RTNL mutex, the network stack big kernel lock

Florian Westphal
Red Hat
fw@strlen.de

Abstract

The RTNL mutex is used to serialize rtnetlink[4] [5] requests. rtnetlink is a netlink[3] subsystem used to inspect or change networking related configuration. Types of rtnetlink requests range from changing link state, adding/removing IP or IPv6 addresses and routes to qdiscs and traffic classifiers.

The widespread use of the RTNL mutex in all major network configuration paths is a growing pain point, f.e. a task adding an IP address prevents another from from seemingly unrelated tasks such as dumping tc classifiers. Furthermore, some code paths can hold the RTNL mutex for very long times (in the order of several hundreds of milliseconds in some cases).

Since 4.14 kernel there is basic infrastructure in place to elide the RTNL mutex for some operations.

This talk dives into the history of the RTNL mutex, explains why its widely used, and discusses needed steps to further reduce RTNL mutex usage in the kernel with the goal to improve concurrency and reduce latency.

What is rtnetlink?

rtnetlink is a netlink-based configuration interface for network related configuration in the Linux kernel.

By kernel standards, it is ancient – it was added more than 20 years ago and has been a requirement since 2001 when network support is enabled in the kernel build.

Kernel API

To register a handler within the kernel, one uses the `rtnl_register` function:

```
void rtnl_register(int proto, int msgtype,
                   rtnl_doit_func,
                   rtnl_dumpit_func);
```

The rtnetlink core will then invoke the **doit** function when user space sends an rtnetlink message of type **msgtype** for protocol **protocol**. If a dump is requested (indicated by a flag in the netlink message header), the **dumpit** function is invoked instead.

The callbacks will validate and decode the netlink attributes contained in the message, an then perform the desired action. Actions are diverse, they range from e.g. adding a new GRE tunnel to removal of IP addresses or dumping all configured traffic classifiers to user space.

RTNL mutex problems

Up to and including kernel 4.13, all rtnetlink callbacks are serialized by the RTNL mutex:

```
void rtnetlink_rcv(struct sk_buff *skb)
{
  rtnl_lock();
  netlink_rcv_skb(skb, &rtnetlink_rcv_msg);
  rtnl_unlock();
}
```

In other words, if e.g. a routing daemon is adding a new entry to the IP FIB, an unrelated program that wishes to take a look at the configured network interfaces needs to wait as both requests depend on the same lock.

Another major data structure that is protected by the RTNL mutex is the list of network namespaces.

Contention on the RTNL mutex is made more severe by the fact that the mutex can be held for very long times, as callbacks can sleep, for example due to `GFP_KERNEL` allocation or because a callback needs to wait for rcu read side critical sections to finish. Waiting until all CPUs have completed their read-side sections can be expensive, especially on busy systems without kernel preemption and many cores[2]. In configuration paths the network stack often uses `synchronize_net()` which will speed things up when the RTNL mutex is held (by using `synchronize_rcu_expedited()` at cost of more latency).

Examples of callbacks that need to wait for RCU include moving a device to another namespace or removing a configured link from the system.

While various ways were added to avoid or minimize calls to `synchronize_rcu`, such as batching multiple exiting net namespaces, or to provide ways to release the RTNL faster (`synchronize_net`), the obviously better choice would be to avoid RTNL serialization as much as possible.

rtnetlink in 4.14

In Linux 4.14 infrastructure to elide the RTNL mutex was added[6]. Simply put, the rtnetlink core now provides a way for in-kernel users of the API to indicate that the doit callback should be invoked without acquiring the RTNL mutex:

```
void rtnetlink_rcv(struct sk_buff *skb)
{
```

```
 netlink_rcv_skb(skb, &rtnetlink_rcv_msg);
}

rtnetlink_rcv_msg() {
 [..]
 flags = handlers[type].flags;
 doit = handlers[type].doit;
 if (flags & RTNL_FLAG_DOIT_UNLOCKED)
      return doit(skb, nlh, extack);
 rtnl_lock();
 err = doit(skb, nlh, extack);
 rtnl_unlock();
 return err;
```

This opens a path to slowly convert callbacks to not depend on RTNL locking guarantees anymore.

converting callbacks to not depend on RTNL lock

An example for a doit callback that can elide RTNL mutex is IP and IPv6 `RTM_GETROUTE`: It performs a route lookup with input keys defined by user space and returns the result to user space. This callback is read-only, i.e. no kernel data structures are modified.

Another example for a easy-to-convert handler that does modify in-kernel data are IPv6 address labels. The table that is modified is already protected by a spinlock so serialization vs. other tasks that read or change IPv6 address labels is already guaranteed even if the RTNL mutex is not acquired anymore. Unfortunately, most callbacks are more complicated, even if they do not modify state.

RTNL removal obstacles: consistent guarantees

There are several reasons why the RTNL mutex cannot be blindly removed from handlers. The major obstacle is that the RTNL mutex also provides consistency:

```
rtnl_fill_ifinfo:
 nla_put_string(skb, IFNAME, dev->name);
```

If this code part would be called without the RTNL mutex, then there is a small chance that it would provide a garbled interface name back to user space.

This is not an unavoidable problem, the kernel already provides a helper function to obtain a consistent name, its only a matter of changing the above to use `netdev_get_name()` helper which uses an internal sequence lock to synchronize with a rename done by another task.

This is one of the reasons why even dump requests (which just send back a netlink-serialized view of the current state to user space) are all serialized via RTNL – and thus block both other dumps and new RTNL config requests.

Unfortunately, the above is just one of many examples where existing handlers make assumptions about RTNL mutex being held – two other problematic cases are the RTNL AF and link operations. Out of the two, the former is easier to resolve[1]:

```
struct rtnl_af_ops {
 int family;
 (*fill_link_af)(struct sk_buff *,
                 struct net_device *,
                 u32 ext_filter_mask);
 (*get_link_af_size)
 (struct net_device *,
          u32 filter_mask);
 (*validate_link_af)(struct net_device *,
                     struct nlattr *attr);
 (*set_link_af)(struct net_device *dev,
                struct nlattr *attr);
 (*fill_stats_af)(struct sk_buff *skb,
                  struct net_device *dev);
 (*get_stats_af_size)(struct net_device *);
};
```

[1] structure edited for brevity

Only a few instances of these af ops exist in the kernel, and the various implementations of the callbacks are small, making it much easier to audit them for places that make RTNL-is-locked assumptions – none of them appear to need the RTNL mutex. Also, no callback needs to sleep, i.e. it is enough to convert the callers to take the rcu read lock and using appropriate rcu synchronization primitives when af ops are unregistered.

The only apparent downside is that such a conversion has no immediate benefit, because all callers hold the RTNL mutex. However, it is a prerequisite for converting more doit handlers in the future and also removes the af ops from the list of obstacles that stand in the way of further RTNL mutex removal from handlers.

RTNL removal obstacles: link ops

Link ops on the other hand are added from a myriad of different places (usually by drivers such as ipoib, bonding, ppp and various types of tunnels). At the very least the `new` and `dellink` callbacks invoke functions that assume RTNL mutex is held.

What appears to be problematic is synchronizing with `rtnl_link_unregister` occurring on another CPU. Right now there there are no issues: `rtnl_link_unregister` grabs the RTNL mutex, deletes the `rtnl_link_op` from the global list and then triggers unregister/dellink of all net devices that use the link op.

Because all places that dereference `dev->rtnl_link_ops` depend on RTNL, the unregister function will block until all rtnetlink operations that use it are done. rtnetlink ops that come later will fail to find the device, as it has already been removed from list.

This means that the following appears to not work:

```
dev = dev_get_by_index(ifindex);
...
rtnl_lock();
dev->rtnl_link_ops->...()
rtnl_unlock();
```

A parallel rmmod can unregister the link ops that `dev->rtnl_link_ops` points to. However, looking at `rtnl_link_unregister` in detail:

```
rtnl_link_unregister(struct rtnl_link_ops *ops)
{
 struct net *net;
```

```
 /* Close the race with cleanup_net() */
 mutex_lock(&net_mutex);
 rtnl_lock_unregistering_all();
 for_each_net(net)
   __rtnl_kill_links(net, ops);
 list_del(&ops->list);
 rtnl_unlock();
 mutex_unlock(&net_mutex);
}
```

Deleting links removes affected devices from the global list (i.e., further `dev_get_by_index()` calls won't return these devices anymore). `rtnl_unlock` will block until all these devices have their reference counts drop to zero. This means that

```
rcu_read_lock();
dev = dev_get_by_index_rcu(ifindex);
...
dev->rtnl_link_ops->...()
rcu_read_unlock();
```

... is also safe because `rtnl_link_unregister` also waits for at least one rcu grace period to elapse.

In other words, RTNL link ops handling is safe in all doit handlers provided at least one of the following is true:

1. doit acquires RTNL mutex
2. doit takes reference count of the device that the `link_ops` are assigned to
3. doit uses rcu read lock + `dev_get_by_index_rcu`

This leaves only one more possible issue: `IFLA_INFO_KIND`. When a new link is to be created, user space provides the link type name in a netlink attribute. The kernel will then retrieve the `rtnl_link_ops` struct that corresponds to the given link name. Currently all these callers hold the RTNL mutex, so this is safe. However, because no device exists yet the only alternative to RTNL would be to use rcu read lock instead (or adding a reference count to `rtnl_link_ops`, but that should be avoided if possible.

Issues and ongoing work

Given the intertwined nature of various components of the network configuration backplane RTNL mutex removal is error prone.

The current work focuses on slowly "pushing down" mutex lock/unlock operations. This often means that the initial work consist of changing code like

```
dev = __dev_get_by_name(net, devname);
```

to use a lookup version that either increments the device reference count (`dev_get_by_name()`) or relies on rcu protection (`dev_get_by_name_rcu`).

Such patches are, by themselves, useless – but they are a required initial step to possibly allow calling such functions without holding the RTNL mutex.

RTNL mutex removal is further complicated by the large amount of code that can be executed while under RTNL mutex protection, for instance adding bridge FDB entries can cause calls into the driver for offloading purposes (ndo ops).

Unfortunately, such ndo op calls into network drivers are not uncommon, due to ever increasing number of network offload features, including vlan, SR-IOV, switching and bpf offloading.

This means that even if RTNL mutex removal is safe in some cases it might be needed to add per-driver locks to protect interaction with hardware.

The following two sections provide two examples to illustrate further issues with RTNL mutex removal, especially dependencies created via use of netdevice notifiers.

devinet

`net/ipv4/devinet.c` implements IP device support routines. This includes, among others, device sysctl settings (e.g. `forwarding`, `accept_redirects`, `rp_filter` and so forth), an ioctl interface to the IP network stack (used by the "old" net-tools such as `ifconfig` and `route`), and IP address assignment to interfaces.

devinet handlers commonly acquire the RTNL mutex for serializing requests coming from either rtnetlink or the older ioctl based interface.

RTNL mutex removal in devinet is complicated by address addition. Specifically, when a new address is supposed to be added to an interface, the kernel invokes RTNL notifier chain:

```
int __inet_insert_ifa(struct in_ifaddr, ...

ret = blocking_notifier_call_chain(
&inetaddr_validator_chain,
        NETDEV_UP, &ivi);
```

This allows in-kernel users such as the ipvlan driver, to veto (and thus prevent) adding addresses that would cause a conflict with whatever internal state the vetoing entity has.

Its clear that this requires some sort of serialization to prevent races, but it does not appear impossible to realize this synchronization by a mechanism other than the RTNL mutex.

The same notifier facility exists in the IPv6 stack.

IP FIB

Adding and deleting entries from the IPv4 FIB also occurs via rtnetlink, serialized by the RTNL mutex.

As mutexes can't be acquired in the network packet path (i.e. soft irq context), and most FIB lookups occur from the packet path, lookups in the IPv4 FIB are already fully RCU safe.

Like with devinet, it seems appealing to introduce a new FIB-private mutex to enforce serialization within FIB without affecting other parts of the kernel.

However, adding another mutex introduces the possibility of ABBA-style deadlocks:

```
rtnl_lock(); ...
mutex_lock(&fib_private);
```

vs.

```
mutex_lock(&fib_private); ...
rtnl_lock()
```

So any addition of a new private mutex is only feasible if strict ordering can be guaranteed. Since it makes no sense to add a new mutex that can only be safely acquired when the caller already holds the RTNL mutex the only sane solution is to make the common add/delete manipulations only take the new FIB mutex.

```
mutex_lock(&fib_private);
/* do add/delete from fib */
mutex_unlock(&fib_private);
```

FIB manipulations also occur indirectly by the kernel when e.g. a device state changes to **UP**. The FIB gets notified about such events by inetaddr and netdevice notifiers. These notifiers are invoked with the RTNL mutex held.

This means that in this case it would now be necessary to acquire the fib private mutex (to make changes to the FIB) while the caller is already holding the RTNL mutex – this results in deadlocks unless acquiring the RTNL mutex is not allowed while the private FIB mutex is held.

Not using the RTNL mutex for direct FIB manipulation seems doable, however, this requires auditing the dump consistency checks.

At this time, whenever the IP FIB notifier is invoked it increments a counter, stored per netns:

```
call_fib4_notifiers(struct net *net,
                    enum fib_event_type event,
                    struct fib_notifier_info *i)
{
  ASSERT_RTNL();
  info->family = AF_INET;
  net->ipv4.fib_seq++;
  return call_fib_notifiers(net, event, i);
}

static int fib4_seq_read(struct net *net)
{
   ASSERT_RTNL();
   return net->ipv4.fib_seq +
          fib4_rules_seq_read(net);
}
```

It seems possible to switch `fib_seq` to `atomic_t` to ensure the sequence counter will continue to increment monotonically even if the RTNL mutex is not held anymore, provided that changes to the data structures do not cause inconsistencies during read operations. In the FIB case this is fine because the normal case (FIB lookups from packet path) are already lockless. But this means that its possible to miss updates:

1. a new FIB entry gets added
2. a dump request starts, fetches current counter
3. the new FIB entry is linked into the list
4. the dump request finishes, fetches counter, dump will be considered consistent
5. `call_fib4_notifiers()` is invoked and increments the sequence counter

A function that is affected by this problem is `register_fib_notifier`. The alternative is to change `fib_seq` to `seqcount_t` type, and increment the sequence number twice: once right before adding/removing an entry (making the counter and odd number) and again when the entry has been added to the internal data structures.

This would allow a dump to wait for the counter to stabilize first to not miss the new entry. A similar sequence counter exists in IPv6 FIB and policy rules, the same solutions can be applied there.

netlink dumps

One nice improvement would be to always permit userspace to perform rtnetlink dump requests regardless of normal requests.

This was already attempted a couple of years ago [1] but this change had to be reverted quickly because of a large number of RTNL mutex assumptions.

After recent changes, most parts of `rtnl_fill_ifinfo` now no longer depend on the RTNL mutex. Some notable cases that still need the RTNL mutex are most offload-related NDO ops, such as retrieving names of physical ports attached to switch devices, and obtaining the name of the configured qdisc.

At the time of this writing, without RTNL mutex protection, nothing prevents free of qdisc while netlink dump request is accessing it.

Summary

The widespread use of the RTNL mutex in network configuration paths and the dependencies on the mutex makes it difficult to reduce its usage, albeit not impossible. Allowing readers to run concurrently with an application that modifies state would be an important first step. Initial work has been completed but the more frequently used ones, such as dumps of routing table or qdisc stats are still unsolved.

References

[1] Dumazet, E. 2011a. net: dont hold rtnl mutex during netlink dump callbacks. commit e67f88dd12f610da98ca838822f2c9b4e7c6100e.

[2] Dumazet, E. 2011b. net: use synchronize_rcu_expedited(). commit be3fc413da9eb17cce0991f214ab019d16c88c41.

[3] 2017. Linux man pages project: netlink - communication between user and kernel space. *man 7 netlink*. `https://www.kernel.org/doc/man-pages/`.

[4] 2017a. Linux man pages project: rtnetlink - linux ipv4 routing socket. *man 7 rtnetlink*. `https://www.kernel.org/doc/man-pages/`.

[5] 2017b. Linux man pages project: rtnetlink protocol. *man 3 rtnetlink*. `https://www.kernel.org/doc/man-pages/`.

[6] Westphal, F. 2017. rtnetlink: add rtnl_flag_doit_unlocked. commit 62256f98f244fbb1c7a10465e1ee412f209d8978.