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 'usage state' of lock-classes, and it tracks the dependencies between different lock-classes. Lock usage indicates how a lock is used with regard to its IRQ contexts, while lock dependency can be understood as lock order, where L1 -> L2 suggests that a task is attempting to acquire L2 while holding L1. From lockdep's perspective, the two locks (L1 and L2) are not necessarily related; that dependency just means the order ever happened. The validator maintains a continuing effort to prove lock usages and dependencies are correct or the validator will shoot a splat if incorrect.

A lock-class's behavior is constructed by its instances collectively: when the first instance of a lock-class is used after bootup the class gets registered, then all (subsequent) instances will be mapped to the class and hence their usages and dependecies will contribute to those of the class. A lock-class does not go away when a lock instance does, but it can be removed if the memory space of the lock class (static or dynamic) is reclaimed, this happens for example when a module is unloaded or a workqueue is destroyed.

State

The validator tracks lock-class usage history and divides the usage into (4 usages * n STATEs + 1) categories:

where the 4 usages can be:

- '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 the n STATEs are coded in kernel/locking/lockdep_states.h and as of now they include:

- hardirq
- softirq

where the last 1 category is:

- 'ever used' [== !unused]

When locking rules are violated, these usage bits are presented in the locking error messages, inside curlies, with a total of 2 * n STATEs bits. 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
```

For a given lock, the bit positions from left to right indicate the usage of the lock and readlock (if exists), for each of the n STATEs listed above respectively, and the character displayed at each bit position 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.

The bits are illustrated with an example:

```
(&sio_locks[i].lock){-.-.}, at: [<c02867fd>] mutex_lock+0x21/0x24
                     ||||
                     ||| \-> softirq disabled and not in softirq context
                     || \--> acquired in softirq context
                     | \---> hardirq disabled and not in hardirq context
                      \----> acquired in hardirq context
```

For a given STATE, whether the lock is ever acquired in that STATE context and whether that STATE is enabled yields four possible cases as shown in the table below. The bit character is able to indicate which exact case is for the lock as of the reporting time.

	irq enabled	irq disabled
ever in irq	'?'	'-'
never in irq	'+'	'.'

The character '-' suggests irq is disabled because if otherwise the charactor '?' would have been shown instead. Similar deduction can be applied for '+' too.

Unused locks (e.g., mutexes) cannot be part of the cause of an error.

Single-lock state rules:

A lock is irq-safe means it was ever used in an irq context, while a lock is irq-unsafe means it was ever acquired with irq enabled.

A softirq-unsafe lock-class is automatically hardirq-unsafe as well. The following states must be exclusive: only one of them is allowed to be set for any lock-class based on its usage:

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

This is because if a lock can be used in irq context (irq-safe) then it cannot be ever acquired with irq enabled (irq-unsafe). Otherwise, a deadlock may happen. For example, in the scenario that after this lock was acquired but before released, if the context is interrupted this lock will be attempted to acquire twice, which creates a deadlock, referred to as lock recursion deadlock.

The validator detects and reports lock usage that violates 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 can not be taken in inverse order:

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

because this could lead to a deadlock - referred to as lock inversion deadlock - as attempts to acquire the two locks form a circle which could lead to the two contexts waiting for each other permanently. The validator will find such dependency circle in arbitrary complexity, i.e., there can be any other locking sequence between the acquire-lock operations; the validator will still find whether these locks can be acquired in a circular fashion.

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 that 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.

Annotations

Two constructs can be used to annotate and check where and if certain locks must be held: lockdep_assert_held*(&lock) and lockdep_*pin_lock(&lock).

As the name suggests, lockdep_assert_held* family of macros assert that a particular lock is held at a certain time (and generate a WARN() otherwise). This annotation is largely used all over the kernel, e.g. kernel/sched/ core.c:

```
void update_rq_clock(struct rq *rq)
{
      s64 delta;

      lockdep_assert_held(&rq->lock);
      [...]
}
```

where holding rq->lock is required to safely update a rq's clock.

The other family of macros is lockdep_*pin_lock(), which is admittedly only used for rq->lock ATM. Despite their limited adoption these annotations generate a WARN() if the lock of interest is "accidentally" unlocked. This turns out to be especially helpful to debug code with callbacks, where an upper layer assumes a lock remains taken, but a lower layer thinks it can maybe drop and reacquire the lock ("unwittingly" introducing races). lockdep_pin_lock() returns a 'struct pin_cookie' that is then used by lockdep_unpin_lock() to check that nobody tampered with the lock, e.g. kernel/sched/sched.h:

```
static inline void rq_pin_lock(struct rq *rq, struct rq_flags *rf)
{
      rf->cookie = lockdep_pin_lock(&rq->lock);
      [...]
}

static inline void rq_unpin_lock(struct rq *rq, struct rq_flags *rf)
{
      [...]
      lockdep_unpin_lock(&rq->lock, rf->cookie);
}
```

While comments about locking requirements might provide useful information, the runtime checks performed by annotations are invaluable when debugging locking problems and they carry the same level of details when inspecting code. Always prefer annotations when in doubt!

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. [1]

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).

[1] 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 don't have to validate the chain again.

Troubleshooting:

The validator tracks a maximum of MAX_LOCKDEP_KEYS number of lock classes. Exceeding this number will trigger the following lockdep warning:

```
(DEBUG_LOCKS_WARN_ON(id >= MAX_LOCKDEP_KEYS))
```

By default, MAX_LOCKDEP_KEYS is currently set to 8191, and typical desktop systems have less than 1,000 lock classes, so this warning normally results from lock-class leakage or failure to properly initialize locks. These two problems are illustrated below:

1. Repeated module loading and unloading while running the validator will result in lock-class leakage. The issue here is that each load of the module will create a new set of lock classes for that module's locks, but module unloading does not remove old classes (see below discussion of reuse of lock classes for why). Therefore, if that module is loaded and unloaded repeatedly, the number of lock classes will eventually reach the maximum.
2. Using structures such as arrays that have large numbers of locks that are not explicitly initialized. For example, a hash table with 8192 buckets where each bucket has its own spinlock_t will consume 8192 lock classes -unless- each spinlock is explicitly initialized at runtime, for example, using the run-time spin_lock_init() as opposed to compile-time initializers such as __SPIN_LOCK_UNLOCKED(). Failure to properly initialize the per-bucket spinlocks would guarantee lock-class overflow. In contrast, a loop that called spin_lock_init() on each lock would place all 8192 locks into a single lock class.

   The moral of this story is that you should always explicitly initialize your locks.

One might argue that the validator should be modified to allow lock classes to be reused. However, if you are tempted to make this argument, first review the code and think through the changes that would be required, keeping in mind that the lock classes to be removed are likely to be linked into the lock-dependency graph. This turns out to be harder to do than to say.

Of course, if you do run out of lock classes, the next thing to do is to find the offending lock classes. First, the following command gives you the number of lock classes currently in use along with the maximum:

```
grep "lock-classes" /proc/lockdep_stats
```

This command produces the following output on a modest system:

```
lock-classes:                          748 [max: 8191]
```

If the number allocated (748 above) increases continually over time, then there is likely a leak. The following command can be used to identify the leaking lock classes:

```
grep "BD" /proc/lockdep
```

Run the command and save the output, then compare against the output from a later run of this command to identify the leakers. This same output can also help you find situations where runtime lock initialization has been omitted.

Recursive read locks:

The whole of the rest document tries to prove a certain type of cycle is equivalent to deadlock possibility.

There are three types of lockers: writers (i.e. exclusive lockers, like spin_lock() or write_lock()), non-recursive readers (i.e. shared lockers, like down_read()) and recursive readers (recursive shared lockers, like rcu_read_lock()). And we use the following notations of those lockers in the rest of the document:

> W or E: stands for writers (exclusive lockers). r: stands for non-recursive readers. R: stands for recursive readers. S: stands for all readers (non-recursive + recursive), as both are shared lockers. N: stands for writers and non-recursive readers, as both are not recursive.

Obviously, N is "r or W" and S is "r or R".

Recursive readers, as their name indicates, are the lockers allowed to acquire even inside the critical section of another reader of the same lock instance, in other words, allowing nested read-side critical sections of one lock instance.

While non-recursive readers will cause a self deadlock if trying to acquire inside the critical section of another reader of the same lock instance.

The difference between recursive readers and non-recursive readers is because: recursive readers get blocked only by a write lock *holder*, while non-recursive readers could get blocked by a write lock *waiter*. Considering the follow example:

```
TASK A:                 TASK B:

read_lock(X);
                        write_lock(X);
read_lock_2(X);
```

Task A gets the reader (no matter whether recursive or non-recursive) on X via read_lock() first. And when task B tries to acquire writer on X, it will block and become a waiter for writer on X. Now if read_lock_2() is recursive readers, task A will make progress, because writer waiters don't block recursive readers, and there is no deadlock. However, if read_lock_2() is non-recursive readers, it will get blocked by writer waiter B, and cause a self deadlock.

Block conditions on readers/writers of the same lock instance:

There are simply four block conditions:

1. Writers block other writers.
2. Readers block writers.
3. Writers block both recursive readers and non-recursive readers.
4. And readers (recursive or not) don't block other recursive readers but may block non-recursive readers (because of the potential co-existing writer waiters)

Block condition matrix, Y means the row blocks the column, and N means otherwise.

	W	r	R
W	Y	Y	Y
r	Y	Y	N
R	Y	Y	N

(W: writers, r: non-recursive readers, R: recursive readers)

acquired recursively. Unlike non-recursive read locks, recursive read locks only get blocked by current write lock *holders* other than write lock *waiters*, for example:

```
TASK A:                 TASK B:

read_lock(X);

                        write_lock(X);

read_lock(X);
```

is not a deadlock for recursive read locks, as while the task B is waiting for the lock X, the second read_lock() doesn't need to wait because it's a recursive read lock. However if the read_lock() is non-recursive read lock, then the above case is a deadlock, because even if the write_lock() in TASK B cannot get the lock, but it can block the second read_lock() in TASK A.

Note that a lock can be a write lock (exclusive lock), a non-recursive read lock (non-recursive shared lock) or a recursive read lock (recursive shared lock), depending on the lock operations used to acquire it (more specifically, the value of the 'read' parameter for lock_acquire()). In other words, a single lock instance has three types of acquisition depending on the acquisition functions: exclusive, non-recursive read, and recursive read.

To be concise, we call that write locks and non-recursive read locks as "non-recursive" locks and recursive read locks as "recursive" locks.

Recursive locks don't block each other, while non-recursive locks do (this is even true for two non-recursive read locks). A non-recursive lock can block the corresponding recursive lock, and vice versa.

A deadlock case with recursive locks involved is as follow:

```
TASK A:                 TASK B:

read_lock(X);
                        read_lock(Y);
write_lock(Y);
                        write_lock(X);
```

Task A is waiting for task B to read_unlock() Y and task B is waiting for task A to read_unlock() X.

Dependency types and strong dependency paths:

Dependency types and strong dependency paths:

Lock dependencies record the orders of the acquisitions of a pair of locks, and because there are 3 types for lockers, there are, in theory, 9 types of lock dependencies, but we can show that 4 types of lock dependencies are enough for deadlock detection.

For each lock dependency:

```
L1 -> L2
```

, which means lockdep has seen L1 held before L2 held in the same context at runtime. And in deadlock detection, we care whether we could get blocked on L2 with L1 held, IOW, whether there is a locker L3 that L1 blocks L3 and L2 gets blocked by L3. So we only care about 1) what L1 blocks and 2) what blocks L2. As a result, we can combine recursive readers and non-recursive readers for L1 (as they block the same types) and we can combine writers and non-recursive readers for L2 (as they get blocked by the same types).

With the above combination for simplification, there are 4 types of dependency edges in the lockdep graph:

1. -(ER)->:

   exclusive writer to recursive reader dependency, "X -(ER)-> Y" means X -> Y and X is a writer and Y is a recursive reader.

2. -(EN)->:

   exclusive writer to non-recursive locker dependency, "X -(EN)-> Y" means X -> Y and X is a writer and Y is either a writer or non-recursive reader.

3. -(SR)->:

   shared reader to recursive reader dependency, "X -(SR)-> Y" means X -> Y and X is a reader (recursive or not) and Y is a recursive reader.

4. -(SN)->:

   shared reader to non-recursive locker dependency, "X -(SN)-> Y" means X -> Y and X is a reader (recursive or not) and Y is either a writer or non-recursive reader.

Note that given two locks, they may have multiple dependencies between them, for example:

```
TASK A:

read_lock(X);
write_lock(Y);
...

TASK B:

write_lock(X);
write_lock(Y);
```

, we have both X -(SN)-> Y and X -(EN)-> Y in the dependency graph.

We use -(xN)-> to represent edges that are either -(EN)-> or -(SN)->, the similar for -(Ex)->, -(xR)-> and -(Sx)->

A "path" is a series of conjunct dependency edges in the graph. And we define a "strong" path, which indicates the strong dependency throughout each dependency in the path, as the path that doesn't have two conjunct edges (dependencies) as -(xR)-> and -(Sx)->. In other words, a "strong" path is a path from a lock walking to another through the lock dependencies, and if X -> Y -> Z is in the path (where X, Y, Z are locks), and the walk from X to Y is through a -(SR)-> or -(ER)-> dependency, the walk from Y to Z must not be through a -(SN)-> or -(SR)-> dependency.

We will see why the path is called "strong" in next section.

Recursive Read Deadlock Detection:

We now prove two things:

Lemma 1:

If there is a closed strong path (i.e. a strong circle), then there is a combination of locking sequences that causes deadlock. I.e. a strong circle is sufficient for deadlock detection.

Lemma 2:

If there is no closed strong path (i.e. strong circle), then there is no combination of locking sequences that could cause deadlock. I.e. strong circles are necessary for deadlock detection.

With these two Lemmas, we can easily say a closed strong path is both sufficient and necessary for deadlocks, therefore a closed strong path is equivalent to deadlock possibility. As a closed strong path stands for a dependency chain that could cause deadlocks, so we call it "strong", considering there are dependency circles that won't cause deadlocks.

Proof for sufficiency (Lemma 1):

Let's say we have a strong circle:

```
L1 -> L2 ... -> Ln -> L1
```

, which means we have dependencies:

```
L1 -> L2
L2 -> L3
...
Ln-1 -> Ln
Ln -> L1
```

We now can construct a combination of locking sequences that cause deadlock:

Firstly let's make one CPU/task get the L1 in L1 -> L2, and then another get the L2 in L2 -> L3, and so on. After this, all of the Lx in Lx -> Lx+1 are held by different CPU/tasks.

And then because we have L1 -> L2, so the holder of L1 is going to acquire L2 in L1 -> L2, however since L2 is already held by another CPU/task, plus L1 -> L2 and L2 -> L3 are not -(xR)-> and -(Sx)-> (the definition of strong), which means either L2 in L1 -> L2 is a non-recursive locker (blocked by anyone) or the L2 in L2 -> L3, is writer (blocking anyone), therefore the holder of L1 cannot get L2, it has to wait L2's holder to release.

Moreover, we can have a similar conclusion for L2's holder: it has to wait L3's holder to release, and so on. We now can prove that Lx's holder has to wait for Lx+1's holder to release, and note that Ln+1 is L1, so we have a circular waiting scenario and nobody can get progress, therefore a deadlock.

Proof for necessary (Lemma 2):

Lemma 2 is equivalent to: If there is a deadlock scenario, then there must be a strong circle in the dependency graph.

According to Wikipedia[1], if there is a deadlock, then there must be a circular waiting scenario, means there are N CPU/tasks, where CPU/task P1 is waiting for a lock held by P2, and P2 is waiting for a lock held by P3, ... and Pn is waiting for a lock held by P1. Let's name the lock Px is waiting as Lx, so since P1 is waiting for L1 and holding Ln, so we will have Ln -> L1 in the dependency graph. Similarly, we have L1 -> L2, L2 -> L3, ..., Ln-1 -> Ln in the dependency graph, which means we have a circle:

```
Ln -> L1 -> L2 -> ... -> Ln
```

, and now let's prove the circle is strong:

For a lock Lx, Px contributes the dependency Lx-1 -> Lx and Px+1 contributes the dependency Lx -> Lx+1, and since Px is waiting for Px+1 to release Lx, so it's impossible that Lx on Px+1 is a reader and Lx on Px is a recursive reader, because readers (no matter recursive or not) don't block recursive readers, therefore Lx-1 -> Lx and Lx -> Lx+1 cannot be a -(xR)-> -(Sx)-> pair, and this is true for any lock in the circle, therefore, the circle is strong.

References:

[1]: https://en.wikipedia.org/wiki/Deadlock [2]: Shibu, K. (2009). Intro To Embedded Systems (1st ed.). Tata McGraw-Hill