Runtime locking correctness validator

started by Ingo Molnar <mingo@redhat.com>
additions by Arjan van de Ven <arjan@linux.intel.com>

Lock-class

The basic object the validator operates upon is a 'class' of locks.

A class of locks is a group of locks that are logically the same with respect to locking rules, even if the locks may have multiple (possibly tens of thousands of) instantiations. For example a lock in the inode struct is one class, while each inode has its own instantiation of that lock class.

The validator tracks the 'state' of lock-classes, and it tracks dependencies between different lock-classes. The validator maintains a rolling proof that the state and the dependencies are correct.

Unlike an lock instantiation, the lock-class itself never goes away: when a lock-class is used for the first time after bootup it gets registered, and all subsequent uses of that lock-class will be attached to this lock-class.

State

The validator tracks lock-class usage history into 4n + 1 separate state bits:

- 'ever held in STATE context'
- 'ever held as readlock in STATE context'
- 'ever held with STATE enabled'
- 'ever held as readlock with STATE enabled'

Where STATE can be either one of (kernel/lockdep_states.h)
 - hardirq
 - softirq
 - reclaim_fs

- 'ever used' [== !unused]

When locking rules are violated, these state bits are presented in the locking error messages, inside curlies. A contrived example:

```
   modprobe/2287 is trying to acquire lock:
    (&sio_locks[i].lock){-.-...}, at: [<c02867fd>] mutex_lock+0x21/0x24

   but task is already holding lock:
    (&sio_locks[i].lock){-.-...}, at: [<c02867fd>] mutex_lock+0x21/0x24
```

The bit position indicates STATE, STATE-read, for each of the states listed above, and the character displayed in each indicates:

```
   '.'  acquired while irqs disabled and not in irq context
   '-'  acquired in irq context
   '+'  acquired with irqs enabled
   '?'  acquired in irq context with irqs enabled.
```

Unused mutexes cannot be part of the cause of an error.

Single-lock state rules:

A softirq-unsafe lock-class is automatically hardirq-unsafe as well. The following states are exclusive, and only one of them is allowed to be set for any lock-class:

 <hardirq-safe> and <hardirq-unsafe>
 <softirq-safe> and <softirq-unsafe>

The validator detects and reports lock usage that violate these single-lock state rules.

Multi-lock dependency rules:

The same lock-class must not be acquired twice, because this could lead to lock recursion deadlocks.

Furthermore, two locks may not be taken in different order:

 <L1> -> <L2>
 <L2> -> <L1>

because this could lead to lock inversion deadlocks. (The validator finds such dependencies in arbitrary complexity, i.e. there can be any other locking sequence between the acquire-lock operations, the validator will still track all dependencies between locks.)

Furthermore, the following usage based lock dependencies are not allowed between any two lock-classes:

```
   <hardirq-safe>   ->  <hardirq-unsafe>
   <softirq-safe>   ->  <softirq-unsafe>
```

The first rule comes from the fact the a hardirq-safe lock could be taken by a hardirq context, interrupting a hardirq-unsafe lock - and thus could result in a lock inversion deadlock. Likewise, a softirq-safe lock could be taken by an softirq context, interrupting a softirq-unsafe lock.

The above rules are enforced for any locking sequence that occurs in the kernel: when acquiring a new lock, the validator checks whether there is any rule violation between the new lock and any of the held locks.

When a lock-class changes its state, the following aspects of the above dependency rules are enforced:

- if a new hardirq-safe lock is discovered, we check whether it
  took any hardirq-unsafe lock in the past.

- if a new softirq-safe lock is discovered, we check whether it took
  any softirq-unsafe lock in the past.

- if a new hardirq-unsafe lock is discovered, we check whether any
  hardirq-safe lock took it in the past.

- if a new softirq-unsafe lock is discovered, we check whether any
  softirq-safe lock took it in the past.

(Again, we do these checks too on the basis that an interrupt context
could interrupt _any_ of the irq-unsafe or hardirq-unsafe locks, which
could lead to a lock inversion deadlock - even if that lock scenario did
not trigger in practice yet.)

Exception: Nested data dependencies leading to nested locking
----

There are a few cases where the Linux kernel acquires more than one
instance of the same lock-class. Such cases typically happen when there
is some sort of hierarchy within objects of the same type. In these
cases there is an inherent "natural" ordering between the two objects
(defined by the properties of the hierarchy), and the kernel grabs the
locks in this fixed order on each of the objects.

An example of such an object hierarchy that results in "nested locking"
is that of a "whole disk" block-dev object and a "partition" block-dev
object; the partition is "part of" the whole device and as long as one
always takes the whole disk lock as a higher lock than the partition
lock, the lock ordering is fully correct. The validator does not
automatically detect this natural ordering, as the locking rule behind
the ordering is not static.

In order to teach the validator about this correct usage model, new
versions of the various locking primitives were added that allow you to
specify a "nesting level". An example call, for the block device mutex,
looks like this:

```
enum bdev_bd_mutex_lock_class
{
       BD_MUTEX_NORMAL,
       BD_MUTEX_WHOLE,
       BD_MUTEX_PARTITION
};

 mutex_lock_nested(&bdev->bd_contains->bd_mutex, BD_MUTEX_PARTITION);
```

In this case the locking is done on a bdev object that is known to be a
partition.

The validator treats a lock that is taken in such a nested fashion as a
separate (sub)class for the purposes of validation.

Note: When changing code to use the _nested() primitives, be careful and

check really thoroughly that the hierarchy is correctly mapped; otherwise you can get false positives or false negatives.

Proof of 100% correctness:

The validator achieves perfect, mathematical 'closure' (proof of locking correctness) in the sense that for every simple, standalone single-task locking sequence that occurred at least once during the lifetime of the kernel, the validator proves it with a 100% certainty that no combination and timing of these locking sequences can cause any class of lock related deadlock. [*]

I.e. complex multi-CPU and multi-task locking scenarios do not have to occur in practice to prove a deadlock: only the simple 'component' locking chains have to occur at least once (anytime, in any task/context) for the validator to be able to prove correctness. (For example, complex deadlocks that would normally need more than 3 CPUs and a very unlikely constellation of tasks, irq-contexts and timings to occur, can be detected on a plain, lightly loaded single-CPU system as well!)

This radically decreases the complexity of locking related QA of the kernel: what has to be done during QA is to trigger as many "simple" single-task locking dependencies in the kernel as possible, at least once, to prove locking correctness - instead of having to trigger every possible combination of locking interaction between CPUs, combined with every possible hardirq and softirq nesting scenario (which is impossible to do in practice).

[*] assuming that the validator itself is 100% correct, and no other part of the system corrupts the state of the validator in any way. We also assume that all NMI/SMM paths [which could interrupt even hardirq-disabled codepaths] are correct and do not interfere with the validator. We also assume that the 64-bit 'chain hash' value is unique for every lock-chain in the system. Also, lock recursion must not be higher than 20.

Performance:

The above rules require _massive_ amounts of runtime checking. If we did that for every lock taken and for every irqs-enable event, it would render the system practically unusably slow. The complexity of checking is O(N^2), so even with just a few hundred lock-classes we'd have to do tens of thousands of checks for every event.

This problem is solved by checking any given 'locking scenario' (unique sequence of locks taken after each other) only once. A simple stack of held locks is maintained, and a lightweight 64-bit hash value is calculated, which hash is unique for every lock chain. The hash value, when the chain is validated for the first time, is then put into a hash table, which hash-table can be checked in a lockfree manner. If the locking chain occurs again later on, the hash table tells us that we dont have to validate the chain again.