TRANSMITTING A FRAME

Comparison of terminologies used with that of functions present in receiving a frame:

NET_TX_SOFTIRQ is the flag used for sending a frame similar to *NET_RX_SOFTIRQ*
Net_tx_action is the counterpart of *net_rx_action*.
Output_queue is the list of devices that have something to transmit

```
struct net_device      *output_queue;
```

dev_queue_xmit is for the egress path and *netif_rx* is for the ingress path: each transfers one frame between the driver's buffer and the kernel's queue.
Net_tx_action is called when the device wants to transmit something.
When a device is scheduled for reception, its *_ _LINK_STATE_RX_SCHED* flag is set.
When a device is scheduled for transmission, its *_ _LINK_STATE_SCHED* flag is set.

ENABLING TRANSMISSIONS

__LINK_STATE_XOFF which is present in the member state of *net_device* structure determines whether a device can be scheduled for transmission or not. The flag must be off inorder for the device to be considered for transmission.

```
struct net_device
{
..
unsigned long         state;
..
}
```

Function: netif_start_queue
File: include/linux/netdevice.h

This function enables a device for transmission.

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

Function: __netif_schedule
File: include/linux/netdevice.h

If the *dev_queue_xmit* function is not able to transmit for some reasons such as egress queue is full, lock on the queue is taken, no memory etc it schedules the packet for transmission after sometime using the *__netif_schedule* function. There is one *output_queue* for each CPU. This function adds the device to the head of the *output_queue* list. If the device has already been scheduled for transmission, then the function won't modify anything.

test_and_set_bit(__LINK_STATE_SCHED, &dev->state)

__LINK_STATE_SCHED is used to mark devices that are in the *output_queue* list because they have something to send. __netif_schedule is called by netif_wake_queue which enables the transmission for a device if the device was previously disabled.

ACTUAL TRAMISSION

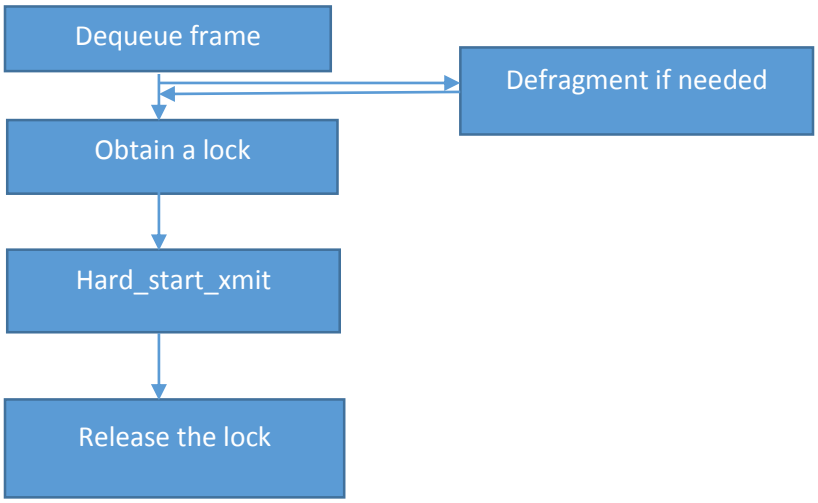
Function: outsl
File: include/asm-ia64/io.h

This function actually writes into the memory mapped IP of the device.

#define outsl(p,s,c) __outsl(p,s,c)

Function: dev_queue_xmit
File: net/core/dev.c
Parameters
*struct sk_buff *skb – Pointer to socket buffer structure.*

Figure illustrating the working of dev_queue_xmit function



For each packet to transmit from the IP layer, the dev_queue_xmit() procedure is called. The L3 layer always calls this function when transmitting a packet, regardless of the kind of device or L2 and L3 protocols used. It queues a packet in the *qdisc* associated to the output interface. It queues a buffer for transmission to a networked device. This function dequeues a frame from the device's egress queue and feeds it to the device's *hard_start_xmit* method. The function receives a pointer to the socket buffer structure as the input. It contains all the necessary information such as

skb->dev = outgoing device
skb->data = beginning of the payload
skb->len = length of the payload

Similarly other information such as next hop, ip address etc, everything is available is in this structure. In the beginning it makes several sanity checks.
It checks if the frame is composed of fragments. If this pointer is NULL, it means that data to be transmitted is a single block instead of fragments.

if (skb_shinfo(skb)->frag_list)

The defragmentation of the packet is done by *__skb_linearize*.

__skb_linearize(skb, GFP_ATOMIC)

Once these works are done, we try to fetch the queue of the devices. Devices can be of two types. Queue enabled devices and Queueless devices. Queueless devices are software devices like loopback. If the device has a queue, it enqueues it for *hard_start_xmit* to actually transmit the data.
It checks if the interface to be transmitted is up

if (dev->flags & IFF_UP)

When a packet is ready for transmission on the networking hardware, it is handed to the hard_start_xmit function pointer of the net_device data structure.

*Int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);*

Once the card has sent a packet or a group of packets, it communicates to the CPU that packets have been sent out. The CPU uses this information in *net_tx_action()* to put the packets into a completion_queue and to schedule a softirq for later deallocating the memory associated with these packets

Function: net_tx_action
File: net/core/dev.c
Parameters
*struct softirq_action *h – Pointer to the action. In this case NET_TX_SOFTIRQ*

This can be triggered with *raise_softirq_irqoff(NET_TX_SOFTIRQ)* by devices in two different contexts.

Netif_schedule - netif_schedule schedules a device for transmission
netif_wake_queue – enables transmission on a device and schedules the device for transmission.
dev_kfree_skb_irq – When a transmission has been completed.

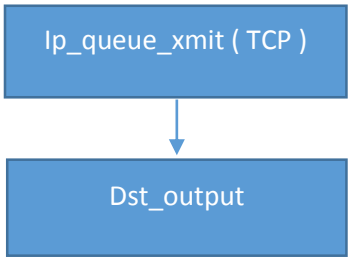
If there is anything already in the completion queue, then it is deallocated to make room for the next packets. Similarly if there is anything in the output queue, it is tried to transmit. Whenever a device is scheduled for transmission, the next frame to transmit is selected by the *qdisc_run* function, which indirectly calls the dequeue virtual function of the associated queuing discipline.

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

If it does not get the lock, then it is scheduled for later processing.

```
netif_schedule(dev);
```

L3 SUBSYSTEM



Function: ip_finish_output2
File: net/ipv4/ip_output.c
Parameters:
*struct sk_buff *skb – Pointer to socket buffer*

Packet transmission in IPV4 sub-system ends with the call to ip_finish_output2. The skb buffer input to *ip_finish_output2*, includes the packet data, along with information such as the device to use for transmission and the routing table cache entry (dst) that was used by the kernel to make the forwarding decision. It checks if it has a cached L2 header.

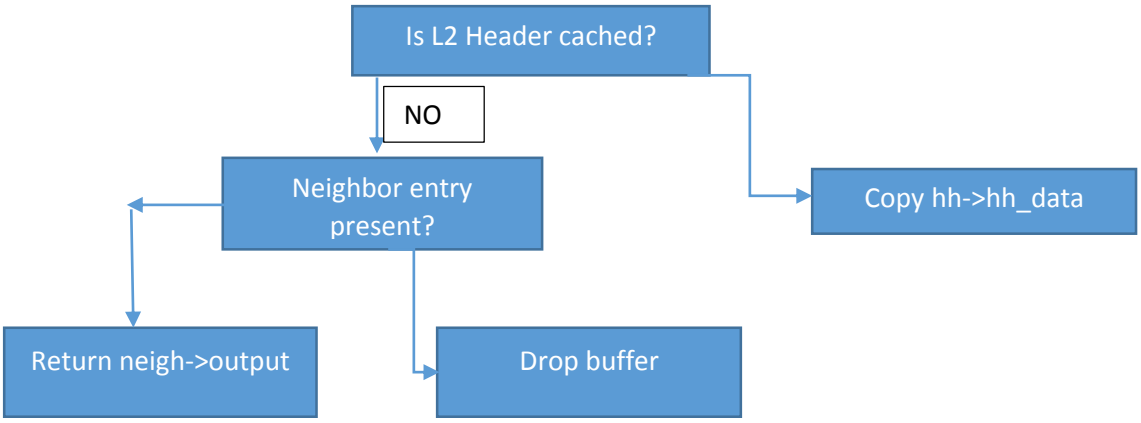
```
struct hh_cache *hh = dst->hh;
```

If it is present, it is copied into the skb buffer. If no cached entry is available, neigh->output method.

```
return dst->neighbour->output(skb);
```

neigh->output will initially be initialized to *neigh_resolve_output*, which will put the packet in the arp_queue queue, try to resolve the address by sending a solicitation request, and wait until the solicitation reply arrives, whereupon the packet is transmitted.

Figure illustrating the working of ip_finish_output2 function



Function: ip_queue_xmit
File: net/ipv4/ip_output.c
Parameters
*struct sk_buff *skb – Pointer to socket buffer structure*
ipfragok – Whether okay to perform fragmentation

The pointer to the L4 data is got through the following statement.

```
struct sock *sk = skb->sk;
```

If the buffer is already assigned the proper routing information (skb->dst), there is no need to consult the routing table.

```
rt = (struct rtable *) skb->dst;
if (rt != NULL)
goto packet_routed;
```

The function checks if a route has already been cached in a socket structure and if so it checks if the route is still valid.

```
rt = (struct rtable *)__sk_dst_check(sk, 0);
```

If the socket does not already have a route, it looks for a new route with *ip_route_output_flow*. The packet is dropped if *ip_route_output_flow* fails. If the route is found, it is stored in the sk data structure so that it can be used directly next time, and the routing table does not have to be consulted again. If for some reason the route is invalidated again, a future.
Till this point, skb contains only the IP payload—generally the header and payload from the L4 layer (TCP). When *ip_queue_xmit* receives skb, *skb->data* points to the beginning of the L3 payload, which is where the L4 protocol writes its own data. The L3 header lies before this

pointer. So *skb_push* is used here to move skb->data back so that it points to the beginning of the L3 or IPheader;

```
iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) + (opt ? opt->optlen : 0));
```

Then the IP options are processed. For example, if the don't fragment bit is set, then the following statement executes.

```
iph->frag_off = htons(IP_DF);
```

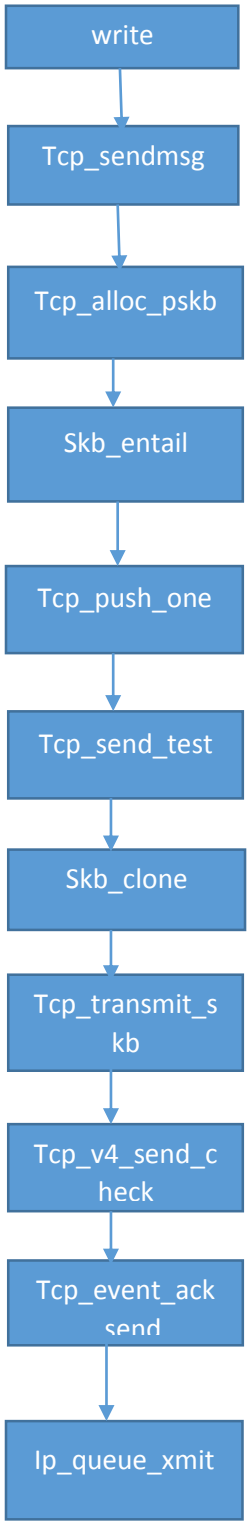
skb->priority is used by Traffic Control to decide which one of the outgoing queues to enqueue the packet in; this in turn helps determine how soon it will be transmitted.

```
skb->priority = sk->sk_priority;
```

Then the dst->output is called. The handler for this function is already registered at boot time

```
int (*input)(struct sk_buff*);
int (*output)(struct sk_buff*);
```

L4 SUBSYSTEM



Function: tcp_sendmsg
File: net/ipv4/tcp.c

When an application wants to write data over a TCP socket, *tcp_sendmsg()* is called inside the kernel. *sk_stream_wait_connect* function determines if the socket is still in connect state. If so, it waits till *connect* state ends. TCP receives data from an application and segments the data into pieces. This segmentation is necessary so that the information can be placed inside the TCP data field. By segmenting the data, TCP creates chunks of data that can be routed separately over whatever connections are needed in order to reach the destination. Any of these segments can be retransmitted to replace the original segments that got lost or *damaged in transmission.*
TCP sends out data in chunks of 1 mss. Maximum segment size is based on MTU, which is a link layer characteristic and can be retrieved from tcp_current_mss() .
There are two loops. The outer loop

```
(while (--iovlen >= 0))
```

accesses the next user buffer in each iteration, and the inner loop

```
(while (seglen > 0))
```

generates segments from each user buffer. The inner loop generates segments of size 1mss. It checks the last segment in the queue to be partial at any point of time. The last segment for the socket can be accessed from the *prev->field* of the queue head since it is a doubly linked link list.

```
skb = sk->sk_write_queue.prev;
```

If we don't find a partial segment in the transmit queue, we need to create a new segment for the user data. Before allocating memory for a new segment, we first check if the socket's quota for the send buffer has exceeded its limit by calling *tcp_memory_free()* . If we have enough memory, *tcp_alloc_pskb()* is called to allocate a new buffer for the TCP segment.
If everything is fine and memory can be allocated, then we queue the new segment at the tail of the transmit queue by calling *skb_entail()*. Else we wait for memory to be available.

```
if (!skb)
    goto wait_for_memory;
```

Linux implements a transmit and retransmit queue as a single queue *(tp → write_queue)*. *tp → send_head* marks the start of the transmit queue. We copy data to the identified segment by calling *skb_add_data()*. We copy the user data to a page through the following statement.

```
err = skb_copy_to_page(sk, from, skb, page,off, copy);
```

The routine is called to copy data from a user buffer from a specified offset within the page and account for the memory usage by the socket buffer.

In case we need to tell the receiver to push data to the application at the earliest, mark the push sequence number as a write sequence number by calling *tcp_mark_push()* and call *_tcp_push_pending_frames()* .

```
if (forced_push(tp)) {
      tcp_mark_push(tp, skb);
      __tcp_push_pending_frames(sk, tp, mss_now, TCP_NAGLE_PUSH);
}
```

If we can ' t force the data to be pushed and there is only one segment in the transmit queue, *tcp_push_one()* is called to push the segment from the transmit queue. This routine is called to send once a full - sized segment is ready for transmission and there is only one segment in the transmit queue.. If data is copied, the socket is released and the user process is notified of the completion.

```
if (copied)
      tcp_push(sk, tp, flags, mss_now, tp->nonagle);
TCP_CHECK_TIMER(sk);
release_sock(sk);
return copied;
```

Function: __tcp_push_pending_frames
File: net/ipv4/tcp_output.c
Parameters
*struct sock *sk – Pointer to the socket buffer*
*struct tcp_sock *tp – Pointer to the tcp socket*
unsigned int cur_mss – Current chosen mss
int nonagle – Whether or not to use Nagle's alogirthm

This routine does all the work required to transmit TCP buffers queued up in the send queue so far. It checks if we have anything to transmit in the write_queue and Calls *tcp_write_xmit()* to try to send out segments.

Function: tcp_write_xmit
File: net/ipv4/tcp_output.c

This function tries to process all the TCP segments queued up at the socket's write queue one by one. It checks for each segment to determine whether we can send it out or not. The next packet to send out can be accessed from *tp → send_head.* It also checks if the current segment can be transmitted right now by the call *tcp_snd_test ().* If the current segment can be transmitted and if the length of the segment is larger than that can be transmitted, then it is fragmented by calling *tcp_fragment*. If the packet need not be

fragmented then it can be transmitted by the function all *tcp_transmit_skb*. A clone of the TCP segment is only passed to *tcp_transmit_skb* as it needs to wait till it gets the ACK. *tcp_transmit_skb()* actually builds TCP header, sends it to the IP layer for processing, and puts the final IP datagram on the device queue for hardware transmission. If the segment is sent successfully, it moves to the next segment by calling *update_send_head*.

References:

Input Routing
http://people.cs.clemson.edu/~westall/853/notes/routein.pdf

Deliver IP packet to transport layer
http://people.cs.clemson.edu/~westall/853/notes/iprecv3.pdf

Linux Networking
http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO-8.html

Path of packet in Linux Kernel
http://raghavclv.wordpress.com/article/network-packet-processing-in-linux-8m286fvf764g-5/

Path of packet in Linux Kernel Stack
http://hsnlab.tmit.bme.hu/twiki/pub/Targyak/Mar11Cikkek/Network_stack.pdf

"Understanding Linux Networking Internals"

"TCP/IP Architecture, Implementation and Design in Linux"

Linux Networking Kernel
http://www.ecsl.cs.sunysb.edu/elibrary/linux/network/LinuxKernel.pdf