Network Devices, the Kernel, and You!

Introduction

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

struct net_device allocation rules

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

There are routines in net_init.c to handle the common cases of
alloc_etherdev, alloc_netdev. These 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(dev)) then it is up to the module exit handler to free that.

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

dev->open:
        Synchronization: rtnl_lock() semaphore.
        Context: process

dev->stop:
        Synchronization: rtnl_lock() semaphore.
        Context: process

Note1: netif_running() is guaranteed false
Note2: dev->poll() is guaranteed to be stopped

dev->do_ioctl:
Synchronization: rtnl_lock() semaphore.
Context: process

dev->get_stats:
Synchronization: dev_base_lock rwlock.
Context: nominally process, but don't sleep inside an rwlock

dev->hard_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. It is recommended to use a try lock
for this and return NETDEV_TX_LOCKED when the spin lock fails.
The locking there should also properly protect against
set_multicast_list. Note that the 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:
o NETDEV_TX_OK everything ok.
o 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.
o NETDEV_TX_LOCKED Locking failed, please retry quickly.
Only valid when NETIF_F_LLTX is set.

dev->tx_timeout:
Synchronization: netif_tx_lock spinlock.
Context: BHs disabled
Notes: netif_queue_stopped() is guaranteed true

dev->set_multicast_list:
Synchronization: netif_tx_lock spinlock.
Context: BHs disabled

struct napi_struct synchronization rules
==
napi->poll:
Synchronization: NAPI_STATE_SCHED bit in napi->state. Device
driver's dev->close 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.