Network Devices, the Kernel, and You!

Introduction

The following is a random collection of documentation regarding network devices.

struct net_device lifetime rules

Network device structures need to persist even after module is unloaded and must be allocated with alloc_netdev_mqs() and friends. If device has registered successfully, it will be freed on last use by free_netdev(). This is required to handle the pathological case cleanly (example: `rmmod mydriver </sys/class/net/myeth/mtu`)

alloc_netdev_mqs() / alloc_netdev() reserve extra space for driver private data which gets freed when the network device is freed. If separately allocated data is attached to the network device (netdev_priv()) then it is up to the module exit handler to free that.

There are two groups of APIs for registering struct net_device. First group can be used in normal contexts where `rtnl_lock` is not already held: register_netdev(), unregister_netdev(). Second group can be used when `rtnl_lock` is already held: register_netdevice(), unregister_netdevice(), free_netdevice().

Simple drivers

Most drivers (especially device drivers) handle lifetime of struct net_device in context where `rtnl_lock` is not held (e.g. driver probe and remove paths).

In that case the struct net_device registration is done using the register_netdev(), and unregister_netdev() functions:

```
int probe()
{
  struct my_device_priv *priv;
  int err;

  dev = alloc_netdev_mqs(...);
  if (!dev)
    return -ENOMEM;
  priv = netdev_priv(dev);

  /* ... do all device setup before calling register_netdev() ...
   */

  err = register_netdev(dev);
  if (err)
    goto err_undo;

  /* net_device is visible to the user! */

err_undo:
  /* ... undo the device setup ... */
  free_netdev(dev);
  return err;
}

void remove()
{
  unregister_netdev(dev);
  free_netdev(dev);
}
```

Note that after calling register_netdev() the device is visible in the system. Users can open it and start sending / receiving traffic immediately, or run any other callback, so all initialization must be done prior to registration.

unregister_netdev() closes the device and waits for all users to be done with it. The memory of struct net_device itself may still be referenced by sysfs but all operations on that device will fail.

free_netdev() can be called after unregister_netdev() returns on when register_netdev() failed.

Device management under RTNL

Registering struct net_device while in context which already holds the `rtnl_lock` requires extra care. In those scenarios most drivers will want to make use of struct net_device's `needs_free_netdev` and `priv_destructor` members for freeing of state.

Example flow of netdev handling under `rtnl_lock`:

```
static void my_setup(struct net_device *dev)
{
  dev->needs_free_netdev = true;
}

static void my_destructor(struct net_device *dev)
{
  some_obj_destroy(priv->obj);
  some_uninit(priv);
}

int create_link()
{
  struct my_device_priv *priv;
  int err;

  ASSERT_RTNL();

  dev = alloc_netdev(sizeof(*priv), "net%d", NET_NAME_UNKNOWN, my_setup);
  if (!dev)
    return -ENOMEM;
  priv = netdev_priv(dev);

  /* Implicit constructor */
  err = some_init(priv);
  if (err)
    goto err_free_dev;

  priv->obj = some_obj_create();
  if (!priv->obj) {
    err = -ENOMEM;
    goto err_some_uninit;
  }
  /* End of constructor, set the destructor: */
  dev->priv_destructor = my_destructor;

  err = register_netdevice(dev);
  if (err)
    /* register_netdevice() calls destructor on failure */
    goto err_free_dev;

  /* If anything fails now unregister_netdevice() (or unregister_netdev())
   * will take care of calling my_destructor and free_netdev().
   */

  return 0;

err_some_uninit:
  some_uninit(priv);
err_free_dev:
  free_netdev(dev);
  return err;
}
```

If struct net_device.priv_destructor is set it will be called by the core some time after unregister_netdevice(), it will also be called if register_netdevice() fails. The callback may be invoked with or without `rtnl_lock` held.

There is no explicit constructor callback, driver "constructs" the private netdev state after allocating it and before registration.

Setting struct net_device.needs_free_netdev makes core call free_netdevice() automatically after unregister_netdevice() when all references to the device are gone. It only takes effect after a successful call to register_netdevice() so if register_netdevice() fails driver is responsible for calling free_netdev().

free_netdev() is safe to call on error paths right after unregister_netdevice() or when register_netdevice() fails. Parts of netdev (de)registration process happen after `rtnl_lock` is released, therefore in those cases free_netdev() will defer some of the processing until `rtnl_lock` is released.

Devices spawned from struct rtnl_link_ops should never free the struct net_device directly.

.ndo_init and .ndo_uninit

`.ndo_init` and `.ndo_uninit` callbacks are called during net_device registration and de-registration, under `rtnl_lock`. Drivers can use those e.g. when parts of their init process need to run under `rtnl_lock`.

`.ndo_init` runs before device is visible in the system, `.ndo_uninit` runs during de-registering after device is closed but other subsystems may still have outstanding references to the netdevice.

MTU

Each network device has a Maximum Transfer Unit. The MTU does not include any link layer protocol overhead. Upper layer protocols must not pass a socket buffer (skb) to a device to transmit with more data than the mtu. The MTU does not include link layer header overhead, so for example on Ethernet if the standard MTU is 1500 bytes used, the actual skb will contain up to 1514 bytes because of the Ethernet header. Devices should allow for the 4 byte VLAN header as well.

Segmentation Offload (GSO, TSO) is an exception to this rule. The upper layer protocol may pass a large socket buffer to the device transmit routine, and the device will break that up into separate packets based on the current MTU.

MTU is symmetrical and applies both to receive and transmit. A device must be able to receive at least the maximum size packet allowed by the MTU. A network device may use the MTU as mechanism to size receive buffers, but the device should allow packets with VLAN header. With standard Ethernet mtu of 1500 bytes, the device should allow up to 1518 byte packets (1500 + 14 header + 4 tag). The device may either: drop, truncate, or pass up oversize packets, but dropping oversize packets is preferred.

struct net_device synchronization rules

ndo_open:

> Synchronization: rtnl_lock() semaphore. Context: process

ndo_stop:

> Synchronization: rtnl_lock() semaphore. Context: process Note: netif_running() is guaranteed false

ndo_do_ioctl:

> Synchronization: rtnl_lock() semaphore. Context: process
>
> This is only called by network subsystems internally, not by user space calling ioctl as it was in before linux-5.14.

ndo_siocbond:

> Synchronization: rtnl_lock() semaphore. Context: process
>
> Used by the bonding driver for the SIOCBOND family of ioctl commands.

ndo_siocwandev:

> Synchronization: rtnl_lock() semaphore. Context: process
>
> Used by the drivers/net/wan framework to handle the SIOCWANDEV ioctl with the if_settings structure.

ndo_siocdevprivate:

> Synchronization: rtnl_lock() semaphore. Context: process
>
> This is used to implement SIOCDEVPRIVATE ioctl helpers. These should not be added to new drivers, so don't use.

ndo_eth_ioctl:

> Synchronization: rtnl_lock() semaphore. Context: process

ndo_get_stats:

> Synchronization: rtnl_lock() semaphore, dev_base_lock rwlock, or RCU. Context: atomic (can't sleep under rwlock or RCU)

ndo_start_xmit:

> Synchronization: __netif_tx_lock spinlock.
>
> When the driver sets NETIF_F_LLTX in dev->features this will be called without holding netif_tx_lock. In this case the driver has to lock by itself when needed. The locking there should also properly protect against set_rx_mode. WARNING: use of NETIF_F_LLTX is deprecated. Don't use it for new drivers.
>
> Context: Process with BHs disabled or BH (timer),
> > will be called with interrupts disabled by netconsole.
>
> Return codes:
>
> - NETDEV_TX_OK everything ok.
> - NETDEV_TX_BUSY Cannot transmit packet, try later Usually a bug, means queue start/stop flow control is broken in the driver. Note: the driver must NOT put the skb in its DMA ring.

ndo_tx_timeout:

> Synchronization: netif_tx_lock spinlock; all TX queues frozen. Context: BHs disabled Notes: netif_queue_stopped() is guaranteed true

ndo_set_rx_mode:

> Synchronization: netif_addr_lock spinlock. Context: BHs disabled

struct napi_struct synchronization rules

napi->poll:

Synchronization:

NAPI_STATE_SCHED bit in napi->state. Device driver's ndo_stop method will invoke napi_disable() on all NAPI instances which will do a sleeping poll on the NAPI_STATE_SCHED napi->state bit, waiting for all pending NAPI activity to cease.

Context:

softirq will be called with interrupts disabled by netconsole.