical sections use optimistic concurrency control instead of pessimistic concurrency control. If a process that is executing an ICS is interrupted, it restarts its execution from the beginning of the critical section. In a real-time system, interruptible critical sections prevent priority inversion. In addition, the ICS mechanism is independent of the scheduling algorithm. We show how several recent ideas in non-blocking and uniprocessor synchronization can be synthesized to provide low-overhead interruptible critical sections.

When an ICS is used, a low-priority task never blocks a high priority task, but a high-priority task can cause the re-execution of a low-priority task. We present several analyses which calculate the increase in critical section execution time due to pre-emptions by higher priority tasks.

Future Work

For our future work, we will investigate methods for implementing 'interruptible locks'. Interruptible locks will restart a task in a critical section only if another task performs a conflicting operation. If interruptible locks are used instead of restartable atomic sequences, we can obtain much tighter bounds on critical section execution times. In addition, a lock-based mechanism is more portable to a parallel system than is a context-switch based mechanism.

In this paper, we have provided a rough guide to estimating schedulability when using interruptible critical sections. We also plan to more carefully analyze the effects of using an ICS on task schedulability in a variety of task and processing models.

References

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. on Computer Systems*, 10(1):53–79, 1992.

[2] T.P. Baker. A stack-based resource allocation policy for realtime processes. In *Real Time Systems Symposium*, pages 191–200, 1990.

[3] B.N. Bershad, D.D. Redell, and J.R. Ellis. Fast mutual exclusion for uniprocessors. In *5th Intl. Conference on ASPLOS*, pages 223–232, 1992.

[4] H. Tokuda C.D. Locke and E.D. Jensen. A time-driven scheduling model for real-time operating systems. Technical report, Carnegie Mellon University, 1991.

[5] M.I. Chen and K.J. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *Real-Time Systems Journal*, 2(4):325–346, 1990.

[6] M.I. Chen and K.J. Lin. A priority ceiling protocol for multiple-instance resources. In *Real Time Systems Symposium*, pages 140–149, 1990.

[7] C.L.Liu and W.J. Leyland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–63, 1973.

[8] S. Davari and S.K. Dhall. An on-line algorithm ofr real-time task allocation. In *IEEE Real-Time Systems Symposium*, 1986.

[9] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proc. of the IFIP Congress*, 1974.

[10] S.R. Faulk and D.L. Parnas. On synchronization in hard real-time systems. *Communications of the ACM*, 31(3):274–287, 1988.

[11] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.

[12] K. Jeffay. Analysis of a synchronization and scheduling discipline for real-time tasks with pre-emption constraints. In *Real Time Systems Symposium*, pages 295–305, 1989.

[13] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison Wesley, 1989.

[14] C.W. Mercer and H. Tokuda. Preemptibility in real-time operating systems. In *Real Time Systems Symposuim*, pages 78–87, 1992.

[15] A.K. Mok and M.L. Dertouzos. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Trans. on Computers*, 15(12):1497–1506, 1989.

[16] E. Moss and W.H. Kohler. Concurrency features for the trellis/owl language. In *European Conference on Object-Oriented Programming*, pages 171–180, 1987. Appears as Springer-Verlag Computer Science Lecture Note number 276.

[17] S. Prakash, Y.H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *Proc. Int'l Conf. on Parallel Processing*, pages II68–II75, 1991.

[18] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real Time Systems Symposium*, 1988.

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

[20] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[21] S.K. Tripathi and V. Nirkhe. Pre-scheduling for synchronization in hard real-time systems. In *Operating Systems of the '90s and Beyond, Int'l Workshop*, pages 102–108, 1991. Appears as Springer-Verlag Computer Science Lecture Note number 563.

[22] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *ACM Symp. on Principles of Database Systems*, pages 212–222, 1992.