550

Read-Copy Update Mutual-Exclusion in Linux

Table of Contents

2/21

Dynix m,c,c

AVAILABILITY

A basic implementation of read-copy update primitives are available for 2.4.x kernels as patch here .

OVERVIEW

Straightforward, well-understood mutual-exclusion mechanisms have served quite well for over a decade. However, times are changing, and computer architectures are changing with them. These changes have caused tried-and-true locking primitives to become increasingly expensive when compared to the cost of ordinary instructions, and provide motivation for development and use of alternative locking mechanisms.

As usual, there is no free lunch. Higher-performance locking mechanisms tend to be more specialized than the more familar general-purpose mechanisms. One useful specialization stems from the observation that many data structures are accessed much more frequently than they are modified. These data structures could benefit from locking mechanisms that reduce the overhead for accessing the structure, while possibly increasing the cost of modifying the structure.

One such locking mechanism is the reader-writer spinlock. This mechanism gives readers better cache locality (and writers worse cache locality), thereby speeding up readers (and slowing down writers). This mechanism is described in detail in a separate document.

Another specialized locking mechanism is the read-copy ``lock" (actually an update discipline). This mechanism reduces the readers' locking overheads to zero. Since the readers are doing absolutely nothing to protect their accesses, the writers must take careful steps to ensure that their updates will allow the readers to safely access the data at any time. The read-copy update discipline mechanism provides primitives that allow writers to easily take such careful steps.

The overhead of these primitives is relatively low, but is considerably greater than than of a simple spinlock. Therefore, like the reader-writer spinlock, this mechanism improves performance only when the protected data structure is read

much more frequently than written. In addition, the readers must be able to tolerate using stale data. Although this restriction sounds severe, it is satisfied quite often. One example is a routing table in a network protocol, which is updated at most every minute or so, but may be accessed several thousand times per second, and which can tolerate stale routing information. Even in the unforgiving realm of numerical analysis, there are algorithms that can tolerate stale data -- so-called ``chaotic relaxation" is one example. Other example applications of this mechanism are listed in the APPLICATIONS OF THIS MECHANISM section.

The remainder of this document describes the motivation for the read-copy update mechanism (e.g., why is it so important to eliminate lock operations), a description of the linux implementation, benefits and drawbacks, application of this mechanism, some example usages of the mechanism, followed by a detailed description of the interface that are available in the patch for 2.4.x kernel. The examples and interface are specialized for the linux kernel, but the mechanism can be adapted to a variety of other environments, including user processes and other kernels.

MOTIVATION

CPU speeds have been increasing by a factor of from 1.5x to 2x per year for more than a decade. However, interconnect speeds (printed circuit boards and busses) have only been increasing by roughly 15 percent per year over the same period.

To illustrate this effect, consider the number of simple instructions that can be executed in the time required to acquire and release a simple in-kernel spinlock for Sequent's Symmetry Model-B and the NUMA-Q "Scorpion" system containing Xeon processors. The Model B shipped in mid-1988, gets about 5 MIPS, and requires about 10 microseconds to acquire and release a spinlock. The Scorpion shipped in late 1988, gets about 540 MIPS, and requires about 4 microseconds to acquire and release a spinlock.

In the 10 years between the Model B and the Scorpion, the CPU speed has increased more than 100-fold (1.6x increase per year), while the lock speed has increased by only 2.5x (1.1x increase per year). The number of instructions that can be executed in the time required to acquire and release a lock has risen from 50 for the model B to over 2,000 for the Scorpion. If a lock is acquired every 200 instructions, then the locking overhead has increased from 20% to over 90% in a 3.5 year period. This vividly illustrates why reducing the number of lock operations can be just as important as reducing lock contention.

This increase in overhead provides the motivation for constructing mutual-exclusion schemes that do not rely so heavily on communication over the relatively slow interconnects between the relatively fast CPUs.

Note that the MIPS ratings given above assume a low miss rate from the on-chip cache. An analysis of the effect of cache misses would uncover a similar increasing penalty for data movement between memory and caches. This is a subject for another document. Note however that low-end machines that trade low latency for limited scaling have lower miss penalties than do higher-end machines such as ours. Larger miss penalties result in locks having higher overhead on high-end machines than on machines with the low latencies made possible by limited scaling.

The read-copy update mechanism allows locking operations to be eliminated on read accesses to the data it protects.

DESCRIPTION

This section describes the read-copy mechanism as implemented on PTX4.0. The principles underlying the mechanism can be applied to other environments (such as other kernels, to real-time systems, and even to application programs). These underlying principles are documented in a PDCS'98 paper. The rest of this section gives an overview of these

principles. This description assumes a non-preemptive kernel, however, a preemptive kernel may be handled easily.

The basis of the read-copy mechanism is the use of the execution history of each CPU to demonstrate that a given destructive operation can be safely performed at a particular time. The mechanism is currently used in three general ways: (1) as an update discipline that allows readers to access a changing data structure without performing any synchronization operations, (2) to guarantee existence of data structures referenced by global pointers, thereby avoidin complex race conditions that would otherwise occur when freeing up these data structures, (3) to reduce deadlock-avoidance complexity, and (4) as a barrier-like synchronization mechanism that allows one process to exclude other processes without requiring these other processes to execute any expensive locking primitives or atomic instructions.

The key point is that all kernel data structures are manipulated only by the kernel context of a user process or by an interrupt routine. Neither the idle loop nor a user process running in user mode may manipulate kernel data structures. The idle loop, user-mode execution, and (by special dispensation) a context switch are therefore ``quiescent states": a CPU in a quiescent state cannot interfere with manipulation of in-kernel data or be interfered with by manipulation of in-kernel data. Of course, a given CPU could switch out of a quiescent state at any time, but such a CPU would only be able to access or modify shared data that was globally accessible at or after that time. Therefore, if a given process removes all globally-accessible references to a particular data structure, then blocks until all CPUs pass through a quiescent state (refering to a summary of CPU execution history to determine when this occurs), it can manipulate that data structure secure in the knowledge that no other CPU can interfere. Likewise, if a given process makes a change to a global variable, then blocks until all CPUs have passed through a quiescent state, it can be sure that all the other CPUs are now aware of the change, even if that variable is not protected by any sort of mutex.

The core of the read-copy mechanism is a primitive named `call_rcu()` that allows a callback to be scheduled once all CPUs have passed through a quiescent state.

An example use of this mechanism is the deletion of a data structure that is still being used by processes and interrupt routines running on other CPUs. To delete the data structure, simply do the following:

- Copy a pointer to it.
- Store `NULL` into the global pointer so that any subsequent processes or interrupts will be unable to access the data structure.
- Use `call_rcu()` to schedule a callback that frees up the data structure.

This procedure will allow all kernel threads of execution that might still be using the data structure to terminate before actually deleting the structure. Note that spinlocks or reference counters could be used to get the same effect, but that these techniques would impose overhead on everything accessing the data structure.

The `call_rcu()` primitive takes a pointer to a `struct rcu_head` control block, a callback function to be invoked at the end of quiescent cycle and the argument to be passed to the callback function.

The function may reuse the `rcu_head` by rescheduling itself, otherwise it may free it up.

`wmb()` primitive is used to maintain memory consistency on CPUs that can reorder memory reads and writes (note however that the existing spinlock primitives act as implicit `wmb()`s). This primitive must be used between the act of filling in the fields internal to a given structure and the act of creating a globally-accessible pointer to that structure. Otherwise, an aggressive super-scalar CPU might reorder the memory write to the global pointer before some of the writes that initialized the structure itself. This reordering could allow other CPUs to access the structure before it was fully initialized, which could significantly reduce the longevity of the kernel.

The preceding primitives form the interface to the core read-copy mechanism, and are described in detail later in this document.

So where did the name ``read-copy" come from? It turns out that this is how the mechanism is used to modify (rather than simply delete) a data structure while allowing readers full concurrent access to that data structure. The procedure is as follows:

1. Allocate new memory for the modified data structure.
2. Copy the contents of the old data structure to the new.
3. Modify the new data structure as needed.
4. Update all pointers that currently point to the old copy to instead point to the new copy.
5. Do a deferred free of the old copy.

In other words, readers can continue reading while updaters make new copies, hence ``read-copy". Use of the mechanism in this manner is sometimes called a ``read-copy lock"; this nomenclature was stolen from the ``reader-writer lock". This analogy with spinlock primitives can be useful, but it is better to think of the read-copy mechanism as enabling the read-copy update procedure shown above than to risk confusing the mechanism with explicit locking.

BENEFITS OF READ-COPY MECHANISM

1. Requires execution of fewer explicit mutex operations are required to protect accesses to shared data structures. In many cases *no* explicit lock operations are required. This results in less contention and lower CPU overhead. Of course, *modification* of shared data structures still requires locking operations. However, all the well-known techniques (such as hashing) may be used to reduce the lock contention incurred when modifying shared data.
2. Requires execution of fewer special instructions that enforce sequential consistency (e.g., instructions taking the ``lock" prefix on the Intel 80x86 instruction-set architecture). These instructions result in CPU pipeline flushes, resulting in degraded performance, especially on the newer super-scalar microprocessors containing multiple instruction pipelines.
3. In code not subject to blocking preemption, requires fewer interrupt-level manipulation operations when accessing shared data structures. These operations often require interaction with relatively slow external interrupt-distribution hardware, and are frequently required even on uniprocessors (in which case the read-copy update mechanism can be said to have *negative* overhead with respect to traditional uniprocessor algorithms). In PTX, interrupt-level manipulation operations are usually combined with the locking operations and are still usually required when *modifying* shared data structures.

   An example of this occurs when a data structure is modified at interrupt level. If the interrupt code can use the read-copy mechanism, then any process-level code that reads the data structure no longer needs to exclude the update code in the interrupt routine. Therefore, the process-level code no longer needs to block interrupts for the duration of its access.

   In environments allowing preemption, interrupt-level manipulations or operations that disable preemption may still be required.

4. This mechanism is not as susceptible to deadlock as are explicit lock operations. The reduced susceptibility to deadlock results in reduction in or elimination of code that would otherwise be required to detect and repair or tc avoid deadlocks, which in turn reduces software development and maintenance costs.

RESTRICTIONS FOR USE OF READ-COPY MECHANISM

As noted in the overview, high-performance locking mechanism like RCU are specialized, and can therefore be used only in certain situations. This section lists these restrictions along with metrics that can help to determine when it is beneficial to use the RCU mechanism.

1. Writers cannot exclude readers. This can be an advantage in cases where there may be bursts of very heavy write activity, causing conventional locking primitives to slow down or or entirely lock out read activity. However, readers must be excluded in those cases where writers cannot make their modifications in such a way that each intermediate state appears consistent to a reader. This read-copy mechanism is unable to provide this form of exclusion.

2. Readers cannot exclude writers. Readers may therefore see stale data. Therefore, algorithms employing the RCU mechanism must tolerate stale data, or must explicitly flag stale data (see Pugh's skip-list papers for ways of accomplishing this).

3. There is a significant latency imposed by the primitives making up the write side of the read-copy mechanism, even in absence of contending readers. This latency averages about 25 milliseconds on lightly-loaded systems. In other words, there will be about a 25-millisecond delay between the time that `call_rcu()` is invoked until the time that the RCU subsystem invokes the corresponding callback.

   This figure assumes a 10-millisecond clock interrupt as well as short in-kernel code paths. Varying the clock frequency will change the average delay. Long in-kernel code paths will increase the average delay. Note that speeding up the clock (thus decreasing the average delay) will in turn decrease the number of requests satisfied by a single quiescent-state detection, which in turn can degrade performance.

4. The write side of the read-copy update mechanism can be stalled by bugs in the kernel that result in long-running loops. For example, if a buggy kernel driver spins for one second, there can be up to a one-second latency imposed on the write-side primitives. Note that this latency does *not* consume CPU time. PTX 4.0.1 and 4.1 (as well as later OS versions) have a version of the RCU mechanism that issues warnings to the console when such stalls occur. In even later versions of the OS, these warnings include a short-form stack trace of the overly long code segment.

5. In cases where it is permissible to exclude readers, the read-to-write ratio must be rather high in order for the read-copy mechanism to have a performance benefit (in cases where it is not permissible to exclude readers, read-copy is the only game in town). The exact breakeven ratio will vary depending on the details of the situation, but, as a rule of thumb, the read-copy update mechanism will almost certainly provide a performance benefit when the read-to-write ratio is N-to-1, where N is the number of CPUs. If there are a large number of read-copy updates per unit time, the overhead of each update is amortized, so that read-copy update provide a performance benefit with read-to-write ratios as low as 1.10.

   The reason for this wide range is that the primitives making up the read-copy mechanism make heavy use of batching techniques that allows overhead to be spread over multiple concurrent requests.

6. The use of the read-copy mechanism for operations that modify existing data structures sometimes requires that the modification process be restructured into read-copy form. This restructuring is necessary in cases where the modification cannot be made without the data structure passing through inconsistent intermediate states.

   The read-copy form requires that a new data structure be allocated and copied from the old one. The new data structure is then modified as required. The old data structure is then replaced with the newly modified one. Finally, the old data structure is deleted when it is safe to do so.

   Such restructuring will likely require the addition of memory allocation, which can fail. These failures must be correctly handled, either by blocking until memory is available, aborting the modification with an error, preallocating the memory, or using some mechanism to retry the modification at a later time.

   In some cases, the complexity added by this restructuring preclude use of the read-copy mechanism.

   Note that modifications that do not leave the data structure in inconsistent intermediate states do not require this sort of restructuring. Examples include "blind" writes to single aligned bytes, words, or longs, or atomic updates of single aligned bytes, words, longs, or (on Pentiums and better) 8-byte quantities.

7. The read-copy mechanism may not be used to protect access to a data structure across a blocking operation. This same restriction applies also applies to spinlocks.

   Therefore, data structures containing semaphores are not candidates for the read-copy mechanism unless the semaphores can be guaranteed to have no processes sleeping on them at the time of deletion. (It is possible to create a mechanism analogous to the read-copy mechanism where CPUs are replaced by processes and where a different set of quiescent states is selected so that this new mechanism can tolerate blocking. Such a mechanism was created as part of ptx4.5, and is used to protect per-process state such as credentials, current directory, and root directory.)

8. A read-copy update requires the updater to locate all globally accessible pointers to the structure being updated. Local temporary variables need *not* be located, as the read-copy callback will delay until all such temporaries are no longer in use.

9. Readers cannot block while accessing a data structure protected by the RCU mechanism. If a reader blocks, it must restart the access from the beginning, as the data structure may well have undergone arbitrary changes in the meantime. In principle, it is possible to recast the RCU mechanism into a per-process form that would tolerate reader blocking, such as was done to protect per-process state in ptx4.5.

Given all of these restrictions, it might seem that the read-copy mechanism is useful only in a few esoteric situations. In fact, these situations are surprisingly common. The following section lists some areas where the read-copy mechanism is useful and lists some conditions that must be satisfied for it to be used.

APPLICATIONS OF THIS MECHANISM

The following are a few specific examples of applications of this mechanism.

1. Routing tables in computer communications protocol implementations. Here, the readers look up routes in the table for the packets leaving the system. The writers apply routing updates to the table. With the advent of fibrechannel, disk I/O must often be routed just like traditional comms traffic. Read-copy update may therefore be used to route disk-I/O traffic through fibrechannel networks.

2. Address-resolution tables in computer communications protocol implementations. Readers look up link-level addresses for the packets leaving the system and writers apply address-resolution updates to the table.

3. Device state tables. Readers access and possibly modify the individual entries, while writers may add and delete entries. If readers can modify the individual entries, then there will need to be some sort of mutex on the individual entries.

4. Tables whose entries can be more efficiently deallocated in batches than one at a time (for example, by spreading locking overhead over more than one entry).

5. Phased state change. A data structure with an intermediate state between each existing state can switch to the intermediate state, then switch to the destination state once there are no more threads that have reason to assume that the data structure was still in the initial state.

   This is used in ptx/CLUSTERS when starting the recovery phase. When a lock-manager instance determines that recovery is necessary, it sets a flag and registers a callback with the read-copy mechanism. Once the callback is invoked, the lock-manager can be sure that all subsequent processing will be aware that a recovery is in progress. The alternative is to protect all code checking or setting the flag with an explicit (and expensive) mutex.

6. Existence guarantees. This is used to more simply handle races between use of a service and teardown of that service. This approach is used in TCP/IP to handle the case where an IP protocol (such as TCP, UDP, or, of more practical interest, ICMP) is shut down while packets are still being received and processed. Each such protocol has a pointer to its private data structures, and these data structures must be freed up as part of the shutdown process. Clearly, it would be disasterous if a packet, received just before shutdown, was still being processed when the relevant data structures were freed. This disaster could be prevented using global locks or reference counts on each and every packet reception, but this would result in cache thrashing.

   Instead, we first `NULL` out the pointer to the private data structures so that any subsequent packets will be properly rejected. Then, the read-copy mechanism is used to post a callback when all current kernel threads of execution -- including those currently processing recently-received packets -- have terminated. At this point, it is safe to delete the private data structures.

   Existence guarantees are also used to ensure that all CPUs are finished using a data structure before deleting it. This case can occur during processing of a Unix `close()` system call. The kernel context of the process doing the close can set a flag indicating that the data structure is no longer available. The mechanism may then be used to wait until all currently-executing kernel threads of control (which may not have seen the new value of the flag) have terminated, at which point it is safe to delete the data structure.

7. Poor-man's garbage collection. A function that returns a pointer to a dynamically-allocated structure could relieve the caller of the responsibility for freeing the structure by doing a deferred deletion before returning. It is not yet clear whether this is a useful tool or a shameless kludge.

8. Simplify locking design when freeing a structure. It is sometimes convenient to hold a structure's lock throughout the entire deletion process. However, one problem with this approach is that if some other CPU attempts to acquire the lock just after the deleting CPU acquires it, the second CPU can end up holding the lock just after the element is returned to the freelist, where it might be reused for some unrelated purpose arbitrarily quickly. Doing a deferred rather than an immediate deletion is one way to resolve this problem -- the element could not be returned to the freelist until the second CPU was finished looking at it. This would allow the deleting CPU to set a flag in the structure indicating that it was no longer valid.

This technique is used in ptx/TCP 4.2.0 and later to avoid deadlocks in STREAMS accept/t_accept processing. I is also used in ptx/TCP 4.2.0-4.4.x to avoid socket_peer deadlocks in STREAMS open processing.

Cases 1, 2, and 3 will have read-side code with the following structure when coded using Dynix/PTX spinlock primitives:

```
spin_lock(&my_lock);
/* look something up. */
spin_unlock(&my_lock);
/* use the item looked up. */
```

This situation offers the most natural opportunity for use of these primitives. The information is used outside of the lock, and therefore may be stale. In contrast, code with the following structure:

```
spin_lock(&my_lock);
/* look something up. */
/* use the item looked up. */
spin_unlock(&my_lock);
```

is probably *not* a candidate for use of these primitives, as this code guarantees that the information is up-to-date while i is being used. As noted in the previous section, the locking design would have to be restructured before read-copy update could be used.

DESIGN RULES FOR `call_rcu()`

Applying the `call_rcu()` function to a data structure that is linked together with many other data structures can be an exercise in frustration if done in an ad-hoc manner. An effective way to manage the complexity is to restrict the pointer-manipulation to a few functions as described below. This approach can also do much to simplify existing code, independent of the use of the read-copy update mechanism.

The first thing to note is that use of call_rcu() assumes that updates are much less frequent than accesses. Therefore, the code that performs the updates usually need not be highly optimized, and that the function-call overhead that will be incurred by grouping the needed pointer manipulations in a few functions is not of concern.

The second thing to note is that a common mistake in the use of `call_rcu()` is to treat a structure slated for deletion as if it were the most up-to-date copy. The best way to guard against this is to maintain a flag in the structure that indicates that it is slated for deletion and to keep a pointer to its replacement, as shown in the following examples.

An ``xxx" structure that supports the read-copy mechanism might therefore be defined as follows:

```
typedef struct xxx xxx_t;

struct xxx {
        xxx_t       *xxx_next;
        xxx_t       *xxx_prev;
        long        xxx_flags;
        xxx_t       *xxx_replacement;
        long        *xxx_commonctr;       /* example field to be */
                                          /* shared among all versions */
                                          /* of an object. */
        /* other fields as needed, e.g... */
        long        xxx_address;          /* used to identify xxx. */
        spinlock_t xxx_lock;  /* used to lock individual */
                                          /* xxx_t. */
        struct rcu_head rch;
};

#define XXX_DEFDEL 0x4000 /* Deletion in progress. */
#define XXX_DEFDUP 0x8000 /* Deletion of duplicate. */
```

```
xxx_t      xxx_head;  /* Global list header. */
spinlock_t xxx_mutex;/* Global update mutex. */
```

Implementations in languages such as C++ or SmallTalk that support a more natural expression of object properties should of course use a more object-oriented approach. In either case, some algorithms may be structured so as to allow the `xxx_next` pointer to be used in place of the `xxx_replacement` pointer.

Allocation of an ``xxx" structure should be performed by a function of the form:

```
xxx_t *
xxx_alloc(xxx_t *oldp, int flag)
{
        xxx_t *newp;
        long *commonctr = NULL;

        if (oldp == NULL) {

                /*
                 * There is no old version, so allocate
                 * all fields that are common to all
                 * versions.
                 */

                commonctr = kmalloc(sizeof(*commonctr), flag);
                if (commonctr == NULL) {
                        return (NULL);
                }
                *commonctr = 0;
        }

        newp = kmalloc(sizeof(xxx_t), flag);
        if (newp == NULL) {
                return (NULL);
        }

        if (oldp != NULL) {
                INSIST(oldp->xxx_flags & XXX_DEFDEL == 0,
                       "xxx_alloc: update of deleted item!");
                /*
                 *+ Internal software error.  Caller attempted
                 *+ to modify a structure already slated for
                 *+ deletion.  Corrective action:  change
                 *+ code to modify the current version.
                 *+ This may involve eliminating a race
                 *+ condition.
                 */

                /* Also, caller must hold xxx_mutex. */

                *newp = *oldp;
                oldp->xxx_flags |= XXX_DEFDEL | XXX_DEFDUP;
                oldp->xxx_replacement = newp;
        } else {

                newp->xxx_commonctr = commonctr;

                /* initialize nonzero fields of newp. */
        }

        return (newp);
}
```

Once a structure has been allocated, the calling code will likely modify some of the fields as needed. If the structure is to replace an existing one, the following function would be used:

```
void
xxx_replace(xxx_t *newp, xxx_t *oldp)
{
        /* Caller must hold xxx_mutex. */

        /* Commit to memory before pointer update. */
```

```
        wmb();

        /*
         * Update all pointers to old version to now
         * point to new version.
         */

        if (newp->xxx_next->xxx_prev == oldp) {
                newp->xxx_next->xxx_prev = newp;
        }
        if (newp->xxx_prev->xxx_next == oldp) {
                newp->xxx_prev->xxx_next = newp;
        }

        call_rcu(&oldp->xxx_rch, xxx_cleanup, oldp);
}
```

Another function named `xxx_insert()` would be needed to insert an entirely new structure into the lists. A fourth function named `xxx_delete()` would be used to remove a structure without replacing it with a new copy. Its form would be similar to that of `xxx_replace()`.

```
void
xxx_insert(xxx_t *newp)
{
        /* Caller must hold xxx_mutex. */

        /* Commit to memory before pointer update. */

        wmb();

        /* Link into all relevant lists.  */

        newp->xxx_prev = &xxx_head;
        newp->xxx_next = xxx_head->xxx_next;
        xxx_head.xxx_next->xxx_prev = newp;
        xxx_head.xxx_next = newp;
}

void
xxx_delete(xxx_t *oldp)
{
        /* Caller must hold xxx_mutex. */

        /* Link out of all relevant lists.  */

        oldp->xxx_prev->xxx_next = oldp->xxx_next;
        oldp->xxx_next->xxx_prev = oldp->xxx_prev;

        /* Mark as deleted. */

        INSIST(oldp->xxx_flags & XXX_DEFDEL == 0,
               "xxx_delete: deletion of deleted item!");
        /*
         *+ Internal software error.  Caller attempted to
         *+ delete a structure already slated for deletion.
         *+ Corrective action:  change code to modify the
         *+ current version.  This may involve eliminating
         *+ a race condition.
         */

        oldp->xxx_flags |= XXX_DEFDEL;

        /*
         * Delete the structure when safe to do so.
         * Also does implicit wmb().
         */
        call_rcu(&oldp->xxx_rch, xxx_cleanup, oldp);
}
```

If the structure has pointers to other structures which themselves may need to be freed, then an `xxx_cleanup()` function may be used to handle this. This example has a field `xxx_commonctr` that is common to all versions of a given structure. Therefore, this field must be freed only when the object is deleted -- not when a version is deleted after being

replaced by a newer version. If the structure did not have any such fields, then the `xxx_cleanup` passed to the call to `call_rcu()` above would be replaced with `NULL` in order to omit any cleanup processing.

```
void
xxx_cleanup(void *ptr)
{
        /*
         * Pointer to the rcu_head is passed. So compute the pointer
         * to the xxx_t structure
         */
        xxx_t *xp = (xxx_t *)ptr;

        if (!(xp->xxx_flags & XXX_DEFDUP)) {

                /*
                 * This is not an old duplicate, so we
                 * must free up all fields common to
                 * all versions.
                 */

                kmem_free(xp->xxx_commonctr,
                          sizeof(*xp->xxx_commonctr));

                /*
                 * Insert code here to free up other
                 * fields common to all versions.
                 */
        }

        /*
         * Insert code here to free up any other structures
         * pointed to by this one that are not referenced
         * by other structures.
         */

        /*
         * Free the xxx_t structure.
         */
        kfree((void *)xp);

}
```

To illustrate, to update the structure pointed to by xp from within a process context:

```
spin_lock(&xxx_mutex);

/*
 * If the initial search for xp was done under the lock, then
 * this "while" loop would not be necessary, see below.
 */

while (xp->xxx_replacement != NULL) {
        xp = xp->xxx_replacement;
}
newp = xxx_alloc(xp, GFP_KERNEL|GFP_ATOMIC);
/* insert code here to modify newp's fields as appropriate. */
xxx_replace(newp, xp);
spin_unlock(&xxx_mutex);
```

An entirely new structure would be added as follows:

```
spin_lock(&xxx_mutex);
newp = xxx_alloc(NULL, GFP_KERNEL|GFP_ATOMIC);
/* insert code here to initialize newp's fields. */
xxx_insert(newp);
spin_unlock(&xxx_mutex);
```

An existing structure would be deleted (without replacement) as follows:

```
spin_lock(&xxx_mutex);

/*
```

```
 * If the initial search for xp was done under the lock, then
 * this "while" loop would not be necessary, see below.
 */

while (xp->xxx_replacement != NULL) {
          xp = xp->xxx_replacement;
}
xxx_delete(xp);
spin_unlock(&xxx_mutex);
```

The `while` loop in both the replacement and deletion cases is not needed if the list traversal that results in the xp pointer is conducted under the lock. For example, if the lookup and deletion code were combined, the following would result:

```
spin_lock(&xxx_mutex);
xp = xxx_lookup(xxx_address);
if (xp == NULL) {
          spin_unlock(&xxx_mutex);

          /*
           * Do whatever is necessary to report failure and
           * return control to the caller.
           */
}
xxx_delete(xp);
spin_unlock(&xxx_mutex);
```

Since the lock is held while looking up the pointer, it is not possible for the pointer to become obsolete.

The code for xxx_lookup would be as follows:

```
xxx_t *
xxx_lookup(long xxx_address)
{
          xxx_t *xp;

          for (xp = xxx_head; xp != NULL; xp = xp->xxx_next) {
                    if (xp->xxx_address == xxx_address) {
                              return (xp);
                    }
          }
          return (NULL);
}
```

Note that the caller is responsible for locking. As noted earlier, updaters would use `xxx_mutex` to lock. Readers would just use `RCU_RDPROTECT()` and `RCU_RDUNPROTECT()` as follows:

```
RCU_RDPROTECT();
xp = xxx_lookup(xxx_address);
if (xp != NULL) {
          /* Do something with xp. */
}
RCU_RDUNPROTECT();
```

ADDITIONAL DESIGN RULES TO PROTECT LIST OF ELEMENTS

The RCU discipline is often used to protect a list of elements, but each element itself may need to be protected by a normal spinlock. In this case, the following code would be used to find the element and lock it:

```
RCU_RDPROTECT();
xp = xxx_lookup(xxx_address);
if (xp != NULL) {
```

```
                spin_lock(&xp->xxx_lock);
                if (xp->xxx_flags & XXX_DEFDEL) {
                        /*
                         * handle case where item is deleted,
                         * possibly by retrying the search.
                         */
                } else {
                        /* Do something with xp. */
                }
                spin_unlock(&xp->xxx_lock);
        }
        RCU_RDUNPROTECT();
```

In particular, the following code would delete an item:

```
        spin_lock(&xxx_mutex);
        xp = xxx_lookup(xxx_address);
        if (xp != NULL) {
                spin_lock(&xp->xxx_lock);
                ASSERT_DEBUG(!(xp->xxx_flags & XXX_DEFDEL),
                                "item deleted without xxx_mutex");
                xxx_delete(xp);
                spin_unlock(&xp->xxx_lock);
        }
        spin_unlock(&xxx_mutex);
```

An new structure would be added as follows:

```
        spin_lock(&xxx_mutex);
        newp = xxx_alloc(NULL, GFP_KERNEL|GFP_ATOMIC);
        /* insert code here to modify newp's fields as appropriate. */
        spin_lock(&xp->xxx_lock);
        xxx_insert(newp);
        spin_unlock(&xp->xxx_lock);
        spin_unlock(&xxx_mutex);
```

Note that it is not necessary to use the read-copy primitives in order to update a single element in place, as the xp->xxx_lock may be used to guard such updates.

DESIGN RULES FOR using `kmem_cache_free()`

Unlike `[kv]free_rcu()`, `kmem_cache_free()` requires one additional pointer - the pointer to the cache to free any memory allocated from it. This causes a bit of a problem for RCU interfaces as they expect only one pointer. However, this can be easily avoided by using a specific *destroyer* function for each cache that requires deferred freeing. If the cache pointer is a global variable, the *destroyer* function can directly use that and invoke `kmem_cache_free()` in the callback. Otherwise, the cache pointer can be embedded in the data structure along with `struct rcu_head` and can be re-computed by the *destroyer* function.

For global cache pointers -

```
        static kmem_cache_t xxx_cache;

        struct xxx_entry {
                struct xxx_entry *next;
                struct xxx_entry *prev;
                int flag;
                int info;
                struct rcu_head rch;
        }

        void xxx_destroy(void *ptr) {
                struct xxx_entry *xp = (struct xxx_entry *)ptr;
                kmem_cache_free(&xxx_cache, (void *)xp);
        }
```

```
void xxx_delete(struct xxx *xp) {
        call_rcu(&xp->rch, xxx_destroy, xp);
}
```

If the cache pointer is not available as a global variable, it can still be embedded in the structure and accessed during destruction -

```
struct xxx_entry {
        struct xxx_entry *next;
        struct xxx_entry *prev;
        int flag;
        int info;
        kmem_cache_t *cachep;
        struct rcu_head rch;
}

void xxx_destroy(void *ptr) {
        struct xxx_entry *xp = (struct xxx_entry *)ptr;
        kmem_cache_free(&xp->cachep, (void *)xp);
}

void xxx_delete(struct xxx *xp) {
        call_rcu(&xp->rch, xxx_destroy, xp);
}
```

EXAMPLE OF ELIMINATING TABLE-LOOKUP MUTEX

It is fairly common to have a list structure protected by a global mutex whose individual elements are protected by a per-element mutex. In most cases, searches of the list are *much* more common than are insertions or deletions.

For example, the following code gives (simplistic) search and delete functions for a circular doubly-linked list of elements:

```
spinlock_t hash_lock;

struct element {
        struct element *next;
        struct element *prev;
        spinlock_t element_lock;
        int address;
        int data;
        int deleted;  /* used later... */
};

struct element head[NHASH];  /* waste a bit of space... */

#define hashit(x) ((x) % NHASH)

/*
 * Find element with specified address, lock it, and return
 * a pointer to it.  Return NULL if no such address.
 * Drop the global lock if rlsgbl is set.
 * If element not found, drop all locks.
 */

struct element *
search(int addr, int rlsgbl)

{
        struct element *p;
        struct element *p0;

        p0 = &head[hashit(addr)];
        spin_lock(&hash_lock);
        p = p0->next;
        while (p != p0) {
                if (p->address == addr) {
```

```
                                spin_lock(&p->element_lock);
                                if (rlsgbl) {
                                        spin_unlock(&hash_lock);
                                }
                                return (p);
                        }
                        p = p->next;
                }
                spin_unlock(&hash_lock);
                return ((struct element *)NULL);
}

/*
 * Delete the specified element from the table.  Element
 * must be locked, global lock must -not- be held.
 */

void
delete(struct element *p)

{
                int addr;

                ASSERT_DEBUG(!p->deleted, "Element already deleted");

                /* Try to get global lock. */

                if (spin_trylock(&hash_lock) == CPLOCKFAIL) {

                        /* Someone else has the global lock.
                         * To avoid deadlock, we must release the
                         * element lock and acquire the locks in
                         * the proper order.  In the meantime, someone
                         * may have deleted the element, so we must
                         * check.  We must do an explicit search, as
                         * we cannot trust our pointer to survive
                         * unless we have one of the locks held.
                         * Don't bother coding inline, as this
                         * should be a rather rare case.  In fact,
                         * if efficiency is a concern, the body
                         * of this if-statement should be coded
                         * out-of-line.
                         */

                        addr = p->address;
                        spin_unlock(&p->element_lock);
                        if ((p = search(addr, 0)) == NULL) {

                                /* Someone else deleted us. */

                                return;
                        }

                        /*
                         * The element is still there and we have
                         * all the locks we need.
                         */
                }

                /*
                 * We now have all needed locks, so
                 * delete the element from the list, release
                 * locks, free up the element, and return.
                 */

                p->next->prev = p->prev;
                p->prev->next = p->next;
                spin_unlock(&p->element_lock);
                spin_unlock(&hash_lock);
                kfree(p);
                return;
}
```

This code has lots of lock round-trips and has some rather ugly deadlock-avoidance code. Compare to the following code which uses `call_rcu`:

```
struct element {
```

```
        struct element *next;
        struct element *prev;
        spinlock_t element_lock;
        int address;
        int data;
        int deleted;  /* used later... */
        struct rcu_head rcu;
};

/*
 * To allocate an element
 */
elem = kmalloc(sizeof(struct element), GFP_KERNEL);


/*
 * Find element with specified address, lock it, and return
 * a pointer to it.  Return NULL if no such address.
 */

struct element *
search(int addr)

{
        struct element *p;
        struct element *p0;

        p0 = &head[hashit(addr)];
        RCU_RDPROTECT();
        p = p0->next;
        while (p != p0) {
                if ((p->address == addr) &&
                    !p->deleted) {
                        spin_lock(&p->element_lock);
                        RCU_RDUNPROTECT();
                        return (p);
                }
                p = p->next;
        }
        RCU_RDUNPROTECT();
        return ((struct element *)NULL);
}

/*
 * Delete the specified element from the table.  Element
 * must be locked.
 */

void
delete(struct element *p)

{
        int addr;

        if (p->deleted == 1) {
                spin_unlock(&p->element_lock);
                return;
        }

        /*
         * Get global lock.  There is no possible deadlock,
         * since the global lock is now always acquired
         * after the per-element lock.
         */

        spin_lock(&hash_lock);

        /*
         * Delete the element from the list, release
         * locks, free up the element, and return.
         */

        p->deleted = 1;
        p->next->prev = p->prev;
        p->prev->next = p->next;
        spin_unlock(&p->element_lock);
        spin_unlock(&hash_lock);
        call_rcu(&p->rcu, kfree, p);
        return;
```

```
}
```

There is one less lock round-trip in `search()`, and the code for `delete()` is much simpler because there is now no possibility of deadlock.

DOCUMENTATION OF INTERFACES

call_rcu

```
void call_rcu(struct rcu_head *head, void (*func)(void *head))
```

Invokes a callback function `func` after all the CPUs have gone through atleast one quiescent state. For performance reasons, good implementation of RCU will not wait for completion of the quiescent cycle. Instead, it will queue it in batches and return. The callback will be invoked later when all the CPUs go through atleast one quiescent state. The callback function may be invoked from bottom-half context, so the user must take that into consideration.

wmb

```
wmb()
```

Force a memory-system synchronization point. This is required for systems (such as the 80486) which do not support strict sequential consistency. On such systems, reads and writes may be executed out of order, which could potentially cause data elements to be linked into data structures before they are completely initialized, even though the source code specifies all initialization operations before the link-in operations. Placing an `wmb()` invocation between the initialization and link-in operations will ensure that the initialization is completed before the link-in starts.

wmb_dd

```
wmb_dd()
```

Force a read-side memory-system synchronization point. This is required for certain styles of read-copy-update usage on CPUs such as the DEC Alpha that do not support a strong memory barrier. Although the easiest way to adapt a read-copy update algorithm to such a CPU uses `rmb()`, there are more efficient methods.

CPUS WITH WEAK MEMORY BARRIERS

Although read-copy update has been designed to handle weak memory consistency models, it does assume that a strong memory-barrier operation is available. Such an operation is not available on certain CPUs including DEC's Alpha. Lack of a strong memory-barrier operation causes the following read-copy update code fragments to fail:

```
/* Insert new after pred.  Caller has initialized *new. */

void
insert(foo_t *pred, foo_t *new)
{
```

```
        spin_lock(&mymutex);
        new->next = pred->next;
        wmb();
        pred->next = new;
        spin_unlock(&mymutex);
}

/* Search. */

foo_t *
search(foo_t *head, int addr)
{
        foo_t *p = head;

        while (p != NULL) {
                if (p->addr == addr) {
                        return (p);
                }
                p = p->next;
        }
        return (NULL);
}
```

The problem is that the MB and WMB instructions do *not* force ordering of invalidation operations. Instead, they simply:

1. Force all invalidations received by the current CPU from other CPUs and DMA devices to be processed.

2. Force all current write buffers from the current CPU to be sent out from the current CPU to the "coherency point before any subsequent writes from the current CPU. MB and WMB do *not* wait for completion of the invalidations corresponding to current write buffers.

This means that the search code in the previous example does not work. For example, some other CPU might see the changed value of `pred->next`, but see the old (garbage) value of `new->next`. Even though the `wmb()` forced the new value of `new->next` out of the CPU before the new value of `pred->next`, it did not force all invalidations of the old value of `new->next` to complete before all invalidations of the old value of `pred->next`. Therefore, if `new->next` took a long time to invalidate, some other CPU might see the garbage value for it after the new value for `pred->next`. This could lead to data corruption, a panic, or both.

The following are some ways to fix this code:

1. Come up with some sort of Alpha PAL procedure that has the desired effect of waiting for all outstanding invalidations to complete. My contacts who know about the Alpha tell me that this is not possible, that the bus protocol does not let the CPU in on this secret.
2. Insert wmb() operations into the search() function as follows:

```
/* Search. */

foo_t *
slow_search(foo_t *head, int addr)
{
        foo_t *p = head;

        while (p != NULL) {
                rmb();
                if (p->addr == addr) {
                        return (p);
                }
                p = p->next;
        }
        return (NULL);
}
```

This code is functionally correct, but can perform very poorly on Alpha CPUs, since it forces gratuitous memory-

barrier instructions on the searching CPU. If the linked list were very long, it might be cheaper to use explicit locking (although in some existing code, use of explicit locking would cause deadlocks).

A more desireable approach would place little or no overhead on the search code.

3. Replace the `wmb()` in the insert() function with a primitive that does a "global MB shootdown", that is, sends each CPU a high-priority interrupt, then spins until each CPU has executed at least one MB instruction. The cod for the `insert()` function would then look something like the following:

```
/* Insert new after pred.  Caller has initialized *new. */

void
insertMB(foo_t *pred, foo_t *new)
{
          long ticket;

          spin_lock(&mymutex);
          new->next = pred->next;
          ShootdownMB();
          pred->next = new;
          spin_unlock(&mymutex);
}
```

This approach allows the `search()` function to be used unchanged, thereby retaining the full speed of the PTX/x86 implementation on the search side.

4. The need for a special `ShootdownMB()` primitive can be dispensed with by observing that (given the current PTX implementation) a quiescent period also implies a global MB shootdown. Therefore, the `wmb()` in the insert() function can be replaced with a call that blocks for the duration of a quiescent period. Such a call can easily be constructed from the read-copy update primitives:

```
int
rcu_wait_quiescent_period_wake(void *ptr);
{
          sema_t *sp = &((struct rcu_wait_sema *)
                    (ptr - offsetof(struct rcu_wait_sema, rch)))->sema;
          wake_up(sp);
}

struct rcu_wait_sema {
          sema_t sema;
          struct rcu_head rch;
};

void
rcu_wait_quiescent_period()
{
          struct rcu_wait_sema *sp;

          sp = (struct rcu_wait_sema *)kmalloc(sizeof(*sp),
                              GFP_KERNEL|GFP_ATOMIC);
          init_sema(&sp->sema, 0);
          call_rcu(&sp->rch, rc_wait_quiescent_period_wake);
          down(&sp->sema);
          kfree(sp);
}
```

The insertMB() function would then be as follows:

```
/* Insert new after pred.  Caller has initialized *new. */

void
insertMB(foo_t *pred, foo_t *new)
{
          long ticket;

          spin_lock(&mymutex);
```

```
        new->next = pred->next;
        rcu_wait_quiescent_period();
        pred->next = new;
        spin_unlock(&mymutex);
}
```

One problem with this approach from a PTX viewpoint is that `insertMB()` cannot be called from interrupt context. However, PTX is unlikely to ever run on an Alpha, and Digital UNIX does almost all of its work in process/thread context. However, any OS that expects to run on an Alpha which also does significant work in interrupt context would need to implement ShootdownMB() so that blocking is unnecessary.

Another problem is that the blocking approach throttles the update rate. Each update is blocked for the duration of a quiescent period. Therefore, OSes such as Digital UNIX that do most of their work in the process/thread context might consider one of the following approaches.

5. Take the approach outlined above, but use hashing or a similar technique to partition the data. Then separate locks may be used for each part of the data, and multiple updates may proceed in parallel.

   This can greatly speed up updates, but only in cases where multiple threads are updating the data structure simultaneously *and* where the hash function has been carefully chosen. If both search speed and simplicity are paramount, and the update code always runs in process context, this is most likely the method of choice.

6. Batch updates. The idea is to use a conservatively locked supplementary data structure to hold items waiting to be inserted into the main data structure. Items must wait in the supplementary data structure for a full quiescent period to pass (or for a global MB shootdown) before they can be linked into the main data structure. The supplementary data structure is invisible to searching processes.

   Updating processes must examine both the main and the supplementary data structures when making modifications. Conflicting updates must be resolved in such a way that in not prone to indefinitely postponing the update. This can be done by maintaining three supplementary data structures (similar to the three lists kept by the read-copy update mechanism itself). Alternatively, a single supplementary list could be maintained with generation numbers used to determine when a given item could move from the supplementary to the main list.

   One downside of this approach is that it becomes much more difficult for a caller of `insert()` to predict when a change will be visible to callers of `search()`. In most cases, this will not be important, but some algorithms may require a callback or a wakeup mechanism. In constrast, the PTX implementation is able to guarantee that any search starting after the completion of an `update()` will see that update.

   Another downside is that some rather complex machinery is required to manipulate the supplementary lists. Although it may be possible to abstract much of this machinery out into common primitives, the great variety of searches performed with read-copy update make it unlikely that all of it could be pulled out.

7. One final approach (due to Wayne Cardoza) is to impose a small amount of overhead onto the `search()` function. The idea is to define an invalid value for each field touched by the `search()` function. The `update()` function can then maintain a pool of invalidated `foo_t` objects that can safely be inserted into the list accessed by `search()`.

   The `search()` function would check the values as it advances to each item. If one or more is invalid, it refetches the pointer and tries again. The `search()` function is *not* required to check any value that it is not going to use. New items must wait for a full quiescent period (or, alternatively, for a global MB shootdown) after being initialized before they can be added to the pool. Using this approach, the `search()` and `update()` functions would appear as follows:

```
/* Insert new after pred.  Caller has initialized *new. */

void
insert_inv(foo_t *pred, foo_t *new)
{
          foo_t *newinv;

          newinv = new_foo_inv(GFP_KERNEL);
          spin_lock(&mymutex);
          new->next = pred->next;
          pred->next = new;
          spin_unlock(&mymutex);
}

/* Search. */

foo_t *
search_inv(foo_t *head, int addr)
{
          foo_t *p = head;
          foo_t *q = NULL;

          while (p != NULL) {
                    if ((p->addr == INVALID_ADDR) ||
                        (p->next == INVALID_POINTER)) {
                              if (q == NULL) {
                                        p = head;
                              } else {
                                        p = q->next;
                              }

                              /*
                               * Spin waiting for the memory
                               * system to get the new value
                               * out to this CPU.
                               * The semantics of the "mb"
                               * ("memory barrier") instruction
                               * guarantee at most two passes
                               * through this loop.
                               */

                              asm("mb");
                              continue;
                    }
                    if (p->addr == addr) {
                              wmb(); /* if needed. */
                              return (p);
                    }
                    q = p;
                    p = p->next;
          }
          return (NULL);
}


/*
 * Get a new, invalidated foo_t.  A non-toy implementation
 * would maintain per-CPU lists of these, most likely
 * leveraging kmem_cache functionality.
 */

foo_t *
new_foo_inv(flags)
{
          int i;
          foo_t *p == NULL;

          for (;;) {
                    spin_lock(&nfi_mutex);
                    if (nfi_head != NULL) {
                              p = nfi_head;
                              nfi_head = p->nfi_next;
                                        /* Can't use next! */
                              spin_unlock(&nfi_mutex);
                              return (p);
                    } else if (nfiw_head == NULL) {
                              for (i = 0; i < NFI_TARGET; i++) {
                                        p = (foo_t *)
                                                  kmalloc(sizeof *p,
```

```
                                                        flags);
                                        if (p == NULL) {
                                                break;
                                        }
                                        p->next = INVALID_POINTER;
                                        p->addr = INVALID_ADDR;
                                        p->nfi_generation = nfi_gen;
                                        p->nfi_next = nfiw_head;
                                        nfiw_head = p;
                                }
                                if (nfiw_head != NULL) {
                                        rc_callback(nfiw_rc);
                                }
                                if (!CANSLEEP(flags)) {
                                        spin_unlock(&nfi_mutex);
                                        return (NULL);
                                }
                                p_sema_v_lock(&nfi_sema,/*@@@@*/
                                                PZERO + 1,
                                                nfi_mutex);
                        } else {
                                p_sema_v_lock(&nfi_sema,/*@@@@*/
                                                PZERO + 1,
                                                nfi_mutex);
                        }
                }
                /*NOTREACHED*/
}

/*
 * Called at the completion of a quiescent period that
 * started after the above call to call_rcu().
 */

void
new_foo_inv_callback()
{
        spin_lock(&nfi_mutex);
        INSIST(nfiw_head != NULL, "inconsistent state!");
        nfi_head = nfiw_head;
        nfiw_head = NULL;
        vall_sema(&nfi_sema);
        spin_unlock(&nfi_mutex);
}
```

The `wmb()` in `search_inv()` is only required if the caller is going to be using fields in the item that are not guarded *by `search_inv()`* by the invalid-value trick. (Since the caller does not have access to the predecessor item that `search_inv()` tracks in the "q" pointer, it cannot make use of the invalid-value trick itself, except by making additional calls to `search_inv()`).

This approach allows insertions to proceed at full speed. The speed at which the insertions may be applied may be tuned by adjusting the value of `NFI_TARGET`. However, it does impose an overhead on the `search_inv()` function. This overhead will be negligible in the case shown here, since the cost of two compares is way down in the noise. However, searches that examine a large number of fields in each item could be significantly slowed.