Device Layer output processing

Bypassing the details of ARP for the moment we consider the *dev_queue_xmit()* function that is used to queue a buffer for transmission. Linux supports priority based output scheduling policies that are described via the *Qdisc* structure defined in *include/net/sch_generic* as:

```
struct Qdisc
{
   int (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
   struct sk_buff *  (*dequeue)(struct Qdisc *dev);
   unsigned                 flags;
#define TCQ_F_BUILTIN   1
#define TCQ_F_THROTTLED 2
#define TCQ_F_INGRESS   4
   int                      padded;
   struct Qdisc_ops         *ops;
   u32                      handle;
   u32                      parent;
   atomic_t                 refcnt;
   struct sk_buff_head      q;
   struct net_device        *dev;
   struct list_head         list;

   struct gnet_stats_basic     bstats;
   struct gnet_stats_queue     qstats;
   struct gnet_stats_rate_est  rate_est;
   spinlock_t                 *stats_lock;
   struct rcu_head             q_rcu;
   int          (*reshape_fail)(struct sk_buff *skb,
                                struct Qdisc *q);

   /* This field is deprecated, but it is still used by CBQ
    * and it will live until better solution will be invented.
    */
   struct Qdisc             *__parent;
};
```

Structure elements are used as follows:

enqueue: This function enqueues an *sk_buff* in the proper position on the proper queue. The default function is *pfifo_fast_enqueue()*. It selects among three queues based upon a packet *priority* that is derived from the IP *tos* and employees FIFO discipline within each queue.

dequeue: The default function here is *pfifo_fast_dequeue()*. It removes the oldest packet from the highest priority non—empty queue.

data: This USED TO BE is a place holder for an array of *sk_buff_head* structures that serve as the bases for the packet queues. Now it seems that an *unnamed* array appended to the end of the structure serves this function.

The *dev_queue_xmit()* function.

For devices that support priority queuing, the *dev_queue_xmit()* function enqueues and then attempts to dequeue and initiate the transmission of the *sk_buff* that is passed as a parameter. It seems a bit odd to enqueue and then immediately dequeue a packet, but in the absence of multiple competing packet streams that is the normal case.

For devices that don't support priority queuing, *dev_queue_xmit()* will attempt to convey the packet directly to the device driver.

At entry to this routine, it is necessary that *skb->dev* point to the outgoing *net_device* and that *skb->priority* contain a value between 0 and 15.

```
int dev_queue_xmit(struct sk_buff *skb)
{
   struct net_device *dev = skb->dev;
   struct Qdisc *q;
   int rc = -ENOMEM;

   /* GSO will handle the following emulations directly. */
   if (netif_needs_gso(dev, skb))
      goto gso;

```

Non GSO devices

Even non-GSO devices may support fragment list structures (though its questionable how many might fall into that category) and they also may support hardware checksumming.

If the device does not support fragment lists, scatter gather operations, and high memory DMA it is necessary to make the *sk_buff* linear. If that doesn't work the packet is dropped.

```
    if (skb_shinfo(skb)->frag_list &&
        !(dev->features & NETIF_F_FRAGLIST) &&
        __skb_linearize(skb))
       goto out_kfree_skb;

/* Fragmented skb is linearized if device does not support SG,
 * or if at least one of fragments is in highmem and device
 * does not support DMA from it.
 */
    if (skb_shinfo(skb)->nr_frags &&
        (!(dev->features & NETIF_F_SG) ||
         illegal_highdma(dev, skb)) &&
         _skb_linearize(skb))
         goto out_kfree_skb;
```

Checksum computation

Checksums, especially for TCP, are now performed by some advanced NIC's. It appears that for UDP packets in Linux CHECKSUM_HW will *never* be set. The variable *skb->h* is a union containing various names for pointers to the *transport* header. The *skb_checksum_help()* function computes a checksum over the region from *skb->h.raw* to *skb->tail* and stores it at an offset of *skb->csum* from *skb->h.raw* . Thus the *csum* field must already be set to the offset in bytes of the 16 bit checksum from the start of the transport header.

The micro code in the NIC does *NOT* understand the *struct sk_buff*. But smart NICs understand the location of the headers in the packet and can differentiate between UDP and TCP (but NOT COP) and can compute proper IP checksums.

Non IP protocols are also assumed non-hardware checksummable. The value of *skb->csum* is the offset from the start of the transport header to the location of the checksum.

```
*(u16*)(skb->h.raw + skb->csum) = csum_fold(csum);
```

```
/* If packet is not checksummed and device does not support
 * checksumming for this protocol, complete checksumming here.
 */
if (skb->ip_summed == CHECKSUM_HW &&
    (!(dev->features & NETIF_F_GEN_CSUM) &&
     (!(dev->features & NETIF_F_IP_CSUM) ||
      skb->protocol != htons(ETH_P_IP))))
         if (skb_checksum_help(skb, 0))
             goto out_kfree_skb;
```

Enqueing the packet

As seen below a device *must* provide a *struct Qdisc,* but the *struct Qdisc* may or may not provide an *enqueue()* function. If an *enqueue()* function has been provided by the device, it is invoked and passed pointer to the *sk_buff* and *Qdisc* structures. For generic ethernet drivers *q->enqueue* points to the function *pfifo_fast_enqueue()* which is defined in *net/sched/sch_generic.c.*

The *goto gso* observed earlier skips the linearization and checksum code and arrives here.

BOTH the *enqueue* and the *dequeue* code runs in the context of both application and soft irq.

- A protocol that supports ACKs will send them in the context of an Rx *softirq*
- When a device transitions from the *stopped* state it will schedule a Tx *softirq* that will call *qdisc_run()*
- Hence the *front* end of *dev_queue_xmit()* can run in the context of an application or an Rx softirq and the back end can run in any of the three contexts.

Hence a rather subtle locking scheme is used to prevent preemption while manipulating the packet queues. Both preemption and sleeping are equally fatal while holding a spin lock.

```
gso:
    spin_lock_prefetch(&dev->queue_lock);

    /* Disable soft irqs for various locks below. Also
     * stops preemption for RCU.
     */
    rcu_read_lock_bh();

    /* Updates of qdisc are serialized by queue_lock.
     * The struct Qdisc which is pointed to by qdisc is now a
     * rcu structure - it may be accessed without acquiring
     * a lock (but the structure may be stale.) The freeing of the
     * qdisc will be deferred until it's known that there are no
     * more references to it.
     *
     * If the qdisc has an enqueue function, we still need to
     * hold the queue_lock before calling it, since queue_lock
     * also serializes access to the device queue.
     */
```

Enqueuing the packet

The device *must* have a Qdisc. If not, line 1478 would cause a kernel oops.. but the Qdisc doesn't necessarily have to have an enqueue function, but all normal Ethernet devices do. The device's queue lock must be held when the device's enqueue() function is called.

```
1473
1474    q = rcu_dereference(dev->qdisc);
1475 #ifdef CONFIG_NET_CLS_ACT
1476    skb->tc_verd = SET_TC_AT(skb->tc_verd,AT_EGRESS);
1477 #endif
1478    if (q->enqueue) {
1479       /* Grab device queue */
1480       spin_lock(&dev->queue_lock);
1481       q = dev->qdisc;
1482       if (q->enqueue) {
1483          rc = q->enqueue(skb, q);
```

On return to *dev_queue_xmit()* the *qdisc_run()* function is called to *attempt to dequeue the packet that was just enqueued.* The queue lock must be held when calling *qdisc_run()* and *must* be dropped before return .

```
1484          qdisc_run(dev);
1485          spin_unlock(&dev->queue_lock);
1486
```

After the return from qdisc_run() an unconditional jump to the exit point is made.

```
1487          rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
1488          goto out;
1489       }
```

Arrival here means the queue structure didn't have an enqueue function so the queue lock is dropped.

```
1490       spin_unlock(&dev->queue_lock);
1491    }
1492
```

Devices that don't have queuing disciplines

Falling into this code means that the *device didn't support a priority queue structure*. Software devices such as loopback and tunnels often don't support the priority queuing mechanism. Some of this code is duplicated in *qdisc_run()*.

```
    /* The device has no queue. Common case for software devices:
       loopback, all the sorts of tunnels...

       Really, it is unlikely that netif_tx_lock protection is necessary
       here.  (f.e. loopback and IP tunnels are clean ignoring statistics
       counters.)
       However, it is possible, that they rely on protection
       made by us here.

       Check this and shot the lock. It is not prone from deadlocks.
       Either shot noqueue qdisc, it is even simpler 8)
     */
     if (dev->flags & IFF_UP) {
        int cpu = smp_processor_id(); /* ok because BHs are
                                             off */

```

Each device has a spinlock called the *xmit_lock()* that prevents multiple CPU's from simultaneously running the driver's *hard_start_xmit()* function.

```
        HARD_TX_LOCK(dev, cpu);

```

The HARD_TX_LOCK macro operates as shown.

```
#define HARD_TX_LOCK(dev, cpu) {                        \
   if ((dev->features & NETIF_F_LLTX) == 0) {   \
        netif_tx_lock(dev);                              \
   }                                                     \
}

```

```
static inline void netif_tx_lock(struct net_device *dev)
{
   spin_lock(&dev->_xmit_lock);
   dev->xmit_lock_owner = smp_processor_id();
}
```

Testing for stopped queue

The *netif_queue_stopped* macro tests the *__LINK_STATE_XOFF* bit in the *dev->state*. Virtually all modern NICs support hardware queuing of pending *tx* requests. When the hardware queue is full, the device driver uses the *netif_stop_queue()* to set this bit. When some packets have drained the driver will reset the bit with a call to *netif_start_queue()*. It is always *verboten* to call *dev->hard_start_xmit()* with the device in the *XOFF* state.

```
        if (!netif_queue_stopped(dev)) {
```

Passing the sk_buff to the device driver

Back in the good ole days, this call was *dev->hard_start_xmit()*. Now a new layer has been injected primarily to deal with GSO. If the start works, then a jump is made to the output point.

```
            rc = 0;
            if (!dev_hard_start_xmit(skb, dev)) {
               HARD_TX_UNLOCK(dev);
               goto out;
            }
         }
```

Exception handling

Arrival here means that the interface is stopped or *dev_hard_start_xmit()* failed. Since this is alledgedly a virtual device, *that is a bad thing.*

```
         HARD_TX_UNLOCK(dev);
         if (net_ratelimit())
            printk(KERN_CRIT "Virtual device %s asks to "
                  "queue packet!\n", dev->name);
```

Lock conflict

Falling into this block means dev->xmit_lock_owner == cpu. If control reaches this point, then this cpu has re-entered the *tx* code with the xmit lock was being held by this cpu. One possible way for this to happen is for an interrupt to cause a transmission. The packet is dropped in this case.

```
          } else {
             /* Recursion is detected! It is possible,
              * unfortunately */
             if (net_ratelimit())
                printk(KERN_CRIT "Dead loop on virtual device "
                       "%s, fix it urgently!\n", dev->name);
          }
```

Arrival here appears to mean that the interface is *down*. In that case the queue lock is dropped and and so is the packet.

```
       }

       rc = -ENETDOWN;
       rcu_read_unlock_bh();

    out_kfree_skb:
       kfree_skb(skb);
       return rc;
    out:
       rcu_read_unlock_bh();
       return rc;
    }
```

The *dev_hard_start_xmit()* function

This function acts as an interface to the device level starter. It has two primary purposes dealing with the NIT devices and GSO. The NIT queue is the queue in which *dev_add_pack()* with packet type ETH_P_ALL live. Any packet *transmitted* on *this* machine is immediately delivered to all NIT handlers on this machine.

```
int dev_hard_start_xmit(struct sk_buff *skb,
                                  struct net_device *dev)
{
    if (likely(!skb->next)) {
```

Do the NIT queue if necessary.

```
       if (netdev_nit)
          dev_queue_xmit_nit(skb, dev);

       if (netif_needs_gso(dev, skb)) {
          if (unlikely(dev_gso_segment(skb)))
              goto out_kfree_skb;
          if (skb->next)
              goto gso;
       }

```

This is the call that passes the *skb* to the device driver.

```
       return dev->hard_start_xmit(skb, dev);
    }

```

Handling GSO

We will not pursue the details of GSO, but it looks as though things might get really ugly if the device queue becomes stopped in the middle of this..

Aha, later we will see that the *netdevice* has a nasty new pointer that points to the current *sk_buff* in a GSO chain.

```
gso:
    do {
        struct sk_buff *nskb = skb->next;
        int rc;

        skb->next = nskb->next;
        nskb->next = NULL;
        rc = dev->hard_start_xmit(nskb, dev);
        if (unlikely(rc)) {
            nskb->next = skb->next;
            skb->next = nskb;
            return rc;
         }
         if (unlikely(netif_queue_stopped(dev) && skb->next))
             return NETDEV_TX_BUSY;
    } while (skb->next);

    skb->destructor = DEV_GSO_CB(skb)->destructor;

out_kfree_skb:
    kfree_skb(skb);
    return 0;
}
```

Mapping IP *tos* to *skb->priority* to output queue number.

Although almost 100% of IP traffic is handled as "best effort" somewhere along its path, Linux provides a complicated framework that may be used by private networks to provide some level of *diffserv*.

Queue selection is historically tied to the IP type of service. The IP type of service is an 8 bit field that is transmitted in the IP header. The bits are mapped as follows:

PPPDTRCX

The PPP values represent a three bit integer having values from 0 (Routine) through 7 (Network Control), and they are ignored by the default Linux scheduler. Mapping of *tos* to *priority* uses the DTRC bits.

- *D* bit means minimize delay.
- *T* bit means maximize throughput.
- *R* bit means maximize realibilty.
- *C* bit means minmize cost.
- *X* bit is reserved and overloaded by Linux to specify that the destination host must be ONLINK.

Earlier military precedence values include:

- FLASH_OVERRIDE
- FLASH
- IMMEDIATE
- PRIORITY
- ROUTINE

These are not single bit values but are encoded as binary numbers in the high order 3 bits.

TOS and Precedence bit definitions

The bit definitions are in ip.h

```
#define IPTOS_TOS_MASK          0x1E
#define IPTOS_TOS(tos)          ((tos)&IPTOS_TOS_MASK)
#define IPTOS_LOWDELAY          0x10
#define IPTOS_THROUGHPUT        0x08
#define IPTOS_RELIABILITY       0x04
#define IPTOS_MINCOST           0x02
#define IPTOS_PREC_MASK         0xE0
#define IPTOS_PREC(tos)         ((tos)&IPTOS_PREC_MASK)
#define IPTOS_PREC_NETCONTROL           0xe0
#define IPTOS_PREC_INTERNETCONTROL      0xc0
#define IPTOS_PREC_CRITIC_ECP           0xa0
#define IPTOS_PREC_FLASHOVERRIDE        0x80
#define IPTOS_PREC_FLASH                0x60
#define IPTOS_PREC_IMMEDIATE            0x40
#define IPTOS_PREC_PRIORITY             0x20
#define IPTOS_PREC_ROUTINE              0x00
```

Linux packet *priority*

Its painful to construct a scheduling system based upon DTRC and precedence. How does one map that to a "you go in front of her rule?" Conversely, priority systems are easy to build. Packets with the same priority go FIFO and packets of higher priority preempt packets of lower priority. But even these don't work so well because they can starve low priority traffic all together.

Nevertheless, the basic priority queuing mechanism is based upon a numeric priority in the range

- 0 - bad
- 15 - excellent

The priority is associated with a socket and is stored in *sk->sk_priority.* As seen in UDP the value of *sk->sk_priority* is inherited by *skb->priority.*

Setting of *sk_priority*

The IP_TOS *setsockopt()* allows an application to set the *tos*. The *tos* lives in the *inet_sock*

```
static int do_ip_setsockopt(struct sock *sk, int level,
             int optname, char __user *optval, int optlen)
 :

      case IP_TOS:    /* This sets both TOS and Precedence */
           if (sk->sk_type == SOCK_STREAM) {
                val &= ~3;
                val |= inet->tos & 3;
           }
           if (IPTOS_PREC(val) >= IPTOS_PREC_CRITIC_ECP &&
                 !capable(CAP_NET_ADMIN)) {
                 err = -EPERM;
                 break;
           }
           if (inet->tos != val) {
               inet->tos = val;
               sk->sk_priority = rt_tos2priority(val);
               sk_dst_reset(sk);
           }
           break;
```

Mapping *tos* to *priority*

The value of *inet->tos* is mapped to *skb->priority* as follows. The IP_TOS() macro eliminates the RTO_ONLINK bit and PPP leaving DTRC which is shifted before the table lookup.

```
#define IPTOS_TOS_MASK          0x1E
#define IPTOS_TOS(tos)          ((tos)&IPTOS_TOS_MASK)
```

The *ip_tos_2prio[]* table is used to map the 16 possible values of DTRC to a priority number which is also constrained to the range 0 to 15.

```
static inline char rt_tos2priority(u8 tos)
{
    return ip_tos2prio[IPTOS_TOS(tos)>>1];
}
```

We will see that only those shown in *red* are normally used.

```
#define TC_PRIO_BESTEFFORT              0
#define TC_PRIO_FILLER                  1
#define TC_PRIO_BULK                    2
#define TC_PRIO_INTERACTIVE_BULK        4
#define TC_PRIO_INTERACTIVE             6
#define TC_PRIO_CONTROL                 7
#define TC_PRIO_MAX                     15
```

The following macro is used to fill in spots in the table on the next page (generally) for spots where the COST bit in the DTRC tos is on.

```
#define ECN_OR_COST(class)      TC_PRIO_##class
```

The *tos2prio* mapping table

Note that only priority values 0, 1, 2, 4, and 6 are used:
Pure DTRC map as follows:

D -> 6 // minimize delay
T -> 2 // maximize throughput
R -> 0 // maximize reliability
C -> 1 // minimize cost
DT -> 4 // minimize delay and maximize throughput

ToS classes D, DC, DR and DRC map to priority (6) which maps to queue 0 (the best one)
Tos classes 0, R, and RC map to priority (0) which maps to queue 1 (the middle one)
Tos class C maps to priority (1) which maps to queue 2 the worst one.
Tos classes T, TC, TR and TRC map to queue 2 the worst one.
The DT tos classes map to priority 4 which also maps to queue 1.
Priorities (1, 2) map to queue 2 (the worst one)

```
__u8 ip_tos2prio[16] = {             tos  pri tos
        TC_PRIO_BESTEFFORT,            0 -> 0   -
        ECN_OR_COST(FILLER),           1 -> 1   C
        TC_PRIO_BESTEFFORT,            2 -> 0   R
        ECN_OR_COST(BESTEFFORT),       3 -> 0   RC
        TC_PRIO_BULK,                  4 -> 2   T
        ECN_OR_COST(BULK),             5 -> 2   TC
        TC_PRIO_BULK,                  6 -> 2   TR
        ECN_OR_COST(BULK),             7 -> 2   TRC
        TC_PRIO_INTERACTIVE,           8 -> 6   D
        ECN_OR_COST(INTERACTIVE),      9 -> 6   DC
        TC_PRIO_INTERACTIVE,          10 -> 6   DR
        ECN_OR_COST(INTERACTIVE),     11 -> 6   DRC
        TC_PRIO_INTERACTIVE_BULK,     12 -> 4   DT
        ECN_OR_COST(INTERACTIVE_BULK), 13 -> 4   DTC
        TC_PRIO_INTERACTIVE_BULK,     14 -> 4   DTR
        ECN_OR_COST(INTERACTIVE_BULK) 15 -> 4   DTRC
```

Mapping priority to queue index

The *prio2band[ ]* array is used to map *skb->priority* to one of three output queues. The value of *skb->priority* is derived from the IP *tos* via the rt/ip_*tos2priority()* function. For standard Unix scheduling only the entries shown in blue are actually used.

```

static const u8 prio2band[TC_PRIO_MAX+1] =
  { 1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1 };

/* 3-band FIFO queue: old style, but should be a bit faster
  than generic prio+fifo combination.
*/

#define PFIFO_FAST_BANDS 3
```

Thus in the current 3 queue system, the *default* is to use the "middle" or best effort queue.

Priority	Queue	
0	1	Best effort
1	2	Bulk
2	2	Bulk
4	1	Best effort
6	0	Interactive

Enqueing the *sk_buff*

Generic Ethernet drivers do support the *enqueue* mechanism. For these drivers *q->enqueue* points to the function *pfifo_fast_enqueue()* which is defined in *net/sched/sch_generic.c*.
It used to be the case that the *qdisc->data* place holder represented a table of 3 *sk_buff_head* structures. Now the table is presumed to follow the Qdisc structure as an unnamed vraiable. Each *net_device* structure also holds the maximum transmit queue length in *dev->tx_queue_len.*

If that length is not presently exceeded, the standard *skb* helper function is used to add the *sk_buff* to the appropriate queue. For ethernet devices the value of *tx_queue_len* is set to *1000* packets in the function *ether_setup()*. This used to be 100 in kernel 2.4. Note that under heavy loads it is possible to drop a packet here before it even reaches the outgoing device driver! This situation can be produced by starting enough full rate UDP senders that the sum of their *wmem* capacity in packets exceeds 1000. At ye olde queue max of 100 that was easy to do, but it is much more challenging now. It would seem to be more reasonable to have the process generating the excess traffic sleep. However, since this code also runs in the context of an IRQ it is simply not possible to sleep here.

The queue lock *must* be held before calling this function so shortcuts are safe.

```
static int pfifo_fast_enqueue(struct sk_buff *skb,
                              struct Qdisc* qdisc)
{
   struct sk_buff_head *list = prio2list(skb, qdisc);

   if (skb_queue_len(list) < qdisc->dev->tx_queue_len) {
      qdisc->q.qlen++;
      return __qdisc_enqueue_tail(skb, qdisc, list);
   }

   return qdisc_drop(skb, qdisc);
}
```

Queue selection

The *qdisc_priv()* function returns the address of the correct queue. It uses the *qdisc_priv()* function to obtain the address of the unnamed array of list headers and the prio2band[] array shown on the previous array to find the correct list.

```
static inline struct sk_buff_head *prio2list(
                          struct sk_buff *skb,
                         struct Qdisc *qdisc)
{
   struct sk_buff_head *list = qdisc_priv(qdisc);
   return list + prio2band[skb->priority & TC_PRIO_MAX];
}
```

This function returns address of the unnamed table of sk_buff headers.

```
static inline void *qdisc_priv(struct Qdisc *q)
{
   return (char *) q + QDISC_ALIGN(sizeof(struct Qdisc));
}
```

If the queue is full, the packet is (possibly) dropped here. A reliable transport protocol holds the original copy of the packet and it will be retransmitted after timeout.

```
static inline int qdisc_drop(struct sk_buff *skb,
                              struct Qdisc *sch)
{
    kfree_skb(skb);
    sch->qstats.drops++;

    return NET_XMIT_DROP;
}
```

Interface state management

Interface states have been defined in *include/linux/netdevice.h.*

These are bit numbers of bits in the *state* element of the *net_device* structure. The `__LINK_STATE_QDISC_RUNNING` bit is used to serialize execution of the *__qdisc_run* function.

```

/* These flag bits are private to the generic network queueing
 * layer, they may not be explicitly referenced by any other
 * code.
 */

enum netdev_state_t
{
        __LINK_STATE_XOFF=0,
        __LINK_STATE_START,
        __LINK_STATE_PRESENT,
        __LINK_STATE_SCHED,
        __LINK_STATE_NOCARRIER,
        __LINK_STATE_RX_SCHED,
        __LINK_STATE_LINKWATCH_PENDING,
        __LINK_STATE_DORMANT,
        __LINK_STATE_QDISC_RUNNING,
};
```

Consuming packets from the device output queues

The *qdisc_run()* wrapper makes sure that

- the queue is not stopped and
- *qdisc_run()* is not already active on this device on another CPU

```
static inline void qdisc_run(struct net_device *dev)
{
   if (!netif_queue_stopped(dev) &&
   !test_and_set_bit(__LINK_STATE_QDISC_RUNNING, &dev->state))
      __qdisc_run(dev);
}
```

The *__qdisc_run()* function, defined in *include/net/pkt_sched.h*, continually invokes *qdisc_restart()* while the interface is not stopped and while *qdisc_restart* indicates that the *queue is not empty by returning a value < 0.* Each call to *qdisc_restart()* results in *one packet* being passed to the device driver. Modern NICs commonly have hardware queuing facilities that are capable of storing tens of packets. When the hardware queue of the NIC is full, the device driver will call *netif_stop_queue().*

The __*qdisc_run()* function

Note that for UDP this code runs in the context of the process that called *sendto()* but might also result in packets that have been enqueued by other processes being transmitted.

```

void __qdisc_run(struct net_device *dev)
{
   if (unlikely(dev->qdisc == &noop_qdisc))
       goto out;

   while (qdisc_restart(dev) < 0 && !netif_queue_stopped(dev))
                 /* NOTHING */;

out:
  clear_bit(__LINK_STATE_QDISC_RUNNING, &dev->state);
}
```

Dequeuing a packet and transmitting a packet.

The *qdisc_restart()* function, also defined in net/sched/sch_generic.c removes a packet from the device queue and passes it to the device driver. Normally this will be the packet that was just enqueued a microsecond ago!

```
static inline int qdisc_restart(struct net_device *dev)
{
   struct Qdisc *q = dev->qdisc;
   struct sk_buff *skb;

```

The *dev->gso_skb* field is a *hack-o-matic* temporary holding spot for the next packet in a GSO fragment chain. This is the head of a possible list of additional fragments and must necessarily have priority over ALL QUEUES.

The dequeue function associated with an ethernet device is *pfifo_fast_dequeue()*. *It will dequeue and return the oldest packet in the highest priority queue.*

```
   /* Dequeue packet */
   if (((skb = dev->gso_skb)) || ((skb = q->dequeue(q)))) {
      unsigned nolock = (dev->features & NETIF_F_LLTX);

      dev->gso_skb = NULL;
```

Arrival here means that a packet is available for transmission. The *trylock* function with try to obtain the device tx lock and will return 0 if it is successful.

```
        /*
         * When the driver has LLTX set it does its own locking
         * in start_xmit. No need to add additional overhead by
         * locking again. These checks are worth it because
         * even uncongested locks can be quite expensive.
         * The driver can do trylock like here too, in case
         * of lock congestion it should return -1 and the packet
         * will be requeued.
         */
        if (!nolock) {
           if (!netif_tx_trylock(dev)) {
```

Failure of *trylock* to get the *xmit_lock*

Arrival here indicates that the driver lock was held. As seen before it might be held by this CPU. Here the situation is portrayed as more serious than before!

```
           collision:
              /* So, someone grabbed the driver. */

              /* It may be transient configuration error,
                 when hard_start_xmit() recurses. We detect
                 it by checking xmit owner and drop the
                 packet when deadloop is detected.
              */
              if (dev->xmit_lock_owner == smp_processor_id()) {
                 kfree_skb(skb);
                 if (net_ratelimit())
                    printk(KERN_DEBUG "Dead loop on netdevice
                                %s, fix it urgently!\n", dev-
                 return -1;
              }
              __get_cpu_var(netdev_rx_stat).cpu_collision++;
              goto requeue;
           }
        }
```

Sending the packet on to the device driver

Arrival here means that the tx lock was successfully obtained. The queue lock is released and if the device is not stopped the *dev_hard_start_xmit* wrapper is called. It will eventually call *dev->hard_start_xmit()*.

```
      {
         /* And release queue */
         spin_unlock(&dev->queue_lock);

         if (!netif_queue_stopped(dev)) {
            int ret;

            ret = dev_hard_start_xmit(skb, dev);
```

If the driver accepted the packet, and returned "OK", then that means "keep 'em coming". So a -1 is returned to *__qdisc_run()*.

```
            if (ret == NETDEV_TX_OK) {
               if (!nolock) {
                  netif_tx_unlock(dev);
               }
               spin_lock(&dev->queue_lock);
               return -1;
            }
```

A lock conflict can occur in the device driver too if it is a *"nolock"* device. "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. Note that the use of NETIF_F_LLTX is deprecated. Don't use it for new drivers."

```
            if (ret == NETDEV_TX_LOCKED && nolock) {
               spin_lock(&dev->queue_lock);
               goto collision;
            }
         }

```

Arrival here means that the test on 136 for a stopped device failed. If the *dev* is stopped, release the driver lock and retake the queue lock

```
153             /* NETDEV_TX_BUSY - we need to requeue */
154             /* Release the driver */
155             if (!nolock) {
156                netif_tx_unlock(dev);
157             }
158             spin_lock(&dev->queue_lock);
159             q = dev->qdisc;
160          }
```

If the device lock was held it is necessary to requeue the packet and reschedule the execution of *qdisc_run* in the context of a softirq.

```
161
162          /* Device kicked us out :(
163             This is possible in three cases:
164
165             0. driver is locked
166             1. fastroute is enabled
167             2. device cannot determine busy state
168                before start of transmission (f.e. dialout)
169             3. device is buggy (ppp)
170           */
171
```

If the *sk_buff* has a non-zero *next* pointer here, this *must be* a GSO packet.

```
172 requeue:
173          if (skb->next)
174             dev->gso_skb = skb;
175          else
176             q->ops->requeue(skb, q);
177          netif_schedule(dev);
178          return 1;
179       }
```

Arrival here means that the *if* statement on line 89 found nothing to send.

```
180       BUG_ON((int) q->q.qlen < 0);
181       return q->q.qlen;
182 }
```

Dequeuing of packets

The *pfifo_fast_dequeue()* function searches the three queues in high priority order attempting to find an *skb* that has been enqueued.

```
static struct sk_buff *pfifo_fast_dequeue(struct Qdisc* qdisc)
{
   int prio;
   struct sk_buff_head *list = qdisc_priv(qdisc);

   for (prio = 0; prio < PFIFO_FAST_BANDS; prio++) {
      if (!skb_queue_empty(list + prio)) {
         qdisc->q.qlen--;
         return __qdisc_dequeue_head(qdisc, list + prio);
      }
   }

   return NULL;
}

static inline struct sk_buff *__qdisc_dequeue_head(
                                 struct Qdisc *sch,
                                 struct sk_buff_head *list)
{
   struct sk_buff *skb = __skb_dequeue(list);

   if (likely(skb != NULL))
       sch->qstats.backlog -= skb->len;

   return skb;
}
```

Interface state management

Interface states have been defined in *include/linux/netdevice.h*. The enum below identifies bits in the *dev->state* variable. The ones that are highlighted are relevant to this section.

```

/* These flag bits are private to the generic network queuing
 * layer, they may not be explicitly referenced by any other
 * code.
 */

enum netdev_state_t
{
        __LINK_STATE_XOFF=0,
        __LINK_STATE_START,
        __LINK_STATE_PRESENT,
        __LINK_STATE_SCHED,
        __LINK_STATE_NOCARRIER,
        __LINK_STATE_RX_SCHED,
        __LINK_STATE_LINKWATCH_PENDING,
        __LINK_STATE_DORMANT,
        __LINK_STATE_QDISC_RUNNING,
};
```

State management functions

A collection of functions, defined in *include/linux/netdevice.h* manage interface state. This one schedules the *tx_action* softirq if the device is *not* in the XOFF state.

```
static inline void netif_schedule(struct net_device *dev)
{
    if (!test_bit(__LINK_STATE_XOFF, &dev->state))
            __netif_schedule(dev);
}
```

This one clears the XOFF bit. It can be used when the device becomes ready to service requests for the first time.

```
static inline void netif_start_queue(struct net_device *dev)
{
    clear_bit(__LINK_STATE_XOFF, &dev->state);
}
```

If the device was in the XOFF state, this one will clear the XOFF bit and schedule the *tx_action* softirq. It is called by a device driver when the TX ring transitions out of the FULL state and the device transitions from XOFF to not XOFF.

```
static inline void netif_wake_queue(struct net_device *dev)
{
#ifdef CONFIG_NETPOLL_TRAP
    if (netpoll_trap())
        return;
#endif
    if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
            __netif_schedule(dev);
}
```

This one stops the device. It is called by the device driver when the TX ring becomes full.

```
static inline void netif_stop_queue(struct net_device *dev)
{
#ifdef CONFIG_NETPOLL_TRAP
   if (netpoll_trap())
      return;
#endif
  set_bit(__LINK_STATE_XOFF, &dev->state);
}
```

This one is used to test to see if the device is presently accepting new start TX requests. It is used by the *dev* layer.

```
static inline int netif_queue_stopped(struct net_device *dev)
{
  return test_bit(__LINK_STATE_XOFF, &dev->state);
}

```

This one is used to see if the device is up and running yet. In contrast to the transitions between XOFF and !XOFF, the transition between START and !START is a very rare event.

```
static inline int netif_running(const struct net_device *dev)
{
   return test_bit(__LINK_STATE_START, &dev->state);
}
```

Freeing transmitted *sk_buffs* and refilling the Tx Ring

When a packet transmit operation completes, the NIC raises an interrupt and the device driver's interrupt handler is invoked. At this point it is necessary to release the *sk_buff*, and, if packets remain queued for the device, to use them to fill newly available slots in the Tx ring.

The following code from the *3c59x* driver releases the transmitted *sk_buff* and if there is space available in the Tx ring calls *netif_wake_queue()*.

```
dev_kfree_skb_irq(skb);
:
vp->dirty_tx = dirty_tx;
if (vp->cur_tx - dirty_tx <= TX_RING_SIZE - 1) {
     if (vortex_debug > 6)
           printk(KERN_DEBUG "boomerang_interrupt: wake queue\n");
     netif_wake_queue (dev);
```

Releasing the *sk_buff*

An important objective of OS design is to maximize responsiveness to hardware interrupts by minimizing the amount of time spent in hardware interrupt handling. The *dev_kfree_skb_irq()* function, defined in *include/linux/netdevice.h* is designed to facilitate this objective. Each CPU has a *softnet_data* structure that contains a pointer to a *completion_queue* of *sk_buffs* that have completed transmission and whose Tx complete interrupt has been handled on this CPU. The *output_queue* is a list of net devices which are in the stopped state with non-empty dev level queues.

```
/*
 * Incoming packets are placed on per-cpu queues so that
 * no locking is needed.
 */

struct softnet_data
{
   struct net_device       *output_queue;
   struct sk_buff_head     input_pkt_queue;
   struct list_head        poll_list;
   struct sk_buff          *completion_queue;

   struct net_device       backlog_dev;    /* Sorry. 8) */
#ifdef CONFIG_NET_DMA
   struct dma_chan         *net_dma;
#endif
};
```

This structure and the code is actually cleaned up some from 2.4. Here is the old version:

```
struct softnet_data
{
     int                     throttle; /* forces pkt drops */
     int                     cng_level; /* from prev page */
     int                     avg_blog;
    struct sk_buff_head     input_pkt_queue;
    struct net_device       *output_queue;
    struct sk_buff          *completion_queue;
} __attribute__((__aligned__(SMP_CACHE_BYTES)));

extern struct softnet_data softnet_data[NR_CPUS];
```

The *dev_kfree_skb_irq()* function

The mission of *dev_kfree_skb_irq()* is to enqueue the buffer upon the completion queue of this CPU's softnet data structure, and raise the *softirq* that will eventually invoke the *net_tx_action()* function that will actually free the buffers.

```
static inline void dev_kfree_skb_irq(struct sk_buff *skb)
{
    if (atomic_dec_and_test(&skb->users)) {
        struct softnet_data *sd;
        unsigned long flags;

        local_irq_save(flags);
        sd = &__get_cpu_var(softnet_data);
```

Here the packet is enqueued on the per processor temporary holding queue.

```
        skb->next = sd->completion_queue;
        sd->completion_queue = skb;
```

The process completes by raising the TX_SOFTIRQ. The softirq will be scheduled later and perform the actual freeing of the packet.

```
        raise_softirq_irqoff(NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

Refilling the Tx Ring

The *netif_wake_queue()* function is defined in *linux/include/netdevice.h.* It clears the bit that indicates that the device is in the stopped state, and if the device was previously in the stop state it invokes *__netif_schedule().*

```
static inline void netif_wake_queue(struct net_device *dev)
{
#ifdef CONFIG_NETPOLL_TRAP
   if (netpoll_trap())
       return;
#endif
   if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
        __netif_schedule(dev);
}
```

The *__netif_schedule()* function

The *__netif_schedule()* function is also defined in *linux/include/netdevice.h.* It unconditionally sets the bit that indicates that the interface is in the the scheduled state and if the bit was previously not in the scheduled state, it enqueues the *net_device* structure on the *output_queue* of the current CPU and then raises the *NET_TX_SOFTIRQ* softirq.

(Note that this was already done in *dev_kfree_skb_irq())* The *next_sched* field in the *net_device* structure is used to link the *net_devices* that are on the queue.

```
void __netif_schedule(struct net_device *dev)
{
   if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
      unsigned long flags;
      struct softnet_data *sd;

      local_irq_save(flags);
      sd = &__get_cpu_var(softnet_data);
```

Note that the scheduling appears to be LCFS.

```
      dev->next_sched = sd->output_queue;
      sd->output_queue = dev;
      raise_softirq_irqoff(NET_TX_SOFTIRQ);
      local_irq_restore(flags);
}

```

Freeing the *sk_buffs* on the *completion_queue*

The raising of the softirq eventually (via a mechanism discussed in the *devrecv* section) causes the *net_tx_action()* function defined in *net/core/dev.c* to be invoked in the context of the softirq.

This function has two primary missions:

- It performs the actual freeing of the buffers that have been placed on the *completion_queue* for this CPU.
- It invokes *qdisc_run()* on all of the *net_devices* that are on the *output_queue* for the CPU.

```
static void net_tx_action(struct softirq_action *h)
{
   struct softnet_data *sd = &__get_cpu_var(softnet_data);

   if (sd->completion_queue) {
       struct sk_buff *clist;

```

Note how disabled time is kept to an absolute minimum by the technique of *queue stealing.*

```
       local_irq_disable();
       clist = sd->completion_queue;
       sd->completion_queue = NULL;
       local_irq_enable();

       while (clist) {
          struct sk_buff *skb = clist;
          clist = clist->next;

          BUG_TRAP(!atomic_read(&skb->users));
         __kfree_skb(skb);
       }
   }
```

Redriving the interfaces on the *ouptut queue.*

In this section, *qdisc_run()* is invoked for each *net_device* on the *output_queue* for which the device's *queue_lock* can be obtained. If the lock is held, then *netif_schedule()* is called instead.

```
    if (sd->output_queue) {
        struct net_device *head;

```

Note the clever *queue stealing* strategy used again here to minimize disabled time.

```
        local_irq_disable();
        head = sd->output_queue;
        sd->output_queue = NULL;
        local_irq_enable();

```

Serially service each *net_device (head)* on the queue.

```
        while (head) {
           struct net_device *dev = head;
           head = head->next_sched;

```

Indicate that this device no longer has *net_tx_action()* pending.

```
           smp_mb__before_clear_bit();
           clear_bit(__LINK_STATE_SCHED, &dev->state);

```

If the queue lock is free, take the lock and try to send some packets.

```
            if (spin_trylock(&dev->queue_lock)) {
                 qdisc_run(dev);
                spin_unlock(&dev->queue_lock);
```

Otheriise *netif_schedule()* calls *__netif_schedule()* which puts the *net_device* structure back on the completion queue and reschedules the *softirq*.

```
            } else {
                netif_schedule(dev);
            }
        }
     }
  }
```