# Overview of Linux-Kernel Reference Counting

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@us.ibm.com

Document number: N2167=07-0027
Date: 2007-01-12
Project: Programming Language C++, Evolution Working Group
Reply-to: paulmck@linux.vnet.ibm.com
Revision: Draft 1

## Abstract

This document describes several reference-counting disciplines used in the Linux kernel, and concludes by summarizing the memory-barrier, atomic-instruction, and compiler-control required by each. This material is adapted from a tutorial document, so the alert reader may notice a few departures from the typical standards-document style.

These disciplines show ample precedent for specifying that a given variable should have atomic response to normal loads and stores, and for the ability to separately specify atomic operations, memory barriers, and disabling of compiler optimizations in uses of variables.

## 1 Introduction

Reference counting tracks the number of references to a given object in order to prevent that object from being prematurely freed. Although this is a conceptually simple technique, there are a great many variants adapted to different situations. This is analogous to similar situations in construction, for example, a hinge is simply a hinge, but there is an astonishing variety of hinges for different purposes. Insisting that programmers use a single style of reference counting is as nonsensical as insisting that all doors and cabinets use a single style of hinge – after all, a hinge designed for a bank vault might be inappropriate for a kitchen cabinet.[1] The examples herein are taken from the Linux 2.6.19 kernel, with primitives described in Section 3.

| Acquisition Synchronization | Release Synchronization | | |
|---|---|---|---|
| | Locking | Reference Counting | RCU |
| Locking | - | CAM | CA |
| Reference Counting | A | AM | A |
| RCU | CA | MCA | CA |

Table 1: Reference Counting and Synchronization Mechanisms

A key reason for the variety of reference-counting techniques is the wide variety of mechanisms used to protect objects from concurrent access. Furthermore, the same object might be protected by different mechanisms at different times, which further increases the required number of styles of reference counting. In the Linux kernel, the main categories of synchronization mechanisms are (1) locking, including semaphores and mutexes, (2) reference counts,

[1]My wife concurs, with the possible exception of kitchen cabinets filled with truffles. Your mileage with your own spouse may vary.

and (3) RCU. Table 1 divides reference-counting mechanisms into the following broad categories:

1. Simple counting with neither atomic operations, memory barriers, nor alignment constraints (“-”).
2. Atomic counting without memory barriers (“A”).
3. Atomic counting, with memory barriers required only on release (“AM”).
4. Atomic counting with a check combined with the atomic acquisition operation, and with memory barriers required only on release (“CAM”).
5. Atomic counting with a check combined with the atomic acquisition operation (“CA”).
6. Atomic counting with a check combined with the atomic acquisition operation, and with memory barriers also required on acquisition (“MCA”).

However, because all Linux-kernel atomic operations that return a value are defined to contain memory barriers, all release operations contain memory barriers, and all checked acquisition operations also contain memory barriers. Therefore, cases “CA” and “MCA” are equivalent to “CAM”. Therefore, there are sections below for only the first four cases: “-”, “A”, “AM”, and “CAM”. The Linux primitives that support reference counting are presented in Section 3. Later sections cite optimizations that can improve performance if reference acquisition and release is very frequent, and the reference count need be checked for zero only very rarely.

# 2 Implementation of Reference-Counting Categories

Simple counting protected by locking (“-”) is described in Section 2.1, atomic counting with no memory barriers (“A”) is described in Section 2.2 atomic counting with acquisition memory barrier (“AM”) is described in Section 2.3, and atomic counting with check and release memory barrier (“CAM”) is described in Section 2.4.

## 2.1 Simple Counting

Simple counting, with neither atomic operations nor memory barriers, can be used when the reference-counter acquisition and release are both protected by the same lock. In this case, it should be clear that the reference count itself may be manipulated non-atomically, because the lock provides any necessary exclusion, memory barriers, atomic instructions, and disabling of compiler optimizations. This is the method of choice when the lock is required to protect other operations in addition to the reference count, but where a reference to the object must be held after the lock is released. Figure 1 shows a simple API that might be used to implement simple non-atomic reference counting – although simple reference counting is almost always open-coded instead.

```
struct sref {
  int refcount;
};

void sref_init(struct sref *sref)
{
  sref->refcount = 1;
}

void sref_get(struct sref *sref)
{
  sref->refcount++;
}

int sref_put(struct sref *sref,
             void (*release)(struct sref *sref))
{
  WARN_ON(release == NULL);
  WARN_ON(release == (void (*)(struct sref *))kfree);

  if (--sref->refcount == 0) {
    release(sref);
    return 1;
  }
  return 0;
}
```

Figure 1: Simple Reference-Count API

## 2.2 Atomic Counting

Simple atomic counting may be used in cases where any CPU acquiring a reference must already hold a reference. This style is used when a single CPU creates an object for its own private use, but must allow other CPU, tasks, timer handlers, or I/O completion handlers that it later spawns to also access this object. Any CPU that hands the object off must first acquire a new reference on behalf of the recipient object. In the Linux kernel, the `kref` primitives are used to implement this style of reference counting, as shown in Figure 2.

Atomic counting is required because locking is not used to protect all reference-count operations, which means that it is possible for two different CPUs to concurrently manipulate the reference count. If normal increment and decrement were used, a pair of CPUs might both fetch the reference count concurrently, perhaps both obtaining the value “3”. If both of them increment their value, they will both obtain “4”, and both will store this value back into the counter. Since the new value of the counter should instead be “5”, one of the two increments has been lost. Therefore, atomic operations must be used both for counter increments and for counter decrements.

If releases are guarded by locking or RCU, memory barriers are *not* required, but for different reasons. In the case of locking, the locks provide any needed memory barriers (and disabling of compiler optimizations), and the locks also prevent a pair of releases from running concurrently. In the case of RCU, cleanup must be deferred until all currently executing RCU read-side critical sections have completed, and any needed memory barriers or disabling of compiler optimizations will be provided by the RCU infrastructure. Therefore, if two CPUs release the final two references concurrently, the actual cleanup will be deferred until both CPUs exit their RCU read-side critical sections.

Quick Quiz 1: Why isn’t it necessary to guard against cases where one CPU acquires a reference just after another CPU releases the last reference?

```
 1 struct kref {
 2   atomic_t refcount;
 3 };
 4
 5 void kref_init(struct kref *kref)
 6 {
 7   atomic_set(&kref->refcount,1);
 8 }
 9
10 void kref_get(struct kref *kref)
11 {
12   WARN_ON(!atomic_read(&kref->refcount));
13   atomic_inc(&kref->refcount);
14 }
15
16 int kref_put(struct kref *kref,
17              void (*release)(struct kref *kref))
18 {
19   WARN_ON(release == NULL);
20   WARN_ON(release == (void (*)(struct kref *))kfree);
21
22   if ((atomic_read(&kref->refcount) == 1) ||
23       (atomic_dec_and_test(&kref->refcount))) {
24     release(kref);
25     return 1;
26   }
27   return 0;
28 }
```

Figure 2: Linux Kernel kref API

The `kref` structure itself, consisting of a single atomic data item, is shown in lines 1-3 of Figure 2. The `kref_init()` function on lines 5-8 initializes the counter to the value “1”. Note that the `atomic_set()` primitive is a simple assignment, the name stems from the data type of `atomic_t` rather than from the operation. The `kref_init()` function must be invoked during object creation, before the object has been made available to any other CPU.

The `kref_get()` function on lines 10-14 unconditionally atomically increments the counter. The `atomic_inc()` primitive does not necessarily explicitly disable compiler optimizations on all platforms, but the fact that the `kref` primitives are in a separate module and that the Linux kernel build process does no cross-module optimizations has the same effect.

The `kref_put()` function on lines 16-28 checks for the counter having the value “1” on line 22 (in which case no concurrent `kref_get()` is permitted), or if atomically decrementing the counter results in zero on line 23. In either of these two cases, `kref_put()` invokes the specified `release` function and returns “1”, telling the caller that cleanup was performed. Otherwise, `kref_put()` returns “0”.

Quick Quiz 2: If the check on line 22 of Figure 2 fails, how could the check on line 23 possibly succeed?

Quick Quiz 3: How can it possibly be safe to non-atomically check for equality with "1" on line 22 of Figure 2?

## 2.3 Atomic Counting With Release Memory Barrier

This style of reference is used in the Linux kernel's networking layer to track the destination caches that are used in packet routing. The actual implementation is quite a bit more involved; this section focuses on the aspects of `struct dst_entry` reference-count handling that matches this use case, shown in Figure 3.

```
 1 static inline
 2 struct dst_entry * dst_clone(struct dst_entry * dst)
 3 {
 4   if (dst)
 5     atomic_inc(&dst->__refcnt);
 6   return dst;
 7 }
 8
 9 static inline
10 void dst_release(struct dst_entry * dst)
11 {
12   if (dst) {
13     WARN_ON(atomic_read(&dst->__refcnt) < 1);
14     smp_mb__before_atomic_dec();
15     atomic_dec(&dst->__refcnt);
16   }
17 }
```

Figure 3: Linux Kernel dst_clone API

The `dst_clone()` primitive may be used if the caller already has a reference to the specified `dst_entry`, in which case it obtains another reference that may be handed off to some other entity within the kernel. Because a reference is already held by the caller, `dst_clone()` need not execute any memory barriers. The act of handing the `dst_entry` to some other entity might or might not require a memory barrier, but if such a memory barrier is required, it will be embedded in the mechanism used to hand the `dst_entry` off.

The `dst_release()` primitive may be invoked from any environment, and the caller might well reference elements of the `dst_entry` structure immediately prior to the call to `dst_release()`. The `dst_release()` primitive therefore contains a memory barrier on line 14 preventing both the compiler and the CPU from misordering accesses.

Please note that the programmer making use of `dst_clone()` and `dst_release()` need not be aware of the memory barriers, only of the rules for using these two primitives.

## 2.4 Atomic Counting With Check and Release Memory Barrier

The fact that reference-count acquisition can run concurrently with reference-count release adds further complications. Suppose that a reference-count release finds that the new value of the reference count is zero, signalling that it is now safe to clean up the reference-counted object. We clearly cannot allow a reference-count acquisition to start after such clean-up has commenced, so the acquisition must include a check for a zero reference count. This check must be part of the atomic increment operation, as shown below.

Quick Quiz 4: Why can't the check for a zero reference count be made in a simple "if" statement with an atomic increment in its "then" clause?

The Linux kernel's `fget()` and `fput()` primitives use this style of reference counting. Simplified versions of these functions are shown in Figure 4.

Line 4 of `fget()` fetches the a pointer to the current process's file-descriptor table, which might well be shared with other processes. Line 6 invokes `rcu_read_lock()`, which enters an RCU read-side critical section. The callback function from any subsequent `call_rcu()` primitive will be deferred until a matching `rcu_read_unlock()` is reached (line 10 or 14 in this example). Line 7 looks up the file structure corresponding to the file descriptor specified by the `fd` argument, as will be described later. If there is an open file corresponding to the specified file descriptor, then line 9 attempts to atomically acquire a reference count. If it fails to do so, lines 10-11 exit the RCU read-side critical section and report failure. Otherwise, if the attempt is successful, lines 14-15 exit the read-side critical section and return a pointer to the file structure.

```
 1 struct file *fget(unsigned int fd)
 2 {
 3   struct file *file;
 4   struct files_struct *files = current->files;
 5
 6   rcu_read_lock();
 7   file = fcheck_files(files, fd);
 8   if (file) {
 9     if (!atomic_inc_not_zero(&file->f_count)) {
10       rcu_read_unlock();
11       return NULL;
12     }
13   }
14   rcu_read_unlock();
15   return file;
16 }
17
18 struct file *
19 fcheck_files(struct files_struct *files, unsigned int fd)
20 {
21   struct file * file = NULL;
22   struct fdtable *fdt = rcu_dereference((files)->fdt);
23
24   if (fd < fdt->max_fds)
25     file = rcu_dereference(fdt->fd[fd]);
26   return file;
27 }
28
29 void fput(struct file *file)
30 {
31   if (atomic_dec_and_test(&file->f_count))
32     call_rcu(&file->f_u.fu_rcuhead, file_free_rcu);
33 }
34
35 static void file_free_rcu(struct rcu_head *head)
36 {
37   struct file *f;
38
39   f = container_of(head, struct file, f_u.fu_rcuhead);
40   kmem_cache_free(filp_cachep, f);
41 }
```

Figure 4: Linux Kernel fget/fput API

The `fcheck_files()` primitive is a helper function for `fget()`. It uses the `rcu_dereference()` primitive to safely fetch an RCU-protected pointer for later dereferencing (this emits a memory barrier on CPUs such as DEC Alpha in which data dependencies do not enforce memory ordering). Line 22 uses `rcu_dereference()` to fetch a pointer to this task's current file-descriptor table, and line 24 checks to see if the specified file descriptor is in range. If so, line 25 fetches the pointer to the file structure, again using the `rcu_dereference()` primitive. Line 26 then returns a pointer to the file structure or `NULL` in case of failure.

The `fput()` primitive releases a reference to a file structure. Line 31 atomically decrements the reference count, and, if the result was zero, line 32 invokes the `call_rcu()` primitives in order to free up the file structure (via the `file_free_rcu()` function specified in `call_rcu()`'s second argument), but only after all currently-executing RCU read-side critical sections complete. The time period required for all currently-executing RCU read-side critical sections to complete is termed a "grace period". Note that the `atomic_dec_and_test()` primitive contains a memory barrier. This memory barrier is not necessary in this example, since the structure cannot be destroyed until the RCU read-side critical section completes, but in Linux, all atomic operations that return a result must by definition contain memory barriers.

Once the grace period completes, the `file_free_rcu()` function obtains a pointer to the file structure on line 39, and frees it on line 40.

This approach is also used by Linux's virtual-memory system, see `get_page_unless_zero()` and `put_page_testzero()` for page structures as well as `try_to_unuse()` and `mmput()` for memory-map structures.

## 3 Linux Primitives Supporting Reference Counting

The Linux-kernel primitives used in the above examples are summarized in the following list. The RCU primitives may be unfamiliar to some readers, so a brief overview with citations is included in Section 5.

- `atomic_t` Type definition for 32-bit quantity to be manipulated atomically.
- `void atomic_dec(atomic_t *var);` Atomically decrements the referenced variable without necessarily issuing a memory barrier or disabling compiler optimizations.
- `int atomic_dec_and_test(atomic_t *var);` Atomically decrements the referenced variable, returning TRUE if the result is zero. Issues a memory barrier and disables compiler optimizations that might otherwise move memory references across this primitive.
- `void atomic_inc(atomic_t *var);` Atomically increments the referenced variable without necessarily issuing a memory barrier or disabling compiler optimizations.
- `int atomic_inc_not_zero(atomic_t *var);` Atomically increments the referenced variable, but only if the value is non-zero, and returning TRUE if the increment occurred. Issues a memory barrier and disables compiler optimizations that might otherwise move memory references across this primitive.
- `int atomic_read(atomic_t *var);` Returns the integer value of the referenced variable. This is not an atomic operation, and it neither issues memory barriers nor disables compiler optimizations.
- `void atomic_set(atomic_t *var, int val);` Sets the value of the referenced atomic variable to "val". This is not an atomic operation, and it neither issues memory barriers nor disables compiler optimizations.
- `void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head));` Invokes `func(head)` some time after all currently executing RCU read-side critical sections complete, however, the `call_rcu()` primitive returns immediately. Note that `head` is normally a field within an RCU-protected data structure, and that `func` is normally a function that frees up this data structure. The time interval between the invocation of `call_rcu()` and the invocation of `func` is termed a "grace period". Any interval of time containing a grace period is itself a grace period.
- `type *container_of(p, type, f);` Given a pointer "p" to a field "f" within a structure of the specified type, return a pointer to the structure.
- `void rcu_read_lock(void);` Marks the beginning of an RCU read-side critical section.
- `void rcu_read_unlock(void);` Marks the end of an RCU read-side critical section. RCU read-side critical sections may be nested.
- `void smp_mb__before_atomic_dec(void);` Issues a memory barrier and disables code-motion compiler optimizations only if the platform's `atomic_dec()` primitive does not already do so.
- `struct rcu_head` A data structure used by the RCU infrastructure to track objects awaiting a grace period. This is normally included as a field within an RCU-protected data structure.

## 4 Counter Optimizations

In some cases where increments and decrements are common, but checks for zero are rare, it makes sense to maintain per-CPU or per-task counter. See the paper on sleepable read-copy update (SRCU) for an example of this technique [7]. This approach eliminates the need for atomic instructions or memory barriers on the increment and decrement primitives, but still requires that code-motion compiler optimizations be disabled. In addition, the primitives such as `synchronize_srcu()` that check for the aggregate reference count reaching zero can be quite slow. This underscores the fact that these techniques are designed for situations where the references are frequently acquired and released, but where it is rarely necessary to check for a zero reference count.

## 5 Background on RCU

Read-copy update (RCU) is a synchronization API that is sometimes used in place of reader-writer locks. RCU's read-side primitives offer extremely low overhead and deterministic execution time, in fact, non-realtime server-class implementations of RCU implement the `rcu_read_lock()` and `rcu_read_unlock()` primitives as empty C-preprocessor macros, so that neither the compiler nor the CPU see any RCU read-side overhead whatsoever.[2] These properties imply that RCU updaters cannot block RCU readers, which means that RCU updaters can be expensive, as they must leave old versions of the data structure in place to accommodate pre-existing readers. Furthermore, these old versions must be reclaimed after all pre-existing readers complete. The Linux kernel offers a number of RCU implementations, the first such implementation being called "Classic RCU".

A full description of RCU is beyond the scope of this document, however, McKenney's dissertation [3] contains an extensive RCU bibliography as well as detailed descriptions of early Linux implementation and corresponding APIs, along with early Linux RCU usage and corresponding performance results. The Wikipedia RCU page [8] provides a short overview of RCU's operation, also presenting "toy" implementations of RCU infrastructure. Numerous publications compare RCU performance to a number of alternative synchronization mechanisms within the Linux kernel [1, 4, 9], and Hart et al. [2] compare various locking and non-blocking-synchronization schemes to an RCU-infrastructure implementation similar to Linux's "Classic RCU". Of course, the ultimate description of RCU in Linux is the source code [10], which makes significant use of the RCU API [5]. McKenney maintains a list of cscope databases listing RCU API usage on a per-Linux-version basis [6].

[2]The DEC Alpha CPU being the one exception proving this rule.

## 6 Operations Required for Reference Counting

Properties of atomic variables and operations on those variables may be categorized as follows:

1. Atomic response to normal loads and stores.
2. Atomic operations.
3. Memory barriers.
4. Disabling of compiler optimizations that move memory references.

Table 2 shows which of the reference-counting methods described in this paper require which of the above properties. This table shows that while memory barriers can imply disabling of compiler code-motion optimizations without penalty, tying atomic operations to memory barriers may penalize the simple atomic-counting reference-counting scheme described in Section 2.2.

Even more severe penalties are imposed on SRCU [7] if optimization disabling is not provided independently, for example, as provided by the gcc compiler's "memory" attribute on `__asm__` directives. In some cases, the C-language "volatile" keyword can be used, however, the semantics of this keyword are ill-defined. In particular, for some architectures, some compilers will emit expensive memory-barrier instructions that are not needed for SRCU.

Finally, SRCU requires that normal loads and stores to its per-CPU counter variables be atomic, in other words, loads must see either the initial value or the complete result of one preceding store. This ability is provided natively for properly aligned C/C++ integers and pointers on all mainstream CPUs, and should be exposed in any C/C++ specification that touches on multithreaded behavior.

In summary, there is ample precedent for specifying that a given variable should have atomic response to normal loads and stores, and for the ability to separately specify atomic operations, memory barriers, and disabling of compiler optimizations in uses of variables.

| Property | Implementation Type | | | | |
|---|---|---|---|---|---|
| | - | A | AM | CAM | SRCU |
| Atomic Response to Normal Loads/Stores | | | | | Y |
| Atomic Operations | | Y | Y | Y | |
| Memory Barriers | | | Y | Y | |
| Disable Compiler Optimizations | | | Y | Y | Y |

Table 2: Reference Counting and Required Operations

## 7 Answers to Quick Quizzes

**Quick Quiz 1:** Why isn't it necessary to guard against cases where one CPU acquires a reference just after another CPU releases the last reference?
**Answer:** Because a CPU must already hold a reference in order to legally acquire another reference. Therefore, if one CPU releases the last reference, there cannot possibly be any CPU that is permitted to acquire a new reference. This same fact allows the non-atomic check in line 22 of Figure 2.

**Quick Quiz 2:** If the check on line 22 of Figure 2 fails, how could the check on line 23 possibly succeed?
**Answer:** Suppose that `kref_put()` is protected by RCU, so that two CPUs might be executing line 22 concurrently. Both might see the value "2", causing both to then execute line 23. One of the two instances of `atomic_dec_and_test()` will decrement the value to zero and thus return 1.

**Quick Quiz 3:** How can it possibly be safe to non-atomically check for equality with "1" on line 22 of Figure 2?
**Answer:** Remember that it is not legal to call either `kref_get()` or `kref_put()` unless you hold a reference. If the reference count is equal to "1", then there can't possibly be another CPU authorized to change the value of the reference count.

**Quick Quiz 4:** Why can't the check for a zero reference count be made in a simple "if" statement with an atomic increment in its "then" clause?
**Answer:** Suppose that the "if" condition completed, finding the reference counter value equal to one. Suppose that a release operation executes, decrementing the reference counter to zero and therefore starting cleanup operations. But now the "then" clause can increment the counter back to a value of one, allowing the object to be used after it has been cleaned up.

## Acknowledgements

I owe thanks to Michael Wong for his careful review of this paper, and am indebted to Daniel Frye for his support of this effort.

## References

[1] Arcangeli, A., Cao, M., McKenney, P. E., and Sarma, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310.

[2] Hart, T. E., McKenney, P. E., and Brown, A. D. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th IEEE International Parallel and Distributed Processing Symposium* (Rhodes, Greece, April 2006).

[3] McKenney, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[4] McKenney, P. E. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, Australia, January 2004). Available: `http://www.linux.`

org.au/conf/2004/abstracts.html#90 http://www.rdrop.com/users/paulmck/RCU/lockperf.2004.01.17a.pdf [Viewed June 23, 2004].

[5] McKenney, P. E. RCU Linux usage. Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html [Viewed January 14, 2007], October 2006.

[6] McKenney, P. E. Read-copy update (RCU) usage in Linux kernel. Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage/rculocktab.html [Viewed January 14, 2007], October 2006.

[7] McKenney, P. E. Sleepable RCU. Available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf [Viewed August 21, 2006], October 2006.

[8] McKenney, P. E., Purcell, C., Algae, Schumin, B., Cornelius, G., Qwertyus, Conway, N., Sbw, Blainster, Rufus, C., Zoicon5, Anome, and Eisen, H. Read-copy update. Available: http://en.wikipedia.org/wiki/Read-copy-update [Viewed August 21, 2006], July 2006.

[9] Morris, J. Recent developments in SELinux kernel performance. Available: http://www.livejournal.com/users/james_morris/2153.html [Viewed December 10, 2004], December 2004.

[10] Torvalds, L. Linux 2.6. Available: ftp://kernel.org/pub/linux/kernel/v2.6 [Viewed June 23, 2004], August 2003.