Linux-Kernel RCU Shared-Variable Marking

March 13, 2024
May 7, 2024

Read-copy update (RCU) has stricter shared-variable marking guidelines than does the rest of the kernel. This is due to RCU's relatively complexity and highly concurrent design, and the consequent desire to get all the help that is available from tools such as KCSAN.

Oddly enough, these strict guidelines are also appropriate for simple code, for example, that does strict locking. In such cases, any data race at all is a bug, for example, a failure to acquire the proper lock.

This document does not cover the recently proposed `__data_racy` marking that can be applied to a given data object. This marking indicates that all accesses to that object should be treated as if they had been passed through KCSAN's `data_race()` API member. Alternatively, hardened builds of the Linux kernel could treat such objects as if they were `volatile`.

See also the `tools/memory-model/Documentation/access-marking.txt` file in the Linux-kernel source tree.

RCU Shared-Variable Marking Rules

In RCU, the key point is not whether the compiler can or cannot introduce any destructive optimizations, nor is it to minimize the size of the source code. The key point is instead to permit some of the concurrency design to be baked into the source code so as to allow KCSAN to check whether the code matches this design.

Plain C-Language Reads and Writes

Within RCU, use plain C-language reads and writes in code where a data race indicates a concurrency design bug. Thus, use of a C-language read indicates that the variable being read is guaranteed not to be concurrently updated (even to the same value and even by an interrupt handler) at the time of that read, though it might be concurrently read. For example, if all updates to that variable are protected by an exclusive lock that is held at the time of the read, that read can be a plain C-language read. This arrangement enables KCSAN to report lockless (and thus buggy) updates to that variable.

Similarly, use of a C-language write indicates that the variable being written is guaranteed not to be concurrently accessed (at all!) at the time of that write. For example, if all accesses to that variable are protected by an exclusive lock that is held at the time of the write, that write can be a plain C-language write. This arrangement enables KCSAN to report lockless (and thus buggy) plain C-language reads from and writes to that variable. Obtaining the same effect in non-RCU code using non-strict KCSAN rules requires use of `ASSERT_EXCLUSIVE_ACCESS()`, for example, as shown below:

```
x = 1;
ASSERT_EXCLUSIVE_ACCESS(x);
```

Note that for volatile variables such as `jiffies`, a plain C-language read or write is implicitly a `READ_ONCE()` or `WRITE_ONCE()`, respectively.

This rule clearly documents which accesses are subject to data races, making RCU's concurrency design more clear both to human developers and to KCSAN.

READ_ONCE()

Within RCU code, use `READ_ONCE()` in code where the variable being read might be concurrently updated by an appropriately marked write. For example, if the concurrency design allows lockless reads of a variable, then use of `READ_ONCE()` will prevent KCSAN from complaining about that read even when that read executes concurrently with appropriately marked accesses (either read or write). Note that `READ_ONCE()` also prevents load fusing (which can result in infinite loops) and invented loads (which can confuse code expecting to use a consistent value).

The `atomic_read()` family of primitives plays the same role in RCU as does `READ_ONCE()`.

In code where a variable is guaranteed not to be concurrently updated, a plain C-language read should be used instead.

WRITE_ONCE()

Within RCU code, use `WRITE_ONCE()` in code where the variable being written might be either concurrently read or concurrently updated. For example, if the concurrency design permits lockless updates, then use of will prevent KCSAN from complaining about that write even when that write executes concurrently with appropriately marked accesses (either read or write). Note that `WRITE_ONCE()` also prevents invention and fusing of stores.

The `atomic_set()` family of primitives plays the same role in RCU as does `WRITE_ONCE()`.

Of course, one of the objections to RCU's KCSAN configuration is that additional writes must be protected by `WRITE_ONCE()` or similar. On the other hand, RCU's configuration allows plain C-language writes to mean something different than `WRITE_ONCE()`, increasing expressiveness.

As with `READ_ONCE()`, in code where a variable is guaranteed not to be currently accessed, a plain C-language write should be used instead.

Read-Modify-Write Atomic Operations

The Linux kernel's read-modify-write atomic operations are their own markings. As such, they inform both the developer and KCSAN of the possibility of concurrent access to the atomically manipulated variable.

As with `READ_ONCE()` and `WRITE_ONCE()`, it is considered poor form to use an atomic read-modify-write operation in code where concurrent access is not possible.

`data_race()` and `__no_kcsan`

The KCSAN `data_race()` primitive causes KCSAN to ignore any data races involving that `data_race()` instance's argument. This can be helpful in cases where there are no data races except for those that might be produced by diagnostic output. The diagnostic code can enclose its accesses within `data_race()` to prevent KCSAN from reporting data races involving those accesses.

For example, consider a variable that is to be accessed only under a lock, aside from some debugging/statistical accesses. Under RCU's KCSAN rules, marking those debugging/statistical accesses with `READ_ONCE()` would require all the updates to be marked with `WRITE_ONCE()`. This would prevent KCSAN from noticing a buggy lockless `WRITE_ONCE()` update of that variable.

In contrast, if `data_race()` is used for the debugging/statistical accesses and the normal lock-protected accesses are left unmarked (as normal C-language accesses), then KCSAN will complain about any buggy lockless accesses, even if they are marked with `READ_ONCE()` or `WRITE_ONCE()`.

Although an entire diagnostic function can be declared off-limits for data-race reporting by adding `__no_kcsan` to that function's return-value type, this is discouraged in RCU code, as is Makefile-based KCSAN disabling. In both cases, it is all too easy for an access that is neither for debugging or for statistics gathering to slip into that function or that file, which impairs KCSAN's ability to find bugs.

ASSERT_EXCLUSIVE_WRITER()

Within RCU code, use `ASSERT_EXCLUSIVE_WRITER()` to tell KCSAN that a variable updated using `WRITE_ONCE()` should not be subject to any concurrent updates. However, unlike a plain C-language write, KCSAN will not complain about concurrent reads.

The `ASSERT_EXCLUSIVE_WRITER_SCOPED()` macro may be used to cover a scope, and perhaps RCU should introduce scopes enclusing lock-based critical sections in order to make use of this capability.

ASSERT_EXCLUSIVE_ACCESS()

Within RCU code, use `ASSERT_EXCLUSIVE_ACCESS()` to tell KCSAN that there should be no concurrent accesses to the variable. This is different from a plain C-language write (which would otherwise achieve the same thing) in that there is no actual access to the variable, just the KCSAN check.

KCSAN Marking Comparison

This section compares RCU and non-RCU markings. For reference, the KCSAN Settings section lists rcutorture's KCSAN Kconfig options. Again, the point of RCU's marking rules is not to minimize source-code size, but instead to get the most out of KCSAN.

One caution throughout all of this is that there really have been compilers that are happy to tear non-atomic non-volatile stores of certain constants. This is not a problem for CPU architectures featuring full-sized immediates, but last I knew, not all architectures had full-sized immediates.

Exclusive Accesses

This section covers cases where the access in question is intended to be the only access to the object in question. This exclusivity might be enforced by a lock or by the fact that no other entity has access to the object in question.

Reads

The enforcement is as follows:

Rest of Kernel	RCU
`r1 = a;` `ASSERT_EXCLUSIVE_ACCESS(a);`	`r1 = a;` `ASSERT_EXCLUSIVE_ACCESS(a);`

In both cases, a normal C-language read is coupled with a call to the KCSAN `ASSERT_EXCLUSIVE_ACCESS()` function.

Writes

The enforcement is as follows:

Rest of Kernel	RCU
`a = 1;` `ASSERT_EXCLUSIVE_ACCESS(a);`	`a = 1;`

In the rest-of-kernel case, the fact that normal C-language writes are treated the same as is `WRITE_ONCE()` means that a call to the KCSAN `ASSERT_EXCLUSIVE_ACCESS()` function is required. In the RCU case, a normal C-language write suffices.

Atomics

Use of `atomic_t` and friends mean that there is no reasonable normal C-language access. The enforcement is thus as follows:

Rest of Kernel	RCU
`RMW(&a);` `ASSERT_EXCLUSIVE_ACCESS(a);`	`RMW(&a);` `ASSERT_EXCLUSIVE_ACCESS(a);`

In both cases, the desired read-modify-write function is coupled with a call to the KCSAN `ASSERT_EXCLUSIVE_ACCESS()` function.

Concurrent Reads, Excluded Writes

This section covers cases where the access in question is intended to be free of any (other) writes to the object in question. This exclusivity might be enforced by a lock or a reader-writer lock. Alternatively, the object might be updated while holding a lock, but read locklessly.

Note that the rest-of-kernel KCSAN settings will forgive a data race with a write when that write happens to write the same value. For example, if the value of x is already 5, then KCSAN will refrain from complaining when there is a write of that same value of 5. In contrast, the RCU settings cause KCSAN to complain in this same-value-written case.

Reads

This case permits concurrent reads to the object, but forbids concurrent writes, perhaps because readers read-hold a lock that writers must write-hold. The enforcement is as follows:

Rest of Kernel	RCU
`r1 = a;`	`r1 = a;`

In both cases, a normal C-language read suffices.

Writes

This case allows a single update to run concurrently with reads, perhaps because updates are carried out holding an exclusive lock in the presence of lockless readers. The enforcement is as follows:

Rest of Kernel	RCU
`a = 1;` `ASSERT_EXCLUSIVE_WRITER(a);`	`WRITE_ONCE(a, 1);` `ASSERT_EXCLUSIVE_WRITER(a);`

Both cases use `ASSERT_EXCLUSIVE_WRITER()` to cause KCSAN to forbid concurrent writers. In the rest-of-kernel case, the fact that normal C-language writes are treated (by KCSAN) the same as is `WRITE_ONCE()` means that a normal C-language write suffices. In the RCU case, `READ_ONCE()` is used, which is more typing but also better documentation.

Atomics

Use of `atomic_t` and friends mean that there is no reasonable normal C-language access. The enforcement is thus as follows:

Rest of Kernel	RCU
`RMW(&a);` `ASSERT_EXCLUSIVE_WRITER(a);`	`RMW(&a);` `ASSERT_EXCLUSIVE_WRITER(a);`

In both cases, the desired read-modify-write function is coupled with a call to the KCSAN `ASSERT_EXCLUSIVE_WRITER()` function, which causes KCSAN to warn if there are concurrent writers.

Concurrent Accesses

This section covers cases where the access in question might race with any other access to that same object.

Reads

This case permits concurrent reads to the object, but forbids concurrent writes, perhaps because readers read-hold a lock that writers must write-hold. The enforcement is as follows:

Rest of Kernel	RCU
`READ_ONCE(r1, a);`	`READ_ONCE(r1, a);`

In both cases, the `READ_ONCE()` macro is used.

Writes

This case allows a single update to run concurrently with reads, perhaps because updates are carried out holding an exclusive lock in the presence of lockless readers. The enforcement is as follows:

Rest of Kernel	RCU
`a = 1;`	`WRITE_ONCE(a, 1);`

In the rest-of-kernel case, the fact that normal C-language writes are treated the same as `WRITE_ONCE()` means that a normal C-language write suffices. In the RCU case, `WRITE_ONCE()` is used, which is more typing but also better documentation.

Note that the KCSAN configuration used for RCU will detect data races in which the update did not actually change the value of the variable. In contrast, the configuration used in the rest of the kernel will fail to detect such data races.

However, one issue with the use of plain C-language assignments is that architectures that do not support one-byte or two-byte stores cannot interoperate efficiently with other stores or atomic operations to the same shared variable: Doing so requires that all one-byte or two-byte stores be implemented using larger-sized read-modify-write atomic operations.

This might not be a long-term issue given that most (and perhaps all) of the systems lacking hardware support for one-byte and two-byte load and store instructions are likely to be on their way out of the Linux kernel. Nevertheless, the documentation benefits of that explicit WRITE_ONCE() should not be understated.

Atomics

Use of `atomic_t` and friends mean that concurrent access is expected behavior. The enforcement is thus as follows:

Rest of Kernel	RCU
`RMW(&a);`	`RMW(&a);`

In both cases, the desired read-modify-write function suffices.

Interrupt Handlers

Portions of RCU's grace-period execute within the scheduling-clock interrupt handler and other portions execute in the `RCU_SOFTIRQ` handler, and RCU therefore benefits from enabling KCSAN to check data-racy accesses from interrupt handlers. In contrast, code that runs primarily in process context need not care, and the rest-of-kernel KCSAN settings do not check for data races with interrupt handlers.

Note well that use of normal C-language reads at process level combined with any sort of update within an interrupt handler can cause confusion if the compiler invents loads at process level, for example, if the compiler runs out of registers and therefore re-loads the variable shared with that interrupt handler.

Rest of Kernel	RCU
(No facility to detect data races with interrupt handlers.)	Marked and unmarked accesses as shown above.

It is possible that code will appear that allows data races with interrupt handlers but not with other CPUs. Such code might motivate extensions to KCSAN.

RCU KCSAN Settings

The rcutorture scripting enables a number of additional KCSAN Kconfig options when the `--kcsan` argument is selected:

- `CONFIG_KCSAN_STRICT=y` implies:
  - `CONFIG_KCSAN_INTERRUPT_WATCHER=y`, to check for data races between mainline code and interrupt handlers. This is important for RCU because portions of its grace-period state machine run in the scheduling-clock interrupt handler.
  - `CONFIG_KCSAN_WEAK_MEMORY=y`, to check for data races induced by missing memory barriers. (At the expense of less-effective detection of data races due to write reordering, so those running rcutorture on ARM might wish to do at least some runs disabling this Kconfig option.)

    - CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY=n, which checks for data races involving writes that happen to set the value of the variable to the value that it already had. This can be argued to be the right choice in general, but is especially important here because RCU's concurrency design does not rely on same-value writes.
    - CONFIG_KCSAN_ASSUME_PLAIN_WRITES_ATOMIC=n, which does not treat plain C-language writes as if they were instead WRITE_ONCE() invocations. This choice improves KCSAN's ability to detect situations where developers incorrectly assume that a given variable was not being either read or written by some other CPU at the time of the write.
    - CONFIG_KCSAN_IGNORE_ATOMICS=n, which is the normal choice.
- CONFIG_KCSAN_REPORT_ONCE_IN_MS=100000, which tells KCSAN to report a given data race at most one time per second. It might be worth further increasing this time interval.
- CONFIG_KCSAN_VERBOSE=y, which adds held locks and interrupts.

As noted earlier, these settings are also appropriate for naive concurrent code, in which any data race at all is a bug. One example of such code is that protected by pure locking, with no lockless accesses. Additional use cases are called out in the next section.

It might be worth experimenting with additional KCSAN Kconfig options:

- CONFIG_KCSAN_NUM_WATCHPOINTS defaults to 64, and might be worth setting higher when running on multi-socket systems.
- CONFIG_KCSAN_UDELAY_TASK defaults to 80 microseconds, and might be worth some experimentation.
- CONFIG_KCSAN_UDELAY_INTERRUPT defaults to 20 microseconds, and might also be worth some experimentation.
- CONFIG_KCSAN_DELAY_RANDOMIZE provides one way to experiment with the previous pair of delays, providing randomly chosen delays up to the specified limits.

Use Cases for Aggressive KCSAN Settings

As noted above, aggressive KCSAN settings make sense for highly concurrent subsystems such as RCU, but also for very simple use cases. Such use cases include:

- Code using strict locking. In this case, all uses of each shared variable must occur while the corresponding lock is held, so data races indicate a bug in which the developer failed to acquire the required lock.
- Code using lockless queues. In this case, the element is initialized, placed on the queue, removed from the queue, processed, and then freed or otherwise reused. The producer's accesses to the queue element must precede those of the consumer, whose

accesses must in turn precede those of the next producer. Data races indicate a bug, perhaps in the implementation of the queue itself or in the use of the queue's API.

- Code using search structures, for example, those in which data in the structure remains constant. Data races again indicate a bug, for example, cleaning up a data element before having removed it from the search structure.
- Code using fork-join patterns. Here the data is initialized, child processes use that data, and the data is cleaned up after the join. Data races indicate a bug, perhaps due data being improperly used by the child processes.