Read-Copy Update

Paul E. McKenney
Linux Technology Center
IBM Beaverton
pmckenne@us.ibm.com, http://www.rdrop.com/users/paulmck

Jonathan Appavoo
Department of Electrical and Computer Engineering
University of Toronto
jonathan@eecg.toronto.edu

Andi Kleen
SuSE Labs
ak@suse.de

Orran Krieger
IBM T. J. Watson Research Center
okrieg@us.ibm.com, http://www.eecg.toronto.edu/~okrieg

Rusty Russell
RustCorp
rusty@rustcorp.com.au

Dipankar Sarma
Linux Technology Center
IBM India Software Lab
dipankar.sarma@in.ibm.com

Maneesh Soni
Linux Technology Center
IBM India Software Lab
smaneesh@in.ibm.com

Abstract

Traditional operating-system locking designs tend to either be very complex, result in poor concurrency, or both. These traditional locking designs fail to take advantage of the event-driven nature of operating systems, which process many small, quickly completed units of work, in contrast to CPU-bound software such as scientific applications. This event-driven nature can often be exploited by splitting updates into the two phases: 1) carrying out enough of each update for new operations to see the new state, while still allowing existing operations to proceed on the old state, then: 2) completing the update after all active operations have completed. Common-case code can then proceed without disabling interrupts or acquiring any locks to protect against the update code, which simplifies locking protocols, improves uniprocessor performance, and increases scalability. Examples of the application of these techniques include maintaining read-mostly data structures, such as routing tables, avoiding the need for existence locks (and hence avoiding locking hierarchies with the attendant deadlock issues), and dealing with unusual situations like module unloading.

1 Introduction

Operating systems such as Linux often perform expensive synchronization operations in common code

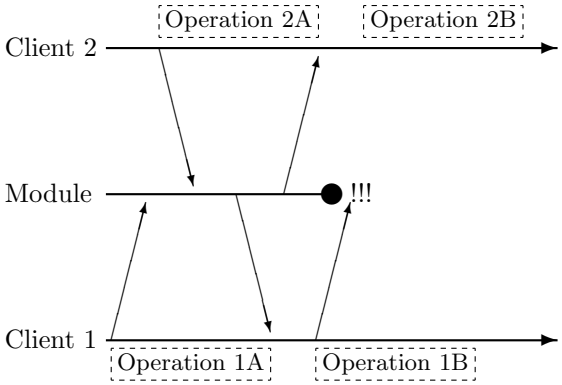

Figure 1: Race Between Teardown and Use of Service

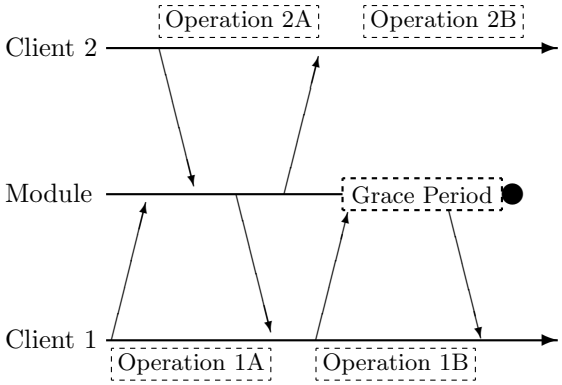

Figure 2: Read-Copy Update Handling Race

paths to protect against infrequent destructive modifications. For example, a read access to a shared file descriptor must acquire a lock to protect against infrequent `file_struct` expansions. As another example, code accessing a kernel module must increment a reference count to protect itself against infrequent module unloads. These synchronization operations on the common code paths result in increased overhead, reduced scalability on a multiprocessor, and are often the sole reason for complex lock hierarchies, which both increase complexity and introduce hard-to-solve deadlock-avoidance issues when locks must be acquired out of order.

Let us consider the example of kernel module unloading. A schematic of accesses racing with the unloading of a module is shown in Figure 1, with time increasing to the right. Operations 1A, 1B, 2A, and 2B denote in-kernel code sequences that do not yield the CPU. The slanted arrows depict invocations of the module and responses from the module. The arrows are not vertical because code executed, interrupts taken, and memory error-correction events can result in delay between a decision to use a module and the unload of that module. For example, when Operation 1B started, the module was loaded, but was unloaded before it could be invoked. Unless Operation 1B has been coded to allow for this, the result is most likely an "oops".

There are a number of ways of handling this race, which will be covered in Sections 2 and 7. One of these ways is read-copy update. The key observation leading to read-copy update is that Operation 2B, which started after the module was unloaded, is not subject to this race condition. This suggests that the module-unloading procedure should provide a "grace period" during which ongoing operations (e.g., Operation 1B) are allowed to continue using the module, but new operations (e.g., Operation 2B) are told that the module is no longer loaded (see Figure 2).

This grace period extends until after the end of all operations that started before the beginning of the grace period. Therefore, any operation that sees the module still loaded is guaranteed to be able to use it. Once the grace period expires, the system may complete any required cleanup, in this case, unmapping the module and freeing up the associated data structures.

The end of a grace period is detected indirectly: when every CPU (or task, in preemptive environments) has passed through a "quiescent state", the grace period may end. A "quiescent state" is a point in the code at which it is guaranteed that all previous operations have completed. For example, in a non-preemptive kernel, context switch is a quiescent state for a given CPU. In Figures 1 and 2, the quiescent states occur between the dashed boxes representing the operations.

Use of quiescent states is pessimistic in the sense that it forces us to wait until *all* pre-existing operations on *all* data structures have completed, when in fact only a few (or none!) of them might be using the data structure of interest. This pessimistic approach allows us to deduce that a given data structure is no longer referenced without having to use locks or atomic operations. This in turn provides good performance and scalability, and may be used in event-driven systems (such as the Linux kernel)

where operations complete quickly.

Implementations may choose different sets of quiescent states, in fact, thus far, four different strategies have been used:

- DYNIX/ptx 2.1 (1993) and Rusty Russell's first `wait_for_rcu()` patch [Russell01a] simply execute onto each CPU in turn. Once they have done this, each CPU has seen at least one context switch. Since context switch on a CPU is a quiescent state for non-preemptive kernels, this procedure finds the end of a grace period, as required. This mechanism is appropriate if blocking and preemption are prohibited in read-side critical sections. These implementations track quiescent states on a per-CPU basis. See Section 4 for an implementation of this approach.

- DYNIX/ptx 4.0 (1994) and Dipankar Sarma's read-copy-update patch for Linux use context switch, execution in the idle loop, execution in user mode, system call entry, trap from user mode, and CPU offline (this last for DYNIX/ptx only) as the quiescent states. Once each CPU has passed through at least one of these quiescent states, any pre-existing operations are guaranteed to have completed. This approach is appropriate for kernels that prohibit blocking and preemption in read-side critical sections. These implementations track quiescent states on a per-CPU basis. See Section 6.2 for a design outline.

- Rusty Russell's second `wait_for_rcu()` patch [Russell01b] uses voluntary context switch as the sole quiescent state. This approach is appropriate for kernels that allow preemptions, but not voluntary context switches, in read-side critical sections. This implementation tracks quiescent states on a per-task basis. See the patch for more details.

- Tornado's and K42's "generation" facility tracks beginnings and ends of operations. When an operation begins, it increments a per-CPU current generation counter. When the operation ends, it decrements this same counter. When the counter goes to zero, then all pre-existing operations that began their execution on that CPU are guaranteed to have completed. This approach is appropriate for kernels that allow both preemption and voluntary context switches in read-side critical sections. These implementations track quiescent states on a per-thread ("task" in Linux) basis. See Section 6.1 for a design outline.

Read-copy update provides a grace period to concurrent accesses by performing destructive updates in two phases: 1) carrying out enough of each update for new operations to see the new state, while allowing pre-existing operations to proceed on the old state, then 2) completing the update after the grace period expires, so that all pre-existing operations have completed. This splitting of destructive updates has resulted in read-copy update being called "two-phase update" in some academic circles.

Read-copy update works best when: 1) it is possible to divide an update into two phases, 2) it is possible for common-case operations to proceed on stale data (e.g. continuing to handle operations by a module being unloaded), and 3) destructive updates are very infrequent. We have found that these situations are very common in existing operating systems. This paper lists how read-copy update has been used in DYNIX/ptx, Tornado, and K42. The paper also shows some "toy" examples and describes patches that provide and use read-copy update in Linux.

2 Toy Example of Read-Copy Update Usage

This section presents a simple circular doubly linked-list example, showing code fragments comparing a reference-counting locking algorithm taken from Linux with its read-copy-update equivalent. This "toy" example illustrates the time and complexity overhead of the different approaches. The same techniques discussed here can be applied to both specific data structures, like this linked-list example, or larger synchronization problems, as discussed in Section 5. Additional locking algorithms are presented in Section 7.

For each algorithm, we present a `search()` and a `delete()` function. The `search()` algorithm returns a pointer to an element in the list given its `addr`, and does whatever operation is required to prevent that element from being freed up. The `delete()` algorithm arranges for the specified element to eventually be freed up. Of course, the

```
 1 struct el {
 2    struct el *next;
 3    struct el *prev;
 4    spinlock_t lock;
 5    long address;
 6    long data;
 7    long refcnt;
 8    long deleted;  /* read-copy only... */
 9    struct kmem_defer_item kd;  /* " " */
10 };
```

Figure 3: List Element Data Structure

```
 1 struct el search(long addr)
 2 {
 3    read_lock(&list_lock);
 4    p = head->next;
 5    while (p != head) {
 6       if (p->address == addr) {
 7          atomic_inc(&p->refcnt)
 8          read_unlock(&list_lock);
 9          return (p);
10       }
11       p = p->next;
12    }
13    read_unlock(&list_lock);
14    return (NULL);
15 }
```

Figure 4: Reference-Counted Search

`delete()` operation may not be able to free up the element immediately due to concurrent searches. Figure 3 shows the list-element data structure used in these `search()` and `delete()` functions.

2.1 Reference-Counted Search and Delete

Figure 4, Figure 5, and Figure 6 show reference-counted search, release, and deletion, respectively. These code fragments are (severely) distilled from `neigh_lookup()` and friends in Linux 2.4.2. These code fragments depart from the `neigh_lookup()` implementation by collapsing `neigh_force_gc()` into the `delete()` function.

Figure 7 shows how the reference-counted `search()` and `delete()` functions might be used. Lines 3 through 5 show how a read-only operation might be carried out. Note that this read-only access still results in cachelines being bounced by lines 3, 7, and 8 of Figure 4 and by line 3 of Figure 5. Lines 9

```
1 static void release(struct el *p)
2 {
3    if (atomic_dec_and_test(&p->el_refcnt) {
4       kfree(p);
5    }
6 }
```

Figure 5: Reference-Counted Release

```
1 struct el delete(struct el *p)
2 {
3    write_lock(&list_lock);
4    p->next->prev = p->prev;
5    p->prev->next = p->next;
6    release(p);
7    write_unlock(&list_lock);
8 }
```

Figure 6: Reference-Counted Deletion

through 15 of Figure 7 show how an update operation, possibly including a deletion, might be carried out. Note that although the cacheline bouncing in lines 3 and 8 of Figure 4 can be greatly reduced by using a brlock, this change would make the write-side locking on lines 3 and 7 of Figure 6 much more costly.

2.2 Read-Copy Update Search and Delete

Figure 8 and Figure 9 show read-copy search and deletion, respectively.

```
 1 /* Read-only access. */
 2
 3 p = search(addr);
 4 /* Read-only access to p. */
 5 release(p);
 6
 7 /* Access and deletion. */
 8
 9 p = search(addr);
10 /* Access and update p. */
11 if (to_be_deleted) {
12    delete(p);
13 } else {
14    release(p);
15 }
```

Figure 7: Reference-Counted search/delete Usage

```
 1 struct el *search(long addr)
 2 {
 3    struct el *p;
 5    p = head->next;
 6    while (p != head) {
 7       if (p->address == addr) {
 8          return (p);
 9       }
10       p = p->next;
11    }
12    return (NULL);
13 }
```

Figure 8: Read-Copy Search

The `search()` function can return a reference to an already-deleted element, but the `kfree_rcu()` guarantees that the element will not be freed (and thus possibly re-used for some other purpose) while this reference exists (see Figure 20 for a definition of `kfree_rcu()`). There are a number of techniques that may be used to ensure that `search()` returns references only to elements that have not yet been deleted; see Section 7.3 for an example. However, there are quite a few algorithms that tolerate "stale data", for example, many algorithms that track state external to the machine must deal with stale data in any case due to communications delays.

The delete function is quite similar to that of a single-threaded application, with the addition of locking, and with `kfree()` replaced by `kfree_rcu()`. The internal implementation of `kfree_rcu()` waits for a grace period before freeing the specified block of memory (see Section 4.2), and also provides the required read-write barriers that allow this function to execute correctly on weakly consistent machines.

The `search()` function contains absolutely *no* locks or atomic instructions, which means that the performance of this function will scale with CPU core clock rate, rather than the much slower memory latencies for an implementation based on locks or atomic operations. In addition, the `search()` does not disable interrupts, which means that read-copy update can improve performance of UP as well as SMP kernels. However, `search()` can return stale data. This can be prevented, if need be, see for example Section 7.3.

Note that `delete()` is very similar to its reference-count counterpart, including the global lock. This particular implementation will therefore give good

```
1 void delete(struct el *p)
2 {
3    spin_lock(&list_lock);
4    p->next->prev = p->prev;
5    p->prev->next = p->next;
6    spin_unlock(&list_lock);
7    kfree_rcu(p, NULL);
8 }
```

Figure 9: Read-Copy Deletion

```
 1 /* Read-only access. */
 2
 3 p = search(addr);
 4 /* Read-only access to the structure. */
 5 /* Next yield of CPU acts as release. */
 6
 7 /* Access and deletion. */
 8
 9 spin_lock(&list_lock);
10 p = search(addr);
11 /* Access and update p. */
12 spin_unlock(&list_lock);
13 if (to_be_deleted) {
14    delete(p);
15 }
16 /* Next yield of CPU acts as release. */
```

Figure 10: Read-Copy search/delete Usage

speedups only if there are many more searches than deletions. In many situations (e.g., routing-table updates), this will be the case. In other situations, the deletion function might use a more complex but more highly parallel design.

Figure 10 shows how the read-copy `search()` and `delete()` functions might be used. Line 3 shows how a read-only operation might be carried out. Note that there is absolutely no cacheline bouncing if all operations are read-only. Lines 9-15 show how an update operation, possibly including a deletion, might be carried out. The `list_lock` serializes concurrent modifications.

2.3 Discussion

The reference-count and read-copy `search()` and `delete()` functions each have their strengths. The read-copy functions avoid all cacheline bouncing for reading tasks, but can return references to deleted elements, and cannot hold a reference to elements across a voluntary context switch. There are hybrid

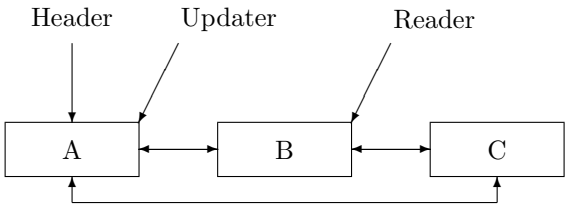

Figure 11: List Initial State

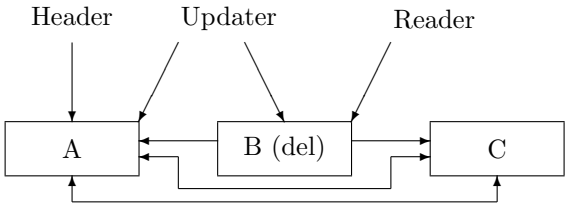

Figure 12: Element B Unlinked From List

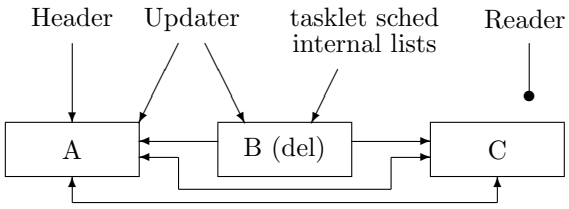

Figure 13: List After Grace Period

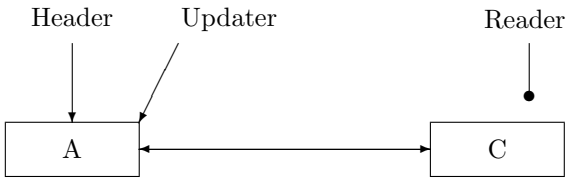

Figure 14: List After Element B Returned to Freelist

designs that combine the strengths of these techniques; a few of these are shown in Section 7.

2.4 Read-Copy Deletion Animated

This section steps through the read-copy deletion function shown in Figure 9 using the example list shown in Figure 11.

To delete element B, the updater task acquires `list_lock` to exclude other list manipulation (line 3 in Figure 9), unlinks element B from the list (lines 4 and 5), and releases `list_lock` (line 6). This results in the situation shown in Figure 12. This action constitutes the first phase of the update.

At this point, any subsequent searches of the list (see Figure 8) will find that element B has been deleted. However, ongoing searches, such as that of the reader task (which is executing line 6 of Figure 8, may still find element B: these tasks see stale data. This stale data has been flagged so that it may be easily ignored [Pugh90], as illustrated in Section 7.3. In some cases, stale data may be tolerated, for example, data representing external state, such as routing tables, can be stale in any case due to unavoidable update delays.

Finally, the updater task passes a pointer to element B to the `kfree_rcu()` primitive (line 7 of Figure 9), which adds the memory to a list waiting to be freed, as shown in Figure 13.

The question answered by read-copy update is "when is it safe to return element B to the freelist?" The answer is "as soon as each pre-existing operation completes", since any new searches will be unable to acquire a reference to element B. All pre-existing operations are guaranteed to have completed at the end of a grace period. The end of the grace period is detected via quiescent states, as noted earlier.

At this point, `kfree_rcu()` can safely return element B to the freelist potentially for immediate reuse, as shown in Figure 14.

For this return to freelist to be safe, the reader task must be prohibited from retaining a reference to element B across a quiescent state. This is equivalent to the prohibition against maintaining similar references outside of the corresponding critical sections in traditional locking. In either case, the data structure might be arbitrarily modified in the meantime, possibly rendering the reference invalid.

2.5 Architectural Trends and Locking Algorithms

Figure 15 shows the historical memory-latency ratios for Sequent's computers. The latency ratio shown in this figure represents the number of in-

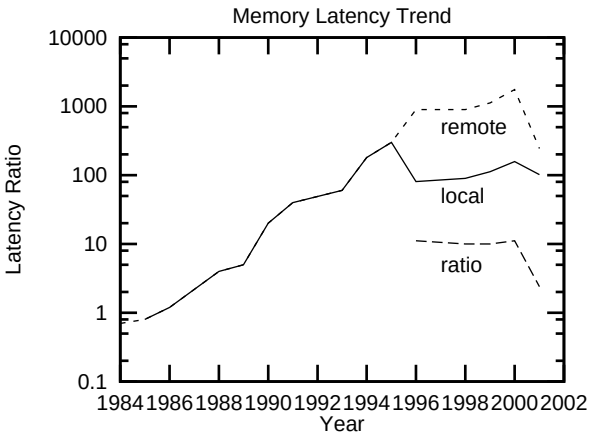

Figure 15: Memory-Latency Ratios for Sequent Computers

structions that could be executed in the time required to do a single load from memory. The line bifurcates in 1996 due to the introduction of NUMA hardware: the upper line (labeled "remote") represents the number of instructions that can be executed during the time required to do a single load from remote memory, and the middle line (labeled "local") represents the number of instructions that can be executed during the time required to do a single load from local memory.

The trend is consistently upwards towards higher memory-latency ratios, thanks to Moore's law. The two exceptions to this trend are the reduction in local memory latencies in 1996 (due to adoption of the NUMA architecture), and the reduction in both local and remote memory latencies in 2001 (due to adoption of a hardware crossbar interconnect and to a combination of relatively fast local memory latencies and relatively slow CPU clock rate on Itanium CPUs compared to contemporary Pentium CPUs or to McKinley expectations).

A key observation here is that traditional locking is limited by worst-case memory latency. This is because each acquisition of the lock must do a write to that lock's data structure, so that the lock's data structure will normally only be in "modified" state in the cache of the last acquiring CPU. The next acquiring CPU will therefore likely incur a remote-memory-access latency penalty when acquiring the lock. Improvements in the speed of lock acquisition will therefore be limited by the slow improvements in remote memory latency, rather than the much faster improvements in CPU core speed [Hennes91, Stone91, Burger96].

This trend illustrates the increasingly large performance penalties of cacheline bouncing, which in turn motivates use of locking techniques that are not limited by remote memory latency. In read-mostly situations, brlock and read-copy update are examples of such techniques.

3 Conditions and Assumptions

Use of read-copy update is most likely to be helpful with read-intensive data structures, where stale data may be either tolerated or suppressed, and where event-driven operations complete quickly.

"Read intensive" means that the update fraction (ratio of updates to total accesses) f is much smaller than the reciprocal of the number of CPUs: if you have eight CPUs, then a "read-intensive" workload would have f much less than 0.125. It is possible for f to be as small as 10^{-10}, for example, in storage-area network (SAN) routing tables (consider 100 disks, each with 100,000-hour mean time between failure, connected to a system doing 3,000 I/Os per second). However, in some special cases, read-copy update can provide performance benefits even though f exceeds 0.9 [McK98a].

Use of a grace period means that reading tasks can see stale data. However, any reading task that starts its access after the first phase of an update is guaranteed to see the new data. This guarantee is sufficient in many cases. In addition, data structures that track state of components external to the computer system (e.g., network connectivity or positions and velocities of physical objects) must tolerate old data because of communication delays. In other cases, old data can be flagged so that the reading task can detect it and take explicit steps to obtain up-to-date data, if required [Pugh90], as shown in Section 7.3.

Read-copy update requires that the modification be compatible with lock-free access. For example, linked-list insertion, deletion, and replacement are compatible: a reading access will see either the old or new state of the list. However, if a list is reordered in place, the reading task can be forced into an infinite loop if the last element is consistently moved to the front of the list each time a reading task reaches it. It is possible to perform an arbitrary read-copy-update modification of any data structure by making a copy of the entire struc-

ture, but this is inefficient for large structures. As we gain more experience with read-copy update, we expect to learn how to efficiently transform more general modifications into read-copy update form.

Another issue with read-copy update is that a modest amount of memory must be available to track memory waiting to be freed, and to allow for the fact that memory is not freed as soon as it would be when using traditional locking designs. However, given the ever-decreasing cost of memory, this issue does not cause much trouble in practice.

Finally, read-copy update is less applicable to non-event-drive software, such as some CPU-bound scientific applications, although similar techniques have been used, as reviewed by Adams [Adams91].

4 Simple Infrastructure to Support Read-Copy Update

This section presents a simple implementation of read-copy-update infrastructure in two parts: (1) a `wait_for_rcu()` primitive that waits for a grace period to expire, and (2) a `kfree_rcu()` primitive that waits for a grace period before freeing a specified block of memory. Please note that the APIs defined in this section are under development and thus subject to change.

4.1 Simple Grace-Period Detection

Figure 16 shows how read-copy update progresses. The boxes represent non-preemptible kernel execution, the space between them represents quiescent-state execution (e.g., context switch, user mode, idle loop, or user-mode execution), and each numbered arrow represents an active entity, for example, a CPU or a task, with time progressing to the right.[1]

The leftmost dashed line indicates the time of the first phase of the read-copy update (e.g., lines 3 through 6 in Figure 9). The second phase of the update (e.g., the actual freeing in the `kfree_rcu()` on line 7 of Figure 9, a simple version of which is shown in Section 4.2) may proceed as soon as all operations that were in progress during the first phase have completed, namely, operations A, E, and L. The earliest time the second phase can safely be initiated is indicated by the rightmost dashed line in Figure 16, and the distance between the two dashed lines is the minimum allowable duration of the grace period – during this time, there may exist tasks that still hold references to element B in Figure 12.

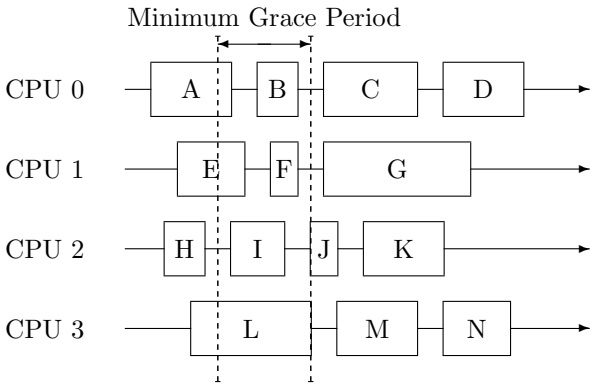

Figure 16: Read-Copy Update Grace Period

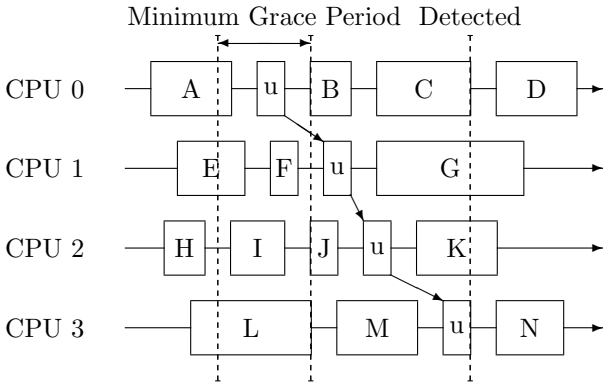

Figure 17: Simple Grace-Period Detection

A simple procedure to determine when the second phase may safely be initiated in a non-preemptive operating-system kernel is depicted in Figure 17. The updater simply forces itself to execute on each CPU in turn. The boxes labeled "u" represent this updater's execution. Once it has run on each CPU, then the non-preemptive nature of the Linux kernel guarantees that all operations that were in progress during phase one must have completed. Rusty Russell's `wait_for_rcu()` primitive, shown in Figure 18, is an example of this procedure.

[1] Rusty Russell [Russell01b] describes one way of relaxing this restriction, so that involuntary context switches (preemptions) are permitted in read-side critical sections. Section 6.1 describes another approach that could be used.

```
 1 void wait_for_rcu(void)
 2 {
 3    unsigned long cpus_allowed;
 4    unsigned long policy;
 5    unsigned long rt_priority;
 6    /* Save current state */
 7    cpus_allowed = current->cpus_allowed;
 8    policy = current->policy;
 9    rt_priority = current->rt_priority;
10    /* Create an unreal time task. */
11    current->policy = SCHED_FIFO;
12    current->rt_priority = 1001 +
13    sys_sched_get_priority_max(SCHED_FIFO);
14    /* Make us schedulable on all CPUs. */
15    current->cpus_allowed =
16                (1UL<<smp_num_cpus)-1;
17
18    /* Eliminate current cpu, reschedule */
19    while ((current->cpus_allowed &= ~(1 <<
20            cpu_number_map(
21             smp_processor_id()))) != 0)
22       schedule();
23    /* Back to normal. */
24    current->cpus_allowed = cpus_allowed;
25    current->policy = policy;
26    current->rt_priority = rt_priority;
27 }
```

Figure 18: Non-Preemptible Grace-Period Detection

Lines 7 through 13 save the current scheduling state, and set up a FIFO scheduling policy with sufficient priority to preempt all other tasks. Lines 15 and 16 create a mask that allows the task to run on any CPU. The loop on lines 19 through 22 repeatedly eliminates the current CPU from the set allowed to run this task, then yields the CPU. Thus, upon loop completion, the task will have run on each of the CPUs, which means that each CPU will have completed whatever it was doing at the time of the call to `wait_for_rcu()`. In the case of the read-copy deletion algorithm in Figure 9, this in turn means that it is now safe to free up the memory passed to `kfree_rcu()`. Lines 24 through 26 restore the scheduling state.

This code is quite straightforward, but it does have a few shortcomings: (1) it would not work in a preemptible kernel unless preemption is suppressed in all read-side critical sections, (2) it cannot be called from an interrupt handler (but `schedule_task()` can be used to call it indirectly), (3) it cannot be called while holding a spinlock or with interrupts disabled (but, again, `schedule_task()` can be used to call it indirectly), and (4) it is relatively slow. Rusty Russell's patch [Russell01b] and Section 6.1 describe two possible ways of addressing item 1. The following section describes the `kfree_rcu()` primitive that addresses items 2 and 3. Section 6.2 describes a faster grace-period-detection algorithm for non-preemptible read-side critical sections that addresses items 2, 3, and 4.

Another way of addressing items 2 and 3 is to define a `call_rcu()` function that queues callbacks onto a list. A separate `free_pending_rcus()` function can then invoke all the pending callbacks after forcing a grace period. Figure 19 shows a straightforward implementation of these two functions. Note that it makes sense to invoke `free_pending_rcus()` when memory pressure needs to be applied.

4.2 Simple Deferred Free

This section describes a simple implementation of a deferred-free function named `kfree_rcu()`.

In many cases, a (mythical) `destroy_soon()` primitive would be ideal – just pass a pointer to the data structure that is to be freed up during the second phase of read-copy update, then go about your business. Unfortunately, there is no safe implementation

```
 1 void call_rcu(struct rcu_head *head,
 2               void (*func)(void *head))
 3 {
 4    unsigned long flags;
 5
 6    head->destructor = func;
 7    spin_lock_irqsave(&rcu_lock, flags);
 8    head->next = rcu_list;
 9    rcu_list = head;
10    spin_unlock_irqrestore(&rcu_lock, flags);
11 }
12
13 void free_pending_rcus(void)
14 {
15    struct rcu_head *list;
16
17    spin_lock_irq(&rcu_lock, flags);
18    list = rcu_list;
19    rcu_list = NULL;
20    spin_unlock_irq(&rcu_lock, flags);
21
22    /* If list nonempty, wait and destroy. */
23    if (list) {
24       wait_for_rcu();
25       while (list) {
26          struct rcu_head *next = list->next;
27
28          list->destructor(list);
29          list = next;
30       }
31    }
32 }
```

Figure 19: Non-Blocking Grace-Period Detection

of `destroy_soon()` that can be called from interrupt handlers or with locks held. This is because we do not want to wait for an entire grace period inside the function: we want to queue the object for destruction and return as quickly as possible, without incurring gratuitous context switches. Unfortunately, any function allocating memory must be able to either block or fail, and `destroy_soon()` can do neither.

Instead, we allocate the memory required for the list when the memory is initially allocated by using `kmalloc_rcu()`, which is shown in lines 7 through 16 of Figure 20. The memory may be freed by invoking `kfree_rcu()`, as shown in lines 28 through 40. If the memory to be freed consists of several linked blocks (e.g., a linked list), a "destructor" function may be passed to `kfree_rcu()`. This destructor function is responsible for freeing up the linked blocks, and `kfree_rcu()` frees up the block whose pointer was initially passed in.

The `kfree_rcu()` function uses an auxiliary `sync_and_destroy()` function (see lines 18 through 26) that is run as a tasklet on line 39 to do the actual freeing. A `struct rcu_head` is used to queue the tasklet.

This approach works well, but still has low performance, since each and every call to `kfree_rcu()` results in a call to `wait_for_rcu()`, which incurs one context switch per CPU. In addition, some small but distracting modifications are required to allow memory from slab allocators to be deferred freed and to handle more general alignment constraints. Finally, in some cases, the `struct rcu_head` can be unioned into the data structure being deferred freed.

A more complex implementation that addresses these issues is described in Section 6.2.

5 Using Read-Copy Update

This section walks through implementations of scalable FD management, hotplug CPU support, and module unloading that use read-copy update.

```
 1 struct rcu_head
 2 {
 3    struct tq_struct task;
 4    void (*destructor)(void *obj);
 5 };
 6
 7 void *kmalloc_rcu(size_t size, int flags)
 8 {
 9    struct rcu_head *ret;
10
11    size += L1_CACHE_ALIGN(sizeof(*ret));
12    ret = kmalloc(size, flags);
13    if (!ret)
14       return NULL;
15    return ret + 1;
16 }
17
18 static void sync_and_destroy(void *rcu_head)
19 {
20    struct rcu_head *head = rcu_head;
21
22    wait_for_rcu();
23    if (head->destructor != NULL)
24       head->destructor(head + 1);
25    kfree(head);
26 }
27
28 void kfree_rcu(void *obj,
29                void (*destructor)(void *))
30 {
31    struct rcu_head *head;
32
33    head = (struct rcu_head *)obj - 1;
34
35    head->task.sync = 0;
36    head->task.routine = &sync_and_destroy;
37    head->task.data = head;
38    head->destructor = destructor;
39    schedule_task(&head->task);
40 }
```

Figure 20: Simple Deferred Free

```
 1 if (i) {
 2    memcpy(new_openset, files->open_fds,
 3           files->max_fdset/8);
 4    memcpy(new_execset, files->close_on_exec,
 5           i * sizeof(struct file *));
 6    memset(&new_openset->fds_bits[i], 0, count);
 7    memset(&new_execset->fds_bits[i], 0, count);
 8 }
 9 nfds = xchg(&files->max_fdset, nfds);
10 new_openset = xchg(&files->open_fds,
11     new_openset);
12 new_execset = xchg(&files->close_on_exec,
13     new_execset);
14 write_unlock(&files->file_lock);
15 free_fdset(new_openset, nfds);
16 free_fdset(new_execset, nfds);
17 write_lock(&files->file_lock);
```

Figure 21: Expanding FD Array

5.1 Scalable FD Management Using Read-Copy-Update

FD management maintains the data structures that map from a file descriptor to the corresponding `struct file`. This mapping is implemented as a set of arrays (pointed to by `fd`, `close_on_exec`, and `open_fds`), which can grow as the process opens more files.

The current FD management code uses a reader-writer lock (`file_lock`) to guard the `files_struct` state, in particular, the `fd`, `close_on_exec`, and `open_fds` pointers. The read-copy-update modifications replace the reader-writer `file_lock` with a spinlock; `read_lock()` calls are deleted, and `write_lock()` calls are replaced with `spin_lock()`.

The `expand_fd_array()` and `expand_fdset()` functions are then cast into read-copy-update form, with the update split into two phases separated by a grace period.

The original form of the update portion of `expand_fd_array()` is shown in Figure 21. In the read-copy-update version, lines 1 through 13 are executed in the first phase, and lines 15 and 16 is executed after a grace period, using the `wait_for_rcu()` function to defer execution of the `free_fdset()` functions. This approach allows any tasks running on other CPUs that are still referencing the arrays pointed to by the old values of `fd`, `close_on_exec`, and `open_fds` to continue normally.

```
 1 if (i) {
 2    memcpy(new_openset, files->open_fds,
 3           files->max_fdset/8);
 4    memcpy(new_execset, files->close_on_exec,
 5           i * sizeof(struct file *));
 6    memset(&new_openset->fds_bits[i], 0, count);
 7    memset(&new_execset->fds_bits[i], 0, count);
 8 }
 9 RC_MEMSYNC();
10 new_openset = xchg(&files->open_fds,
11     new_openset);
12 new_execset = xchg(&files->close_on_exec,
13     new_execset);
14 RC_MEMSYNC();
15 nfds = xchg(&files->max_fdset, nfds);
16 write_unlock(&files->file_lock);
17 wait_for_rcu();
18 free_fdset(new_openset, nfds);
19 free_fdset(new_execset, nfds);
20 write_lock(&files->file_lock);
```

Figure 22: Read-Copy Expanding FD Array

A read-copy version is shown in Figure 22. This code must install the new arrays before updating `max_fdset`, since read-side critical sections are no longer excluded when running this code. The `RC_MEMSYNC()` calls are needed to maintain memory ordering on CPUs with extremely weak memory consistency. The `expand_fdset()` function is modified in a similar fashion, see the patch [Soni01b] for more details.

This patch uses a slightly different approach from that shown in Figure 22. Rather than using `wait_for_rcu()`, it registers read-copy callbacks, which asynchronously invoke auxiliary functions to free the memory after the grace period expires. This somewhat more complex approach is necessary for good performance, as the `wait_for_rcu()` approach results in extra context switches, whose overhead overwhelms read-copy update's performance gains in this case. Future work includes measuring performance using the `kfree_rcu()` interface.

Figure 23 shows the performance benefits of the read-copy version of FD management on the chat benchmark with rooms=20 and messages=500 in a 2.4.2 SMP kernel. These runs used a 1-way, 2-way, 3-way, and a 4-way PIII Xeon 700MHz system with 1MB L2 cache and 1GB of RAM. The read-copy update version attains over 30% more throughput at four CPUs, which should benefit all multithreaded applications that do heavy disk or network I/O. In

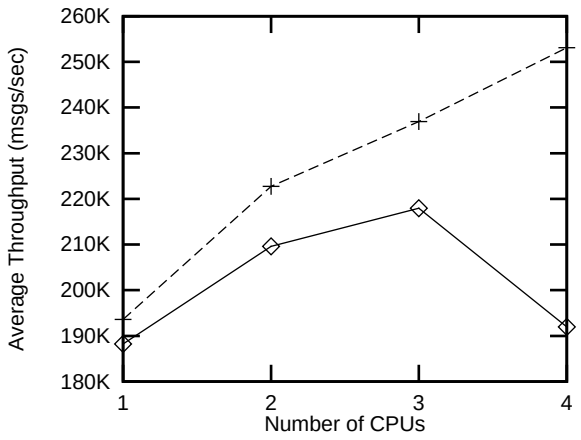

Figure 23: FD Management Performance

addition, this change does not penalize uniprocessor kernels, instead showing a statistically insignificant performance increase (0.65%). In all cases, kernprof measurements revealed greatly reduced hits in the `fget()` function. Since there was no sign of heavy contention on the lock used in this code, it is probable that the increased throughput was due to reduced cacheline bouncing.

5.2 Hotplug CPU Support Using Read-Copy Update

Hotplug CPU support allows a given CPU to be taken offline, so that it will not schedule tasks, take interrupts, or run tasklets. This is useful for benchmarking (as it allows you to vary the number of CPUs available without a reboot) and to remove a CPU that is showing signs of immanent failure (such as high soft-error rate in the CPU's caches).

One consequence of hotplug CPU support is that the current online processors array is no longer read only. Traditional approaches would require that locking be added to all existing code accessing this array. Such a change would be intrusive and awkward. To avoid this pervasive change, the hotplug-CPU-support patch uses read-copy update, which allows code accessing this array to remain unchanged, with no locking whatsoever.

The essence of hotplug CPU support is captured by the `cpu_down()` function, which is shown in Figure 24. Lines 4 through 6 acquire `cpucontrol` to ensure that only one CPU is being taken out of service at a time. Lines 7 through 10 give error EINVAL if

```
 1 int cpu_down(unsigned int cpu)
 2 {
 3    int ret;
 4    if ((ret =
 5       down_interruptible(&cpucontrol)) != 0)
 6       return (ret);
 7    if (!cpu_online(cpu)) {
 8       ret = -EINVAL;
 9       goto out;
10    }
11    if (num_online_cpus() == 1) {
12       ret = -EBUSY;
13       goto out;
14    }
15    current->cpus_allowed = (1<<cpu);
16    schedule();
17    if (smp_processor_id() != cpu)
18       BUG();
19    current->cpus_allowed = -1;
20    ret = __cpu_disable(cpu);
21    if (ret != 0) goto out;
22    if (cpu_online(cpu))
23       BUG();
24    __wait_for_rcu();
25    notifier_call_chain(&cpu_chain,
26                        CPU_OFFLINE,
27                        (void *)cpu);
28    __cpu_die(cpu);
29 out:
30    up(&cpucontrol);
31    return ret;
32 }
```

Figure 24: Hotplug CPU Support

the specified CPU has already been taken out of service. Lines 11 through 14 give error EBUSY if this is the last CPU in service. Lines 15 and 16 switches this task onto the departing CPU, and lines 17 and 18 verify that this switching in fact occurred. Line 19 allows the next context switch to place us on one of the remaining CPUs, and line 20 disables the departing CPU. Line 21 returns the specified error if `__cpu_disable()` fails. Lines 22 and 23 verify that the CPU was in fact disabled. Line 24 uses read-copy update to ensure that all other CPUs become aware of the specified CPU's departure. Then, lines 25 through 27 invoke all the registered notifiers, and line 28 does final cleanup for the departing CPU. Lines 30 and 31 release `cpucontrol` and return any error indication.

The `cpu_up()` function also uses read-copy update to ensure that all CPUs are aware of the arriving CPU before the notifiers are invoked.

Read-copy update allowed hotplug CPU support to be implemented with minimal impact to the rest of the system.

5.3 Module Unloading Using Read-Copy Update

Module unloading in Linux 2.4 is subject to a destructive race when an attempted use of a given module races with unloading of that module. Although the module-unloading code marks the module as "unloaded", it is still possible for a call to that module to see the module as loaded, but not get around to performing the module invocation until after the module was fully unmapped, and all associated data was freed. This race can cause tasks attempting to use the module to access already-freed memory. One of the authors (Maneesh) modified the module-unloading code to use read-copy update, eliminating the race. This modification works by leaving the data structures in place for a grace period, so that racing calls to the kernel module always either see a valid data structure marked "unloaded" or no data structure at all.

Please note that this code is a work in progress that is intended *only* to demonstrate use of read-copy update. We absolutely do *not* recommend that this change be incorporated into the Linux kernel in its current form.

This modification requires that the module writer make some changes to the module:

1. Insert a call to `kmod_def_cleanup()` in the module's cleanup routine.
2. If the module has an open-read/write-release format, insert a call to `MOD_INC_USE_COUNT` in the open function and insert a call to `MOD_DEC_USE_COUNT` in the release function. This allows processes to safely block between the time that they open and close a file.
3. Insert a call to `MOD_INC_USE_COUNT` before any blocking operation in the module code, and insert a call to `MOD_DEC_USE_COUNT` after any such blocking operation. This prevents the module from being unloaded while a given task is blocked inside the module.

In summary, read-copy update protects against uses

```
1 typedef struct module_data_destructor
2    module_data_destructor_t;
3 typedef void
4    (*module_data_destructor_func_t)(
5             void *arg);
6 struct module_data_destructor {
7    module_data_destructor_func_t func;
8    void *arg;
9 };
```

Figure 25: Module Data Destructor Structure

```
1 int
2 kmod_def_cleanup(struct module *mod,
3    module_data_destructor_t *mdd)
4 {
5    rc_callback_t   *cb;
6    if (!(cb = rc_alloc_callback(
7       (rc_callback_func_t)kmod_cu_cb_fn,
8        mod, mdd, GFP_ATOMIC))) {
9       return -ENOMEM;
10   }
11   rc_callback(cb);
12   return 0;
13 }
```

Figure 26: kmod Deferred Cleanup

```
1 void rc_callback(rc_callback_t *rp)
2 {
3    wait_for_rcu();
4    (*(rp->callback))(rp, rp->arg1,
5                      rp->arg2);
6 }
```

Figure 27: Register Read-Copy Callback

```
1 static int
2 kmod_cu_cb_fn(rc_callback_t *cb,
3               struct module *mod,
4               module_data_destructor_t *mdd)
5 {
6    if (mdd)
7        mdd->func(mdd->arg);
8    module_unmap(mod);
9    rc_free_callback(cb);
10   return 0;
11 }
```

Figure 28: kmod Cleanup Function

of the module while it is being unloaded, while the reference count prevents the module from being unloaded while a task is blocked inside it.

The `kmod_def_cleanup()` function in Figure 26 takes a pointer to struct module as its first argument and a pointer to a `module_data_destructor_t` as its second argument. The latter is defined as shown in Figure 25. The `func` field contains a pointer to a function that is invoked after a grace period starting at the beginning of the module unload, and the `arg` field contains an arbitrary value that is passed to this function. This function is used to allow the module to do final cleanup of resources needed by module uses that race with the unload operation.

The `kmod_def_cleanup()` function is shown in Figure 26. Lines 6 through 10 attempt to allocate and initialize a read-copy callback structure, returning `ENOMEM` upon failure. Line 11 registers a read-copy callback. This callback will be invoked after a grace period, resulting in a call to `kmod_cu_cb_fn(cb, mod, mdd)`.

The `rc_alloc_callback()` function simply allocates an `rc_callback_t` structure and initializes its fields to contain the specified function (in this case, `kmod_cu_cb_fn()`) and two arguments to be passed to this function (in this case, `mod` and `mdd`). A simple definition of `rc_callback()` is shown in Figure 27. This executes a `wait_for_rcu()`, then invokes the callback, allowing for final cleanup. A higher-performance implementation of this function is described in Section 6.2.

When invoked from `kmod_def_cleanup()`, the `rc_callback()` function will call `kmod_cu_cb_fn()`, since this function was passed to `rc_alloc_callback` on line 7 of Figure 26. The definition of `kmod_cu_cb_fn()` is shown in Figure 28. Lines 6 and 7 of Figure 28 invoke the module-defined cleanup function, if one exists. Line 8 unmaps the module, and line 9 frees the callback data structure. Finally, line 10 returns to the caller.

The effect of these changes is to defer module cleanup until all racing uses of that module have finished. This eliminates destructive races. More information is available with the patch [Soni01a].

6 Advanced Infrastructure for Read-Copy Update

This section describes the design used in the Tornado and K42 research operating systems [Gamsa99] and a more complex but higher-performance design for non-preemptive Linux kernels. Most, and perhaps all, of the optimizations found in this last implementation can easily be applied to the implementation described in Section 4.

6.1 Tornado/K42 Design for Read-Copy Update

The K42 and Tornado implementations of read-copy update are such that read-side critical sections can block as well as being preempted. For each CPU, the scheduler maintains two generation counters. At any given time one counter is identified as the current generation. When a operation (such as a system call or interrupt) begins, it is associated with the current generation by incrementing the current counter and storing a pointer to that counter in the task. When the operation ends, the corresponding generation counter is decremented. Periodically, the non-current generation is checked to see if it is zero, indicating that all associated operations have terminated. When this happens, the roles of the current and non-current generations are reversed. A separate per-CPU generation sequence number is advanced every time a new generation is identified as current. Given such mechanisms, when the generation sequence has advanced twice we can be assured that all operations in existence prior to the advancement have terminated on the specific CPU. In order to know when all operations have terminated across all the CPUs, a token is constantly circulated across all CPUs. The token is handed from one CPU to the next CPU when the generation sequence has advanced by at least two on the CPU in current possession of the token. Thus when the token returns to a given CPU all operations across the entire system that were in existence, since the last time the CPU had the token, have terminated.

Read-copy update is used pervasively within K42 and Tornado, and is available to user applications and libraries as well as within the kernel. However, systems calls such as `recvmsg()`, which can block indefinitely, must be carefully coded so that this long-term blocking is not considered to be part of an operation.

6.2 High-Performance Design for Read-Copy Update

The read-copy update implementation in Section 4 works well in many situations. However, some additional capabilities can be beneficial in some situations:

1. Providing a function that is capable of doing deferred frees of `kmem_cache_alloc()` memory, as well as of more complex data structures such as linked lists and trees.
2. Providing a mechanism that detects and identifies overly long lock-hold durations, which could otherwise make grace periods excessively long and degrade overall response times.
3. "Batching" grace-period-measurement requests so that a single (expensive) invocation of grace-period measurement can satisfy multiple requests. This can be accomplished by having a list of requests, with each request containing a pointer to a callback function and arguments.
4. Maintaining per-CPU request lists in order to further reduce the per-request overhead of measuring grace periods.
5. Providing a less-costly algorithm for measuring grace-period duration.

Each of these features could potentially be added to the algorithms described in the preceding sections, if required. This section describes the read-copy-update patch that was ported from DYNIX/ptx, which supports all of these capabilities and which was used in the "chat" benchmark runs described in Section 5.1.

The overall data-structure design of this algorithm is shown in Figure 29.

This design uses 'callbacks" that allow code that cannot block to schedule the phase-2 work via a function call, in a manner similar to the tasklet approach used in Section 4.2. These callbacks are placed on the `next`, `current`, and `intr` lists shown

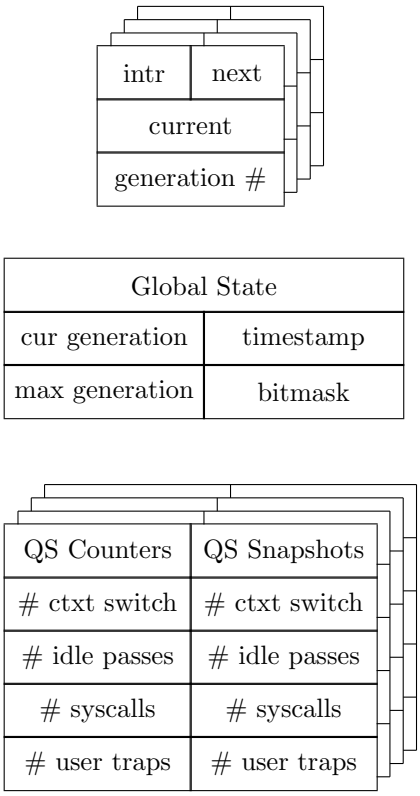

Figure 29: High-Performance Read-Copy Update Data Structures

in the figure. A `wait_for_rcu()` function can easily be implemented using these callbacks.

Item 1 is addressed by providing a `kmem_deferred_free()` function, which takes a pointer to the memory to be deferred-freed, a pointer to a cleanup function, as well as a pointer to a portion of the memory that is used to thread the memory onto a per-CPU list as shown in Figure 30. Once the grace period has expired, the cleanup function is invoked. This cleanup function may then free up the structure(s), using any desired mechanism.

Item 2 is addressed by recording a `timestamp` at the beginning of a grace period. If a given CPU takes too long arriving at a quiescent state, it can print diagnostic information to help track down the offending code path.

Item 3 is also addressed by the lists, since multiple callbacks on the lists (representing multiple requests) can be satisfied by a single set of quiescent states. This batching can greatly improve performance if many update-side read-copy operations are in flight simultaneously.

```
 1 void kmem_deferred_free(
 2          void *ptr,
 3          int (*func)(void *),
 4          struct kmem_defer_item *kp)
 5 {
 6    int cpu = smp_processor_id();
 7    kp->kdi_next = NULL;
 8    kp->kdi_ptr = ptr;
 9    kp->kdi_func = func;
10    __cli();
11    *KMEM_DEFER_PERCPU(cpu)->kmemd_tail = kp;
12    KMEM_DEFER_PERCPU(cpu)->kmemd_tail =
13       &kp->kdi_next;
14    if (KMEM_DEFER_PERCPU(cpu)->kmemd_idle) {
15       kmemd_register_percpu();
16       return;
17    }
18    __sti();
19    return;
20 }
```

Figure 30: Fast Deferred Free

Item 4 is addressed by replicating the lists per-CPU, reducing cacheline bouncing and eliminating the need for locks guarding the lists. Instead, the `rc_callback()` function simply enqueues the `rc_callback_t` onto the current CPU's `next` list with interrupts disabled as shown in Figure 31. The declaration on lines 3 and 4 points `rcpl` to this CPU's rclock state. Line 5 marks the callback as registered, and line 6 collects statistics. Finally, lines 7 through 14 adds the callback to the end of the list. The list is processed by code invoked from the clock interrupt handler.

Item 5 is addressed by maintaining the following:

1. per-CPU counters for each quiescent state, including number of context switches, number of passes through the idle loop, number of system calls, and number of traps from user code (different read-copy update semantics, for example, allowing preemption, can be obtained by selecting a different set of quiescent states and tracking tasks rather than CPUs [Bhatt01]).

2. a bitmask that contains a bit for each CPU, which is set if that CPU needs to pass through a quiescent state.

3. a global current-generation and maximum-

```
 1 void rc_callback(rc_callback_t *rc)
 2 {
 3    rc_plocal_t *rcpl =
 4       RC_PLOCAL(smp_processor_id());
 5    rc->flags |= RCC_REGISTERED;
 6    atomic_inc(&rc_ctrlblkd.nreg);
 7    rc->next = NULL;
 8    __cli();
 9    if (rcpl->rclocknxtlist == NULL) {
10       rcpl->rclocknxtlist = rc;
11    } else {
12       *rcpl->rclocknxttail = rc;
13    }
14    __sti();
15 }
```

Figure 31: Fast Read-Copy Callback Registry

generation counter, along with per-CPU generation counters. These generation numbers count grace periods. The per-CPU generation counters indicate which generation the callbacks in the corresponding `current` list belong to.

This data is processed by a state machine that is invoked from the per-CPU timer interrupt.

A new generation is initiated by setting all CPU's bits in the bitmask, by incrementing the global current generation, and by setting the global maximum generation to one greater than the current value.

As each CPU notices that its bit is set, it copies its counters to corresponding "snapshot" counters. Later, when the CPU notices that any of the counters differs from the snapshot, it clears its bit. If its bit is the last one set, it increments the global current generation. If the global current generation does not then exceed the global maximum generation, the CPU initiates a new generation.

As each CPU notices that the global current generation has advanced past its generation number, it appends the contents of its `current` list to its `intr` list, and schedules a tasklet to process it. If the CPU's `next` list is nonempty, the CPU moves it to its `current` list, and sets its per-CPU generation number to be one greater than the global generation number. If a generation is already in progress, the CPU sets the global maximum generation number to be one greater than its per-CPU generation number, otherwise, it starts a new generation.

The `rc_callback()` function simply adds the callback to the current CPU's `next` list. If the CPU's `current` list is empty, then the CPU notices (at the next timer interrupt) that its `next` list is nonempty, it will move the contents of its `next` list to its `current` list and request a new generation, starting one if there is not already one in progress.

For more details, see the patch [Sarma01], its documentation [McK01a], and the paper reporting the performance of the DYNIX/ptx implementation [McK98a]. This design is more complex than the implementations discussed earlier, but is also more flexible and has much better performance.

7 Other Locking Algorithms

This section presents a few locking algorithms, and shows how they relate to variants of read-copy update. These algorithms all search and delete from a circular doubly linked list, and use the struct definition shown in Figure 3. This section is by no means an exhaustive survey of locking algorithms.

7.1 Data Locking

Data locking associates a separate lock with each instance of a given data structures, and can therefore be arbitrarily scalable. Data locking is prone to deadlock, and requires reading tasks to perform expensive writes to shared memory that result in bounced cache lines (for example, lines 4 and 8 in Figure 32). This data locking example thus serves to show how read-copy update can simplify deadlock avoidance (see Section 7.3). In addition, although the list elements may be manipulated in parallel, searches cannot be done in parallel.

Note that there is no natural lock hierarchy: `search()` must acquire the locks in the opposite order from delete (see lines 4 through 10 of Figure 33). One alternative would be to hold `list_lock` upon return from `search()`, but this change would increase the contention on this global lock to the point where there would be no advantage over a global lock. Another alternative would be to unconditionally drop `p->el_lock` and then acquire `list_lock` and `el_lock` in order, but this also increases contention on `list_lock`.

```
 1 struct el *search(long addr, bool keeplock)
 2 {
 3    struct el *p;
 4    spin_lock(&list_lock);
 5    p = head->next;
 6    while (p != head) {
 7       if (p->address == addr) {
 8          spin_lock(&p->lock);
 9          if (!keeplock) {
10            spin_unlock(&list_lock);
11          }
12          return (p);
13       }
14       p = p->next;
15       }
16   spin_unlock(&list_lock);
17   return (NULL);
18 }
```

Figure 32: Data-Locked Search

Although this deadlock-avoidance code is not too complex in this "toy" example, it has doubled in size compared to Figure 9. Not only is data-locked deletion function more difficult to write and inspect, it is also more difficult to test. Lines 5 through 9 are only executed in case of a race with another CPU, and line 8 requires an additional race at the same time.

7.2 Reader-Writer Locking

Ingo Molnar's brlock is an example of a distributed reader-writer lock [Hseih91]. It can be thought of as a cache-aligned array of per-CPU locks, where a reading task acquires only its CPU's lock, and a writing task must acquire all CPUs' locks. The cacheline containing a given CPU's lock is therefore likely to remain in that CPU's cache, so that readers are much less likely to need to bounce cache lines. However, brlock structures are quite large, so that they normally cannot be embedded within data structures. The code structure is very similar to that of reader-writer locking, but attains much higher concurrency in parallel searches as long as reading is much more common than writing.

Figure 34 shows the `search()` code, which has gained some complexity because the `list_lock` might need to be kept in either read or write mode. However, this added complexity allows reading tasks to search the list and to examine individual elements in parallel. Writing tasks are still serialized.

```
 1 void delete(struct el *p, bool keeplock)
 2 {
 3    long addr;
 4    if (!spin_trylock(&list_lock)) {
 5       addr = p->address;
 6       spin_unlock(&p->lock);
 7       if ((p = search(addr, 1)) == NULL) {
 8          return;
 9       }
10    }
11    p->next->prev = p->prev;
12    p->prev->next = p->next;
13    spin_unlock(&p->lock);
14    if (!keeplock) {
15       spin_unlock(&list_lock);
16    }
17    kfree(p);
18 }
```

Figure 33: Data-Locked Deletion

```
 1 struct el *search(long addr, bool write)
 2 {
 3    struct el *p;
 4    if (write) {
 5       br_write_lock(&list_lock);
 6    } else {
 7       br_read_lock(&list_lock);
 8    }
 9    p = head->next;
10    while (p != head) {
11       if (p->address == addr) {
12          return (p);
13       }
14       p = p->next;
15    }
16    if (write) {
17       br_write_unlock(&list_lock);
18    } else {
19       br_read_unlock(&list_lock);
20    }
21    return (NULL);
22 }
```

Figure 34: Reader-Writer Locked Search

```
1 void delete(struct el *p, bool keeplock)
2 {
3    p->next->prev = p->prev;
4    p->prev->next = p->next;
5    if (!keeplock) {
6       br_write_unlock(&list_lock);
7    }
8    kfree(p);
9 }
```

Figure 35: Reader-Writer Locked Deletion

The deletion code (Figure 35) is reasonably simple. The caller must have kept the `list_lock` in write mode, presumably by having invoked `search()` with `write!=0`.

It is possible to combine the ideas of reader-writer locking and data locking, which allows multiple updating tasks to manipulate individual elements concurrently, but this combining adds more complexity.

7.3 Read-Copy Update Without Stale Data

There are a number of ways of suppressing stale data in read-copy update algorithms. One of the simplest approaches is to combine the ideas of data locking with those of read-copy update, using a per-element lock and a `deleted` flag to detect and reject stale data, as shown in Figures 36 and 37. This approach introduces some cacheline bouncing, since the element must be locked in order to safely test the `delete` flag. Although it is also possible to accomplish this test safely with (for example) compare-and-exchange instructions, the cachelines will still bounce. However, only the cachelines containing the elements themselves will bounce, since there is no need for a list lock to guard the search.

Figure 37 shows the corresponding delete function, which adds only the setting of the `delete` flag and the acquisition and release of the per-element lock to the read-copy delete algorithm shown in Figure 9.

7.4 Read-Copy Update Across Context Switches

There are also a number of ways of holding references across context switches. One of the sim-

```
 1 struct el *search(long addr)
 2 {
 3    struct el *p;
 5    p = head->next;
 6    while (p != head) {
 7       if (p->address == addr) {
 8          spin_lock(&p->lock);
 9          if (!p->deleted) {
10             return (p);
11          }
12          spin_unlock(&p->lock);
13       }
14       p = p->next;
15    }
16    return (NULL);
17 }
```

Figure 36: Read-Copy Search Without Stale Data

```
 1 void delete(struct el *p)
 2 {
 3    spin_lock(&list_lock);
 4    spin_lock(&p->lock);
 5    p->next->prev = p->prev;
 6    p->prev->next = p->next;
 7    p->delete = 1;
 8    spin_unlock(&p->lock);
 9    spin_unlock(&list_lock);
10    kfree_rcu(p, NULL);
11 }
```

Figure 37: Read-Copy Deletion Without Stale Data

```
1 void hold(struct el *p)
2 {
3    spin_lock(&p->lock);
4    atomic_inc(&p->refcnt);
5    spin_unlock(&p->lock);
6 }
```

Figure 38: Read-Copy Hold Across Context Switches

```
1 static void release(struct el *p)
2 {
3    spin_lock(&p->lock);
4    if (atomic_dec_and_test(&p->el_refcnt) {
5       kfree_rcu(p, NULL);
6    }
7    spin_unlock(&p->lock);
8 }
```

Figure 39: Read-Copy Release

plest approaches is to combine the ideas of reference counting with those of read-copy update, using a per-element reference counter as shown in Figures 38, 39, and 40. The `search()` algorithm is identical to that shown in Figure 8, but `hold()` must be called if a reference to the element is to be held across a context switch, and a balancing `release()` must be called some time after return from the context switch. This approach again introduces some cacheline bouncing due to `hold()`'s locking and manipulation of `refcnt`, however, `hold()` need only be called when a context switch is encountered. Although it is again also possible to eliminate the locks via compare-and-exchange instructions, the cachelines will still bounce. However, there is no list lock, and hence no bouncing cachelines corresponding to a list lock.

Figure 40 shows the corresponding delete function, which adds only `release` to the read-copy delete algorithm shown in Figure 9.

8 Concluding Remarks

In restricted but commonly occurring situations, read-copy update can significantly reduce complexity while simultaneously improving performance and scaling. It does so by splitting updates into two phases, with an intervening grace period. This form of update greatly simplifies handling races between modifications and concurrent accesses while still maintaining good performance on contemporary hardware.

```
1 void delete(struct el *p)
2 {
3    spin_lock(&list_lock);
4    p->next->prev = p->prev;
5    p->prev->next = p->next;
6    release(p);
7    spin_unlock(&list_lock);
8 }
```

Figure 40: Read-Copy Deletion Across Context Switches

Although read-copy update is new to Linux, it has been in production use within Sequent's DYNIX/ptx kernel since 1993, and was independently developed for K42 and Tornado. DYNIX/ptx is a highly scalable non-preemptive Unix kernel supporting up to 64 CPUs that is primarily used for high-end database servers, and K42 and Tornado are research operating systems that are designed from the ground up to run efficiently on SMP and NUMA systems.

Read-copy update is used as shown below. The most common use of read-copy update is efficient maintenance of linked data structures as described in Sections 2 and 7.

1. Distributed lock manager: recovery, lists of callbacks used to report completions and error conditions to user processes, and lists of server and client lock data structures. This subsystem inspired read-copy update.

2. TCP/IP: routing tables, interface tables, and protocol-control-block lists.

3. Storage-area network (SAN): routing tables and error-injection tables (used for stress testing).

4. Clustered journaling file system: in-core inode lists and distributed-locking data structures.

5. Lock-contention measurement: B* tree used to map from spinlock addresses to the corresponding measurement data (since the spinlocks are only one byte in size, it is not possible to maintain a pointer within each spinlock to the corresponding measurement data).

6. Application regions manager (which is a workload-management subsystem): maintains lists of regions into which processes may be confined.

7. Process management: per-process system-call tables as well as the multi-processor trace data structures used to support user-level debugging of multi-threaded processes.

8. LAN drivers: resolve races between shutting down a LAN device and packets being received by that device.

The Tornado and K42 [Gamsa99] research operating systems independently developed a form of read-copy update, which is used as follows:

1. To provide existence guarantees throughout these operating systems. These existence guarantees simplify handling of races between use of a data structure and its deletion.

2. To identify quiescent states so that implementations of an object can be swapped on the fly while the object is in active use.

The patches described in this paper show that read-copy update is feasible and useful in the Linux kernel. More work is needed to obtain the right balance between simplicity of read-copy update's implementation and its capability. We will apply read-copy update to more areas in the Linux kernel, and measure the resulting effects on performance and complexity.

Linux continues to evolve, and one possible addition to Linux is in-kernel preemption. Although the current read-copy update patches do not handle preemption, there is ongoing work in this area [Russell01b]. Furthermore both Tornado and K42 provide existence proofs that read-copy update is both feasible and useful in preemptive environments.

9 Acknowledgements

We owe thanks to Stuart Friedberg, Doug Miller, Jan-Simon Pendry, Chandrasekhar Pulamarasetti, Jay Vosburgh, Dave Wolfe, Peter Strazdins, and Anton Blanchard for their willingness to try out read-copy update, to Keith Owens for valuable discussions of module unloading, and to Ken Dove, Brent Kingsbury, John Walpole, James Hook, Dylan McNamee, and especially to Andrew Black for many helpful discussions. We are also indebted to Phil Krueger and Hubertus Franke for their careful review of this paper.

10 Availability

Read-copy update is freely available under GPL [Russell01a, Sarma01, Russell01b]. The FD-management, hotplug-CPU, and module-unloading patches that use read-copy update are also freely available under GPL [Soni01b, Russell01c, Soni01a].

More information is available at:

`http://lse.sourceforge.net/locking/rclock.html`
`http://www.rdrop.com/users/paulmck`

11 References

References

[Adams91] G. R. Adams. *Concurrent Programming, Principles, and Practices*, Benjamin Cummins, (1991).

[Bhatt01] S. Bhattacharya *Re: [PATCH for 2.5] preemptible kernel*, http://www.uwsg.indiana.edu/hypermail/linux /kernel/0104.1/0111.html, April 2001.

[Burger96] D. Burger, J. R. Goodman, and A. Kgi. *Memory bandwidth limitations of future microprocessors*, Proceedings of the 23rd International Symposium on Computer Architecture, May, 1996. pp. 78-89,

[Gamsa99] B. Gamsa, O. Kreiger, J. Appavoo, and M. Stumm. *Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system*, Proceedings of the 3rd Symposium on Operating System Design and Implementation, New Orleans, LA, February, 1999.

[Hennes91] J. L. Hennessy and Norman P. Jouppi. *Computer technology and architecture: An evolving interaction*. IEEE Computer, 24(9). pp. 18-28, September 1991.

[Hseih91] W. C. Hseih and W. E. Wiehl, *Scalable Reader-Writer Locks for Parallel Systems*, Tech report MIT/LCS/TR-521, November, 1991

[McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998.

[McK01a] P. E. McKenney. *Read-Copy Mutual Exclusion in Linux*, http://lse.sourceforge.net/locking/rclock_linux.html, February 2001.

[Pugh90] W. Pugh. *Concurrent Maintenance of Skip Lists*, Department of Computer Science, University of Maryland, CS-TR-2222.1, June, 1990.

[Russell01a] R. Russell. *Re: [PATCH for 2.5] preemptible kernel*, http://www.uwsg.indiana.edu/hypermail/linux /kernel/0103.2/0424.html, March 2001.

[Russell01b] R. Russell. *Re: [PATCH for 2.5] preemptible kernel*, http://www.uwsg.indiana.edu/hypermail/linux /kernel/0103.3/1070.html, March 2001.

[Russell01c] R. Russell and A. Blanchard. *[PATCH] Hot swap CPU support for 2.4.1*, http://www.uwsg.indiana.edu/hypermail/linux /kernel/0102.0/0751.html, February 2001.

[Sarma01] D. Sarma. *Read-Copy Update patch for 2.4.2 kernel (Version 01)*, http://lse.sourceforge.net/locking/patches/rclock-2.4.2-01.patch, February 2001.

[Soni01a] M. Soni. *Module unloading using read-copy-update for 2.4.2 kernel (Version 02)*, http://lse.sourceforge.net/locking/patches /module_unloading-2.4.2-0.1.tar.gz, February 2001.

[Soni01b] M. Soni. *Scalable FD management using read-copy-update for 2.4.2 kernel (Version 02)*, http://lse.sourceforge.net/locking/patches /files_struct_rcu-2.4.2-0.2.patch, February 2001.

[Stone91] H. S. Stone and J. Cocke. *Computer architecture in the 1990s*. IEEE Computer, 24(9), pages 30-38, September 1991.