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:

```c
26 struct Qdisc
27 {
28    int (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
29    struct sk_buff * (*dequeue)(struct Qdisc *dev);
30    unsigned flags;
31    #define TCQ_F_BUILTIN   1
32    #define TCQ_F_THROTTLED 2
33    #define TCQ_F_INGRESS   4
34    int padded;
35    struct Qdisc_ops *ops;
36    u32 handle;
37    u32 parent;
38    atomic_t refcnt;
39    struct sk_buff_head q;
40    struct net_device *dev;
41    struct list_head list;
42
43    struct gnet_stats_basic bstats;
44    struct gnet_stats_queue qstats;
45    struct gnet_stats_rate_est rate_est;
46    spinlock_t *stats_lock;
47    struct rcu_head q_rcu;
48    int (*reshape_fail)(struct sk_buff *skb,
49                          struct Qdisc *q);
50
51    /* This field is deprecated, but it is still used by CBQ
52       * and it will live until better solution will be invented.
53       */
54    struct Qdisc *__parent;
55};
```
Structure elements are used as follows:

**enqueue:**
This function enqueues an `skbuff` 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.

```c
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 \textit{sk_buff} linear. If that doesn't work the packet is dropped.

```c
1430    if (skb_shinfo(skb)->frag_list &&
1431           !(dev->features & NETIF_F_FRAGLIST) &&
1432           __skb_linearize(skb))
1433       goto out_kfree_skb;
1434
1435    /* Fragmented skb is linearized if device does not support SG,
1436     * or if at least one of fragments is in highmem and device
1437     * does not support DMA from it.
1438     */
1439    if (skb_shinfo(skb)->nr_frags &&
1440           (!(dev->features & NETIF_F_SG) ||
1441             illegal_highdma(dev, skb)) &&
1442           __skb_linearize(skb))
1443       goto out_kfree_skb;
1444
1445```


Checksum computation

Checkums, 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.

\[(u16*)(skb->h.raw + skb->csum) = csum_fold(csum);\]

1444  /* If packet is not checksummed and device does not support
1445     * checksumming for this protocol, complete checksumming here.
1446     */
1447    if (skb->ip_summed == CHECKSUM_HW &&
1448        (!(dev->features & NETIF_F_GEN_CSUM) &&
1449         (!(dev->features & NETIF_F_IP_CSUM) ||
1450          skb->protocol != htons(ETH_P_IP)))))
1451         if (skb_checksum_help(skb, 0))
1452             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:  
1455    spin_lock_prefetch(&dev->queue_lock);
1456  
1457    /* Disable soft irqs for various locks below. Also
1458     * stops preemption for RCU.
1459     */
1460    rcu_read_lock_bh();
1461  
1462    /* Updates of qdisc are serialized by queue_lock.
1463    * The struct Qdisc which is pointed to by qdisc is now a
1464    * rcu structure - it may be accessed without acquiring
1465    * a lock (but the structure may be stale.) The freeing of
1466    * qdisc will be deferred until it's known that there are
1467    * more references to it.
1468    *
1469    * If the qdisc has an enqueue function, we still need to
1470    * hold the queue_lock before calling it, since queue_lock
1471    * also serializes access to the device queue.
1472    */
```
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      q = rcu_dereference(dev->qdisc);
1474      #ifdef CONFIG_NET_CLS_ACT
1475          skb->tc_verd = SET_TC_AT(skb->tc_verd, AT_EGRESS);
1476      #endif
1477      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);
1484          qdisc_run(dev);
1485          spin_unlock(&dev->queue_lock);
1486      }
```

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.

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

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

```
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()`.

```c
1493    /* The device has no queue. Common case for software devices:
1494       loopback, all the sorts of tunnels...
1495
1496       Really, it is unlikely that netif_tx_lock protection is necessary
1497       here. (f.e. loopback and IP tunnels are clean ignoring statistics
1498       counters.)
1499       However, it is possible, that they rely on protection
1500       made by us here.
1501
1502       Check this and shot the lock. It is not prone from deadlocks.
1503       Either shot noqueue qdisc, it is even simpler 8)
1504 */
1505    if (dev->flags & IFF_UP) {
1506     int cpu = smp_processor_id(); /* ok because BHs are
1507                  off */
1508     }
```

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

```c
1510      HARD_TX_LOCK(dev, cpu);
1511
```

The `HARD_TX_LOCK` macro operates as shown.

```c
1382 #define HARD_TX_LOCK(dev, cpu) { \
1383     if ((dev->features & NETIF_F_LLTX) == 0) { \
1384         netif_tx_lock(dev); \
1385     } \
1386 }
1387
916 static inline void netif_tx_lock(struct net_device *dev) \
917 { \
918     spin_lock(&dev->xmit_lock);
919     dev->xmit_lock_owner = smp_processor_id();
920 }
```
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.

1512          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.

1513             rc = 0;
1514             if (!dev_hard_start_xmit(skb, dev)) {
1515                 HARD_TX_UNLOCK(dev);
1516                 goto out;
1517             }
1518         }

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.

1519         HARD_TX_UNLOCK(dev);
1520         if (net_ratelimit())
1521             printk(KERN_CRIT "Virtual device %s asks to "
1522                   "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.

```c
1523 } else {
1524     /* Recursion is detected! It is possible,
1525        * unfortunately */
1526     if (net_ratelimit())
1527         printk(KERN_CRIT "Dead loop on virtual device \
1528           "%s, fix it urgently!\n", dev->name);
1529 }
```

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

```c
1530 }
1531
1532 rc = -ENETDOWN;
1533 rcu_read_unlock_bh();
1534
1535 out_kfree_skb:
1536     kfree_skb(skb);
1537     return rc;
1538 out:
1539    rcu_read_unlock_bh();
1540    return rc;
1541 }
1542
1543
1544
```

10
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.

```c
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev)
```

Do the NIT queue if necessary.

```c
    if (likely(!skb->next)) {
        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.

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

```c
```
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.

```c
1358 gso:
1359   do {
1360       struct sk_buff *nskb = skb->next;
1361       int rc;
1362
1363       skb->next = nskb->next;
1364       nskb->next = NULL;
1365       rc = dev->hard_start_xmit(nskb, dev);
1366       if (unlikely(rc)) {
1367           nskb->next = skb->next;
1368           skb->next = nskb;
1369           return rc;
1370       }
1371       if (unlikely(netif_queue_stopped(dev) && skb->next))
1372           return NETDEV_TX_BUSY;
1373   } while (skb->next);
1374
1375   skb->destructor = DEV_GSO_CB(skb)->destructor;
1376
1377 out_kfree_skb:
1378   kfree_skb(skb);
1379   return 0;
1380 }
```
Mapping IP 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 minimize 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

```c
#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

It's 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 \textit{sk->sk_priority}. As seen in UDP the value of \textit{sk->sk_priority} is inherited by \textit{skb->priority}. 
Setting of \textit{sk\_priority}

The IP\_TOS \textit{setsockopt()} allows an application to set the \textit{tos}. The \textit{tos} lives in the \textit{inet\_sock}.

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

509         case IP_TOS: /* This sets both TOS and Precedence */
510             if (sk->sk_type == SOCK_STREAM) {
511                 val &= ~3;
512                 val |= inet->tos & 3;
513             }
514             if (IPTOS_PREC(val) >= IPTOS_PREC_CRITIC_ECP &&
515                 !capable(CAP_NET_ADMIN)) {
516                 err = -EPERM;
517                 break;
518             }
519             if (inet->tos != val) {
520                 inet->tos = val;
521                 sk->sk_priority = rt_tos2priority(val);
522                 sk_dst_reset(sk);
523             }
524             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.

```
22 #define IPTOS_TOS_MASK          0x1E
22 #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.

```
155 static inline char rt_tos2priority(u8 tos)
156 {
157    return ip_tos2prio[IPTOS_TOS(tos)>>1];
158 }
```

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

```
17 #define TC_PRIO_BESTEFFORT 0
18 #define TC_PRIO_FILLER 1
19 #define TC_PRIO_BULK 2
20 #define TC_PRIO_INTERACTIVE_BULK 4
21 #define TC_PRIO_INTERACTIVE 6
22 #define TC_PRIO_CONTROL 7
24 #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.

```
170 #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)  

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

#define PFIFO_FAST_BANDS 3
```

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

<table>
<thead>
<tr>
<th>Priority</th>
<th>Queue</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

19
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 variable. 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.

```c
341 static int pfifo_fast_enqueue(struct sk_buff *skb,  
        struct Qdisc* qdisc)  
342 {  
    struct sk_buff_head *list = prio2list(skb, qdisc);  
344    if (skb_queue_len(list) < qdisc->dev->tx_queue_len) {  
        qdisc->q.qlen++;  
347        return __qdisc_enqueue_tail(skb, qdisc, list);  
348    }  
349    return qdisc_drop(skb, qdisc);  
351 }
```
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.

```c
334 static inline struct sk_buff_head *prio2list(
    struct sk_buff *skb,
    struct Qdisc *qdisc)
335 {
336    struct sk_buff_head *list = qdisc_priv(qdisc);
337    return list + prio2band[skb->priority & TC_PRIO_MAX];
338 }
```

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

```c
20 static inline void *qdisc_priv(struct Qdisc *q)
21 {
22    return (char *) q + QDISC_ALIGN(sizeof(struct Qdisc));
23 }
```

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.

```c
285 static inline int qdisc_drop(struct sk_buff *skb,
    struct Qdisc *sch)
286 {
287    kfree_skb(skb);
288    sch->qstats.drops++;
289    return NET_XMIT_DROP;
290 }
```
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.

```c
/* 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

```c
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 \texttt{__qdisc\_run()} function

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

183
184 void __qdisc\_run(struct net\_device *dev)
185 {
186    if (unlikely(dev->qdisc == &noop\_qdisc))
187        goto out;
188
189    while (qdisc\_restart(dev) < 0 && !netif\_queue\_stopped(dev))
190        /* NOTHING */;
191
192 out:
193    clear\_bit(__LINK\_STATE\_QDISC\_RUNNING, &dev->state);
194 }
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!

```c
91 static inline int qdisc_restart(struct net_device *dev) {
92    struct Qdisc *q = dev->qdisc;
93    struct sk_buff *skb;
94
96    /* Dequeue packet */
97    if (((skb = dev->gso_skb)) || ((skb = q->dequeue(q))) ) {
98       unsigned nolock = (dev->features & NETIF_F_LLTX);
99
100       dev->gso_skb = NULL;
```

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

102  /*
103  * When the driver has LLTX set it does its own locking
104  * in start_xmit. No need to add additional overhead by
105  * locking again. These checks are worth it because
106  * even uncongested locks can be quite expensive.
107  * The driver can do trylock like here too, in case
108  * of lock congestion it should return -1 and the packet
109  * will be requeued.
110  */
111  if (!nolock) {
112      if (!
26

\textbf{Failure of \textit{trylock} to get the \textit{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!

113          collision:
114          /* So, someone grabbed the driver. */
115
116          /* It may be transient configuration error,
117             when hard_start_xmit() recurses. We detect
118             it by checking xmit owner and drop the
119             packet when deadloop is detected.
120          */
121          if (dev->xmit_lock_owner == smp_processor_id()) {
122              kfree_skb(skb);
123              if (net_ratelimit())
124                  printk(KERN_DEBUG "Dead loop on netdevice
125                  %s, fix it urgently!\n", dev-
126                  return -1;
127          }
128          __get_cpu_var(netdev_rx_stat).cpu_collision++;
129          goto requeue;
130      }
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()`.

```c
    { /* 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()`.

```c
    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."

```c
    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.

```c
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.

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

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

```c
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.

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

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

```c
353 static struct sk_buff *pfifo_fast_dequeue(struct Qdisc* qdisc)
354 {  
355    int prio;
356    struct sk_buff_head *list = qdisc_priv(qdisc);
357
358    for (prio = 0; prio < PFIFO_FAST_BANDS; prio++) {
359      if (!skb_queue_empty(list + prio)) {
360        qdisc->q.qlen--;
361        return __qdisc_dequeue_head(qdisc, list + prio);
362      }
363    }
364    return NULL;
365 }
```

```c
203 static inline struct sk_buff *__qdisc_dequeue_head(
204     struct Qdisc *sch,
205     struct sk_buff_head *list)
206 {  
207     struct sk_buff *skb = __skb_dequeue(list);
208     if (likely(skb != NULL))
209       sch->qstats.backlog -= skb->len;
210     return skb;
211 }
```
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.

```c
/* 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.

```c
628 static inline void netif_schedule(struct net_device *dev)
629 {
630    if (!test_bit(__LINK_STATE_XOFF, &dev->state))
631        __netif_schedule(dev);
632 }
```

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

```c
634 static inline void netif_start_queue(struct net_device *dev)
635 {
636    clear_bit(__LINK_STATE_XOFF, &dev->state);
637 }
```

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.

```c
639 static inline void netif_wake_queue(struct net_device *dev)
640 {
641    #ifdef CONFIG_NETPOLL_TRAP
642        if (netpoll_trap())
643            return;
644    #endif
645    if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
646        __netif_schedule(dev);
647 }
```
This one stops the device. It is called by the device driver when the TX ring becomes full.

```c
649 static inline void netif_stop_queue(struct net_device *dev) 650 { 651 #ifdef CONFIG_NETPOLL_TRAP 652 if (netpoll_trap()) 653 return; 654 #endif 655 set_bit(__LINK_STATE_XOFF, &dev->state); 656 }
```

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.

```c
658 static inline int netif_queue_stopped(struct net_device *dev) 659 { 660 return test_bit(__LINK_STATE_XOFF, &dev->state); 661 }
```

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.

```c
663 static inline int netif_running(const struct net_device *dev) 664 { 665 return test_bit(__LINK_STATE_START, &dev->state); 666 }
```
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().

```
2277   dev_kfree_skb_irq(skb);
2285   vp->dirty_tx = dirty_tx;
2286   if (vp->cur_tx - dirty_tx <= TX_RING_SIZE - 1) {
2287       if (vortex_debug > 6)
2288           printk(KERN_DEBUG "boomerang_interrupt: wake
2289                queue\n");
2289   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.

604/*
605 * Incoming packets are placed on per-cpu queues so that
606 * no locking is needed.
607 */
608
609 struct softnet_data
610 {
611    struct net_device *output_queue;
612    struct sk_buff_head input_pkt_queue;
613    struct list_head poll_list;
614    struct sk_buff *completion_queue;
615
616    struct net_device backlog_dev; /* Sorry. 8) */
617 #ifdef CONFIG_NET_DMA
618    struct dma_chan *net_dma;
619 #endif
620 };

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

473 struct softnet_data
474 {
475    int throttle; /* forces pkt drops */
476    int cng_level; /* from prev page */
477    int avg_blog;
478    struct sk_buff_head input_pkt_queue;
479    struct net_device *output_queue;
480    struct sk_buff *completion_queue;
481 } __attribute__((__aligned__((SMP_CACHE_BYTES))));

484 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.

```c
672 static inline void dev_kfree_skb_irq(struct sk_buff *skb) {
673    if (atomic_dec_and_test(&skb->users)) {
674        struct softnet_data *sd;
675        unsigned long flags;
676
678        local_irq_save(flags);
679        sd = &__get_cpu_var(softnet_data);
680        skb->next = sd->completion_queue;
681        sd->completion_queue = skb;
682        raise_softirq_irqoff(NET_TX_SOFTIRQ);
683        local_irq_restore(flags);
684    }
685}
```

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

```c
680    skb->next = sd->completion_queue;
681    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.

```c
682    raise_softirq_irqoff(NET_TX_SOFTIRQ);
683    local_irq_restore(flags);
684}
```
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()`.

```c
639 static inline void netif_wake_queue(struct net_device *dev) {
640    #ifdef CONFIG_NETPOLL_TRAP
641        if (netpoll_trap())
642            return;
643    #endif
644    if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
645        __netif_schedule(dev);
647 }
```
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.

```
1102 void __netif_schedule(struct net_device *dev)  
1103 {  
1104     if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {  
1105         unsigned long flags;  
1106         struct softnet_data *sd;  
1107         local_irq_save(flags);  
1108         sd = &__get_cpu_var(softnet_data);  
1109     }  
1110     dev->next_sched = sd->output_queue;  
1111     sd->output_queue = dev;  
1112     raise_softirq_irqoff(NET_TX_SOFTIRQ);  
1113     local_irq_restore(flags);  
1114 }  
1115
```
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.

```c
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;
        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);
        }
    }
}
```

Note how disabled time is kept to an absolute minimum by the technique of queue stealing.
Redriving the interfaces on the output queue.

In this section, \textit{qdisc\_run()} is invoked for each \textit{net\_device} on the \textit{output\_queue} for which the device’s \textit{queue\_lock} can be obtained. If the lock is held, then \textit{netif\_schedule()} is called instead.

```
1664    if (sd->output_queue) {
1665        struct net_device *head;
1666
1667        local_irq_disable();
1668        head = sd->output_queue;
1669        sd->output_queue = NULL;
1670        local_irq_enable();
```

Note the clever \textit{queue stealing} strategy used again here to minimize disabled time.

```
1667        while (head) {
1668            struct net_device *dev = head;
1669            head = head->next_sched;
```

Indicate that this device no longer has \textit{net\_tx\_action()} pending.

```
1676        smp_mb__before_clear_bit();
1677        clear_bit(__LINK\_STATE\_SCHED, &dev->state);
1678```
If the queue lock is free, take the lock and try to send some packets.

1679       if (spin_trylock(&dev->queue_lock)) {
1680           qdisc_run(dev);
1681           spin_unlock(&dev->queue_lock);

Otherwise \texttt{netif\_schedule()} calls \_\texttt{netif\_schedule()} which puts the \texttt{net\_device} structure back on the completion queue and reschedules the \texttt{softirq}.

1682             } else {
1683                 netif_schedule(dev);
1684             }